30th Euromicro Conference on Real-Time Systems

ECRTS 2018, July 3rd–6th, 2018, Barcelona, Spain

Edited by

Sebastian Altmeyer

LIPICS

Editor

Sebastian Altmeyer
University of Amsterdam
Amsterdam, The Netherlands
`altmeyer@uva.nl`

ACM Classification 2012
Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Software and its engineering → Real-time systems software, Software and its engineering → Real-time schedulability

ISBN 978-3-95977-075-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-075-0.

Publication date
June 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECRTS.2018.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface

Message from the Chairs

Welcome to the **30th Euromicro Conference on Real-Time Systems (ECRTS 2018)** in Barcelona, Spain. ECRTS is the premier conference in Europe in the broad area of real-time and embedded systems. Along with RTSS and RTAS, ECRTS ranks as one of the three top international conferences on real-time systems. For ECRTS 2018, we have received 78 **submissions** with authors from 21 countries, 9 (43%) from outside Europe.

Each submission has been reviewed by at least three members of the technical programm committee – all active researchers and experts in their field – with the help of 62 external reviewers. The submissions have been evaluated and assessed according to their contribution and originality, the technical correctness and writing quality. The program committee has then selected – at the physical program committee meeting in Amsterdam – 26 of these submissions for publication in the proceedings and presentation at the conference.

From the 26 accepted papers, three have been recognized as **Outstanding Papers** by the program committee and will be presented in a dedicated session. One of these three papers will be selected as **Best Paper** by a best paper committee based on both the contribution of the paper and the presentation.

ECRTS takes a leading role in adopting novel concepts and thus shaping the way we do science. In 2016, ECRTS has been the first conference on embedded real-time systems to introduce the **Artifact Evaluation**, with the aim to promote reproducibility of our research. An Artifact Evaluation committee validates the artifacts submitted by the authors and includes a seal of approval for those who passed the replication test. This year, already eight papers (31%) have been submitted to and passed the artifact evaluation and are marked with this seal in the proceedings. In 2017, ECRTS has been again the first conference on embedded real-time systems to introduce an **Open Access publication model**, while retaining the quality-control measures. The open access model has been established with LIPIcs – Leibniz International Proceedings in Informatics established in cooperation with **Schloss Dagstuhl, Leibniz Center for Informatics**. The conference serves the research community and the public best when results are accessible to the largest audience, i.e., the research community and the public. This year again, the proceedings will be accessible free of charge for everyone.

ECRTS 2018 will start with a **keynote on Runtime-Aware Architecture (RAA)** by **Mateo Valero**, director of the **Barcelona Supercomputing Center**. Another keynote will follow on the second day of the conference.

ECRTS 2018 will feature in addition a **Work-In-Progress session** for short papers where novel ideas will be presented to the audience, and a **Journal2Conference** session where work so far only published in journals can be presented to the conference audience. Submissions to the Work-In-Progress and Journal2Conference sessions have been evaluated separately by dedicated committees, and are not part of these published proceedings. Furthermore, a full presentation is dedicated to an **Industrial Challenge** to foster the collaboration between the academic world and industry.

The day before the main conference is dedicated to five outstanding international workshops: the Real-Time Scheduling Open Problems Seminar (**RTSOPS**), the workshop on

Worst-Case Execution Time Analysis (**WCET**), the workshop on Operating Systems Platforms for Embedded Real-Time Applications (**OSPERT**), the workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (**WATERS**), and the workshop Real-Time Networks (**RTN**).

ECRTS 2018 is the result of the hard work of many people. We are especially grateful for the contributions of the following people: the **Barcelona Supercomputing Center** for its support with the local organization, the **Program Committee** and the **external reviewers**, who are listed in subsequent pages; **Martina Maggio** as Chair of the Work-In-Progress session and the Artifact Evaluation Committee; **Patrick Meumeu Yomsi** as Chair of the Journal2Conference session; **Sophie Quinton** for the organization and **Arne Hamann** for the presentation of the Industrial Challenge; **Heechul Yun** and **Adam Lackorzynski** as OSPERT Workshop Chairs; **Mathieu Jan** and **Ramon Serna Olivier** as RTN Workshop Chairs; **Thidapat (Tam) Chantem** and **Dorin Maxim** as RTSOPS Workshop Chairs; **Claire Pagetti** and **Arne Hamann** as WATERS Workshop Chairs; **Forian Brandner** as WCET Workshop Chair. A special thanks to **Marc Herbstritt** of Dagstuhl Publishing and **Björn Brandenburg** with their support in publishing the proceeding, and to **Gerhard Fohler** for his steady guidance and contributions as the Euromicro Real-Time Technical Committee Chair.

Congratulations to all of the authors for their exceptional work. ECRTS 2018 would not exist without the contributions of the authors that submitted their work for review and critique. We are very pleased with the quality, depth, and breadth of this year's technical program. We hope you enjoy yourself at ECRTS 2018!

Francisco J. Cazorla
General Chair, ECRTS 2018

Sebastian Altmeyer
Program Chair, ECRTS 2018

Committees

General Chair

Francisco J. Cazorla, Barcelona Supercomputing Center and IIIA-CSIC, Spain

Program Chair

Sebastian Altmeyer, University of Amsterdam, The Netherlands

Real-Time Technical Committee Chair

Gerhard Fohler, TU Kaiserslautern, Germany

Artifact Evaluation Chairs

Martina Maggio, Lund University, Sweden

Program Committee

Benny Akesson, TNO-ESI, The Netherlands
James Anderson, University of North Carolina at Chapel Hill, USA
Karl-Erik Årzén, Lund University, Sweden
Patricia Balbastre, Universitat Politècnica de València, Spain
Sanjoy Baruah, Washington University in St. Louis, USA
Andrea Bastoni, SYSGO AG, Germany
Marko Bertogna, University of Modena, Italy
Reinder Bril, Technische Universiteit Eindhoven, The Netherlands
Tam Chantem, Virginia Tech, USA
Robert Davis, University of York, UK & INRIA-Paris, France
Rolf Ernst, TU Braunschweig, Germany
Nathan Fisher, Wayne State University, USA
Gerhard Fohler, TU Kaiserslautern, Germany
Christian Fraboul, Université de Toulouse / IRIT - INPT / ENSEEIHT, France
Christopher Gill, Washington University in St. Louis, USA
Steve Goddard, University of Nebraska-Lincoln, USA
Arne Hamann, Robert Bosch GmbH, Germany
Hermann Härtig, TU Dresden, Germany
George Lima, Federal University of Bahia, Brazil
Martina Maggio, Lund University, Sweden
Claire Maiza, Grenoble INP / Verimag, France
Julio Medina, Universidad de Cantabria, Spain
Patrick Meumeu Yomsi, CISTER/INESC-TEC, ISEP, Portugal
Geoffrey Nelissen, CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
Claire Pagetti, ONERA / IRIT-ENSEEIHT, France
Michael Paulitsch, Thales, Austria
Rodolfo Pellizzoni, University of Waterloo, Canada

Isabelle Puaut, University of Rennes / IRISA, France
Sophie Quinton, INRIA-Grenoble Rhône-Alpes, France
Jan Reineke, Saarland University, Germany
Christine Rochange, Université de Toulouse / IRIT, France
Jean-Luc Scharbarg, Université de Toulouse / IRIT - INPT / ENSEEIHT, France
Marcus Völp, University of Luxembourg, Luxembourg

Additional Reviewers

Alejandro Pérez Ruiz
Ali Syed
Anton Cervin
Benjamin Rouxel
Corey Tessler
Dakshina Dasari
Daniel Wiltsche-Prokesch
David García Villaescusa
Dirk Ziegenbein
Eberle Rambo
Ernesto Massa
Federico Terraneo
Florian Heilmann
Frédéric Boniol
Claude-Joachim Hamann
Hannes Weisbach
Houssam Zahaf
Hugo Daigmorte
Ignacio Sanudo Olmedo
J. Javier Gutiérrez
Jacques Combaz
Jean-Dominique Decotignie
Johannes Schlatow
John Cavicchio
Kai Gemlau
Kecheng Yang
Konstantinos Bletsas
Leonie Ahrendts
Maksym Planeta
Marco Solieri
Marcus Hähnel
Micaela Verucchi
Michael González Harbour
Michael Lauer
Michael Pressler
Michael Roitzsch
Miguel Algorri
Ming Yang
Mischa Möstl
Mitra Nasri
Muhammed Ali Awan
Nicola Capodieci
Nils Asmussen
Paolo Burgio
Pascal Sotin
Pieter J. L. Cuijpers
Rany Kahil
Raphael Guerra
Robin Hofmann
Rodrigo Coelho
Ryan Gerdes
Saud Wasly
Sebastian Hahn
Sergey Voronov
Simon Kramer
Stefan Resch
Stephen Tang
Tanya Amert
Valentin Touzeau
Xiaoting Li
Zain Haj Hammadeh
Zhishan Guo

Artifact Evaluators

Gautam Gala, TU Kaiserslautern, Germany
Gautham Nayak Seetanadi, Lund University, Sweden
Roberto Cavicchioli, Università di Modena e Reggio Emilia, Italy
Tobias Klaus, FAU Erlangen-Nürnberg, Germany
Yuanbin Zhou, Hangzhou Dianzi University, China

List of Authors

Jaume Abella
Barcelona Supercomputing Center, Spain
jaume.abella@bsc.es

Kunal Agrawal
Washington University in St. Louis, USA
kunal@wustl.edu

Leonie Ahrendts
TU Braunschweig, Germany
ahrendts@ida.ing.tu-bs.de

Benny Akesson
Embedded Systems Innovation, Eindhoven, The Netherlands
benny.akesson@tno.nl

Waqar Ali
University of Kansas, USA
wali@ku.edu

Tanya Amert
The University of North Carolina at Chapel Hill, USA
tamert@cs.unc.edu

James H. Anderson
The University of North Carolina at Chapel Hill, USA
anderson@cs.unc.edu

Karl-Erik Årzén
Lund University, Sweden
karlerik@control.lth.se

Muhammad Ali Awan
CISTER Research Unit, ISEP-IPP, Porto, Portugal
muaan@isep.ipp.pt

Joshua Bakita
The University of North Carolina at Chapel Hill, USA
jbakita@cs.unc.edu

Sanjoy Baruah
Washington University in St. Louis, USA
baruah@wustl.edu

Iain Bate
University of York, UK
Iain.Bate@york.ac.uk

Pedro Benedicte
Barcelona Supercomputing Center and Universitat Politècnica de Catalunya, Spain
pbenedic@bsc.es

Guillem Bernat
Rapita Systems Ltd., UK
guillem.bernat@rapitasystems.com

Anand Ganpat Bhat
Carnegie Mellon University, USA
anandbha@andrew.cmu.edu

Tjerk Bijlsma
Embedded Systems Innovation, Eindhoven, The Netherlands
tjerk.bijlsma@tno.nl

Enrico Bini
University of Turin, Italy
bini@di.unito.it

Alessandro Biondi
Scuola Superiore Sant'Anna, Italy
alessandro.biondi@sssup.it

Konstantinos Bletsas
CISTER Research Centre and ISEP, Portugal
ksbs@isep.ipp.pt

Thomas Boroske
TU Braunschweig, Germany

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Germany
bbb@mpi-sws.org

Ian Broster
Rapita Systems Ltd., UK
ianb@rapitasystems.com

Georg von der Brüggen
TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de

Alan Burns
University of York, UK
alan.burns@york.ac.uk

Francisco J. Cazorla
Barcelona Supercomputing Center and IIIA-CSIC, Spain
francisco.cazorla@bsc.es

Felipe Cerqueira
Max Planck Institute for Software Systems (MPI-SWS), Germany
felipec@mpi-sws.org

Kuan-Hsun Chen
TU Dortmund University, Germany
kuan-hsun.chen@tu-dortmund.de

Jian-Jia Chen
TU Dortmund University, Germany
jian-jia.chen@cs.uni-dortmund.de

Antoine Colin
Rapita Systems Ltd., UK
antoine.colin@rapitasystems.com

Robert Davis
University of York, UK
rob.davis@york.ac.uk

Marco Di Natale
Scuola Superiore Sant'Anna, Italy
marco@sssup.it

Christian Dietrich
Leibniz Universität Hannover, Germany
dietrich@sra.uni-hannover.de

Tobias Distler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
distler@cs.fau.de

Viktor Edpalm
Axis Communications, Sweden
viktor.edpalm@axis.com

Rolf Ernst
TU Braunschweig, Germany
ernst@ida.ing.tu-bs.de

Heiko Falk
Hamburg University of Technology, Germany
heiko.falk@tuhh.de

Farzad Farshchi
University of Kansas, USA
farshchi@ku.edu

Joachim Fellmuth
Technical University of Berlin, Germany
joachim.fellmuth@tu-berlin.de

Gerhard Fohler
Technische Universität Kaiserslautern, Germany
fohler@eit.uni-kl.de

Johannes Freitag
Airbus, Munich, Germany
johannes.freitag@airbus.com

Sabine Glesner
TU Berlin, Germany
sabine.glesner@tu-berlin.de

Thomas Göthel
Technical University of Berlin, Germany
thomas.goethel@tu-berlin.de

Arpan Gujarati
Max Planck Institute for Software Systems (MPI-SWS), Germany
arpanbg@mpi-sws.org

Carles Hernandez
Barcelona Supercomputing Center, Spain
carles.hernandez@bsc.es

Stuart Hutchesson
Rolls Royce PLC, UK

Kristin Krüger
Technische Universität Kaiserslautern, Germany
krueger@eit.uni-kl.de

Alexander Lint
Oce Technologies, The Netherlands
alexander.lint@oce.com

Martina Maggio
Lund University, Sweden
martina@control.lth.se

Renato Mancuso
Boston University, USA
rmancuso@bu.edu

Alexandre Martins
Axis Communications and Lund University, Sweden
alexandre.martins@axis.com

Katharina Morik
TU Dortmund University, Germany
katharina.morik@tu-dortmund.de

Mitra Nasri
Max Planck Institute for Software Systems (MPI-SWS), Germany
mitra@mpi-sws.org

Geoffrey Nelissen
CISTER/INESC TEC, ISEP, Portugal
grrpn@isep.ipp.pt

Catherine Nemitz
The University of North Carolina at Chapel Hill, USA
nemitz@cs.unc.edu

Dominic Oehlert
Hamburg University of Technology, Germany
dominic.oehlert@tuhh.de

Nathan Otterness
The University of North Carolina at Chapel Hill, USA
otternes@cs.unc.edu

Luigi Pannocchi
Scuola Superiore Sant'Anna, Italy
luigi.pannocchi@sssup.it

Alessandro Vittorio Papadopoulos
Mälardalen University, Sweden
alessandro.papadopoulos@mdh.se

Risat Mahmud Pathan
Chalmers University of Technology, Sweden
risat@chalmers.se

Paolo Pazzaglia
Scuola Superiore Sant'Anna, Italy
paolo.pazzaglia@sssup.it

Nico Piatkowski
TU Dortmund University, Germany
nico.piatkowski@uni-dortmund.de

Sophie Quinton
INRIA-Grenoble Rhône-Alpes, France
sophie.quinton@inria.fr

Ragunathan (Raj) Rajkumar
Carnegie Mellon University, USA
rajkumar@andrew.cmu.edu

Selma Saidi
Hamburg University of Technology, Germany
selma.saidi@tuhh.de

Soheil Samii
General Motors R&D and Linköping University, Sweden, USA
soheil.samii@gm.com

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
wosch@cs.fau.de

F. Donelson Smith
The University of North Carolina at Chapel Hill, USA
smithfd@cs.unc.edu

Pedro F. Souto
University of Porto, Portugal
pfs@fe.up.pt

Eduardo Tovar
CISTER Research Centre and ISEP, Portugal
emt@dei.isep.ipp.pt

Nigel Tracey
ETAS Ltd., UK
nigel.tracey@etas.com

Niklas Ueter
TU Dortmund University, Germany
niklas.ueter@tu-dortmund.de

Sascha Uhrig
Airbus, Munich, Germany
sascha.uhrig@airbus.com

Peter Ulbrich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
ulbrich@cs.fau.de

Theo Ungerer
University of Augsburg, Germany
theo.ungerer@informatik.uni-augsburg.de

Prathap Kumar Valsan
Intel, USA
prathap.kumar.valsan@intel.com

Jacques Verriet
Embedded Systems Innovation, Eindhoven,
The Netherlands
jacques.verriet@tno.nl

Marcus Völp
SnT - Université du Luxembourg,
Luxembourg
marcus.voelp@uni.lu

Peter Wägemann
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany
waegemann@cs.fau.de

Ming Yang
The University of North Carolina at Chapel
Hill, USA
yang@cs.unc.edu

Heechul Yun
University of Kansas, USA
heechul.yun@ku.edu

Deterministic Memory Abstraction and Supporting Multicore System Architecture

Farzad Farshchi
University of Kansas, USA
farshchi@ku.edu

Prathap Kumar Valsan
Intel, USA
prathap.kumar.valsan@intel.com

Renato Mancuso
Boston University, USA
rmancuso@bu.edu

Heechul Yun
University of Kansas, USA
heechul.yun@ku.edu

Abstract

Poor time predictability of multicore processors has been a long-standing challenge in the real-time systems community. In this paper, we make a case that a fundamental problem that prevents efficient and predictable real-time computing on multicore is the *lack of a proper memory abstraction* to express memory criticality, which cuts across various layers of the system: the application, OS, and hardware. We, therefore, propose a new holistic resource management approach driven by a new memory abstraction, which we call *Deterministic Memory*. The key characteristic of deterministic memory is that the platform–the OS and hardware–guarantees small and tightly bounded worst-case memory access timing. In contrast, we call the conventional memory abstraction as best-effort memory in which only highly pessimistic worst-case bounds can be achieved. We propose to utilize both abstractions to achieve high time predictability but without significantly sacrificing performance. We present deterministic memory-aware OS and architecture designs, including OS-level page allocator, hardware-level cache, and DRAM controller designs. We implement the proposed OS and architecture extensions on Linux and gem5 simulator. Our evaluation results, using a set of synthetic and real-world benchmarks, demonstrate the feasibility and effectiveness of our approach.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases multicore processors, real-time, shared cache, DRAM controller, Linux

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.1

Acknowledgements This research is supported by NSF CNS 1718880. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

1 Introduction

High-performance embedded multicore platforms are increasingly demanded in cyber-physical systems (CPS)–especially those in automotive and aviation applications–to cut cost and to reduce size, weight, and power (SWaP) of the system via consolidation [31].

Consolidating multiple tasks with different criticality levels (a.k.a. mixed-criticality systems [58, 8]) on a single multicore processor is, however, extremely challenging because interference in shared hardware resources in the memory hierarchy can significantly alter the tasks' timing characteristics. Poor time predictability of multicore platforms is a major hurdle that makes their adoption challenging in many safety-critical CPS. For example, the CAST-32A position paper by the avionics certification authorities comprehensively discusses the certification challenges of multicore avionics [9]. Therefore, in the aerospace industry, it is a common practice to disable all but one core [28], because extremely pessimistic worst-case-execution times (WCETs) nullify any performance benefits of using multicore processors in critical applications. This phenomenon is also known as the "one-out-of-m" problem [27].

There have been significant research efforts to address the problem. Two common strategies are (1) partitioning the shared resources among the tasks or cores to achieve spatial isolation and (2) applying analyzable arbitration schemes (e.g., time-division multiple access) in accessing the shared resources to achieve temporal isolation. These strategies have been studied individually (e.g., cache [25, 59, 39], DRAM banks [38, 63], memory bus [64, 40]) or in combination (e.g., [27, 51]). However, most of these efforts improve predictability at the cost of a significant *sacrifice in efficiency and performance.*

In this paper, we argue that the fundamental problem that prevents efficient and predictable real-time computing on multicore is the *lack of a proper memory abstraction* to express memory criticality, which cuts across various layers of the system: the application, OS, and hardware. Thus, our approach starts by defining a new OS-level memory abstraction, which we call *Deterministic Memory.* The key characteristic of deterministic memory is that the platform–the OS and hardware–guarantees small and tightly bounded worst-case memory access timing. In contrast, we call the conventional memory abstraction as best-effort memory in which only highly pessimistic worst-case bounds can be achieved.

We propose a new holistic cross-layer resource management approach that leverages the deterministic and best-effort memory abstractions. In our approach, a task can allocate either type of memory blocks in its address space, at the page granularity, based on the desired WCET requirement in accessing the memory blocks. The OS and hardware then apply different resource management strategies depending on the memory type. Specifically, predictability focused strategies, such as resource reservation and predictable scheduling, shall be used for deterministic memory while average performance and efficiency-focused strategies, such as space sharing and out-of-order scheduling, shall be used for best-effort memory. Because neither all tasks are time-critical nor all memory blocks of a time-critical task are equally important with respect to the task's WCET, our approach enables the possibility of achieving high time predictability without significantly affecting performance and efficiency through the selective use of deterministic memory.

While our approach is a generic framework that can be applied to any shared hardware resource management, in this paper, we particularly focus on the shared cache and main memory, and demonstrate the potential benefits of our approach in the context of shared cache and DRAM related resource management. First, we describe OS extensions and an OS-level memory allocation method to support deterministic memory. We then describe a deterministic memory-aware cache design that provides the same level of cache space isolation of the conventional way-based partitioning techniques, while achieving significantly higher cache space utilization. We also describe a deterministic memory-aware DRAM controller design that extends a previously proposed real-time memory controller [55] to achieve similar predictability benefits with minimal DRAM space waste.

We implement the deterministic memory abstraction and an OS-level memory allocator (replacing Linux's buddy allocator) in a Linux 3.13 kernel and implement the proposed deterministic-memory aware memory hierarchy hardware extensions (in MMU, TLB, cache and DRAM controller) in a gem5 full-system simulator [7] modeling a high-performance (out-of-order) quad-core platform as the baseline. We evaluate the system using a set of synthetic and real-world benchmarks from EEMBC [14], SD-VBS [57] and SPEC2006 [19] suites. We achieve the same degree of isolation with conventional way-based cache partitioning for real-time tasks while improving the cache hit rate of co-scheduled non-real-time workloads by 39% on average. In addition, we need significantly less memory space in reserved DRAM banks, while achieving comparable WCET guarantees compared with a state-of-the-art real-time DRAM controller.

The main contributions of this work are as follows:

- We propose a new OS-level memory abstraction, which we call *Deterministic Memory*, that enables efficient cross-layer resource management, balancing time predictability and resource efficiency.
- We present a concrete system design–from the OS down to the entire memory hierarchy, including shared cache and DRAM controller designs–that demonstrate the potential benefits of the new memory abstraction. The key contribution of our design is its Memory Management Unit (MMU) based approach that provides flexible, fine-grained (page-granularity) resource management across the entire memory hierarchy.
- We implement a realistic prototype system on a Linux kernel and a cycle-accurate full system simulator. [1] We also provide extensive empirical results, using both synthetic and real-world benchmarks, that demonstrates the effectiveness of our approach.

The remainder of the paper is organized as follows. Section 2 provides background and motivation. Section 3 describes the proposed Deterministic Memory abstraction. Section 4 provides an overview of the deterministic memory-aware system design. Section 5 presents DM-aware timing analysis. Section 6 details our prototype implementation. Section 7 presents evaluation results. We review related work in Section 8 and conclude in Section 9.

2 Background and Motivation

In this section, we describe why the standard uniform memory abstraction is a fundamental limitation for the development of efficient and predictable real-time computing infrastructures.

CPU-centric Abstractions and Resource Management. Traditionally, the CPU has been the main focus of resource management in real-time systems. This is because, in a unicore processor, only one task at a time can access the entire memory hierarchy and that CPU scheduling decisions have a predominant impact on the response time of real-time tasks. Therefore, CPU-centric abstractions such as core, task and task priority have been the primary focus of resource management. However, in multicore platforms, which have become

[1] We provide the modified Linux kernel source, the modified gem5 simulator source, and the simulation methodology at `http://github.com/CSL-KU/detmem` for replication study.

mainstream over the last decade, extensive inter-core hardware resource sharing in the memory hierarchy heavily impacts task timing. Hence, CPU time management is no longer the sole dimension to explore when reasoning about the temporal behavior of a system. Various OS and hardware-level solutions have been proposed to manage shared resources in the memory hierarchy with the goal of improving time predictability (we will provide a comprehensive review of related work in Section 8). Nonetheless, in most approaches, CPU-centric abstractions are still most widely used to perform allocation and scheduling of shared resources in the memory hierarchy. Unfortunately, CPU-centric abstractions are often too coarse-grained to enact accurate management policies on memory hierarchy resources, such as cache lines and main memory pages. For instance, when a fraction of cache space is reserved for a task, it cannot be used by other tasks, even if it is not fully utilized by the reserved task. Likewise, when DRAM banks are reserved for a task, they cannot be utilized by other tasks, resulting in under-utilized DRAM space, even though not all memory of the task may need to be allocated on the reserved DRAM bank.

The Uniform Memory Abstraction. Operating systems and hardware traditionally have provided a simple uniform memory abstraction that hides all the complex details of the memory hierarchy. When an application requests to allocate more memory, the OS simply maps the necessary amount of *any* physical memory pages available at the time to the application's address space–without considering: (1) how the memory pages are actually mapped to the shared hardware resources in the memory hierarchy, and (2) how they will affect application performance. Likewise, the underlying hardware components treat all memory accesses from the CPU as equal without any regard to differences in criticality and timing requirements in allocating and scheduling the requests.

We argue that this uniform memory abstraction is fundamentally inadequate for multicore systems because it prevents the OS and the memory hierarchy hardware from making informed decisions in allocating and scheduling access to shared hardware resources. As such, we believe that new memory abstractions are needed to enable *both* efficient and predictable real-time resource management. It is important to note that the said abstractions should not expose too many architectural details about the memory hierarchy to the users, to ensure portability in spite of rapid changes in hardware architectures.

3 Deterministic Memory Abstraction

In this section, we introduce the *Deterministic Memory* abstraction to address the aforementioned challenges.

We define deterministic memory as special memory space for which the OS and hardware guarantee small and tightly bounded worst-case access delay. In contrast, we call conventional memory as best-effort memory, for which only highly pessimistic worst-case bounds can be achieved. A platform shall support both memory types, which allow applications to express their memory access timing requirements in an architecture-neutral way, while leaving the implementation details to the platform–the OS and the hardware architecture. This, in turn, enables efficient and analyzable cross-layer resource management, as we will discuss in the rest of the section.

Figure 1 shows the conceptual differences between the two memory types with respect to worst-case memory access delay bounds. For clarity, we divide memory access delay into two components: *inherent access delay* and *inter-core interference delay*. The inherent access delay is the minimum necessary timing in isolation. In this regard, deterministic memory can

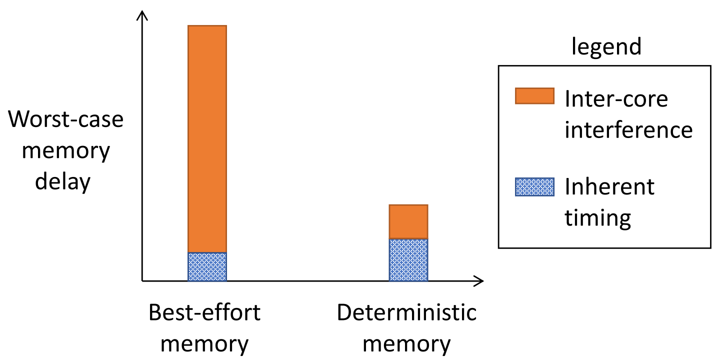


Figure 1 Conceptual differences of deterministic and best-effort memories.

Table 1 Differences in resource management strategies.

	Space allocation	Request scheduling	WCET bounds
Deterministic memory	Dedicated resources	Predictability focused	Tight
Best-effort memory	Shared resources	Performance focused	Pessimistic

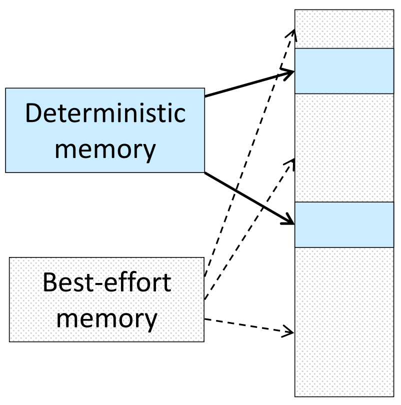
be slower–in principal, but not necessarily–than best-effort memory, as its main objective is predictability and not performance, while in the case of best-effort memory, the reverse is true. The inter-core interference delay is, on the other hand, an additional delay caused by concurrently sharing hardware resources between multiple cores. This is where the two memory types differ the most. For best-effort memory, the worst-case delay bound is highly *pessimistic* mainly because the inter-core interference delay can be severe. For deterministic memory, on the other hand, the worst-case delay bound is *small and tight* as the inter-core interference delay is minimized by the platform.

Table 1 shows general spatial and temporal resource management strategies of the OS and hardware to achieve the differing goals of the two memory types. Here, we mainly focus on shared hardware resources, such as shared cache, DRAM banks, memory controllers, and buses. In contrast, we do not focus on core-private hardware resources such as private (L1) caches as they do not generally contribute to inter-core interference.

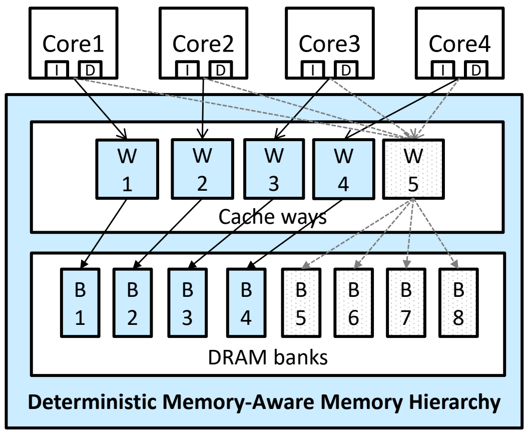
In the deterministic memory approach, a task can map all or part of its memory from the deterministic memory. For example, an entire address space of a real-time task can be allocated from the deterministic memory; or, only the important buffers used in a control loop of the real-time task can be allocated from the deterministic memory, while temporary buffers used in the initialization phase are allocated from the best-effort memory.

Our key insight is that *not all memory blocks of an application are equally important with respect to the application's WCET*. For instance, in the applications we profiled in Section 7.1, only a small fraction of memory pages account for most memory accesses to the shared memory hierarchy (shared cache and DRAM).

Based on this insight, we now provide a detailed design and implementation of deterministic memory-aware OS and architecture extensions with a goal of achieving high efficiency and predictability.



(a) Application view (logical).



(b) System-level view (physical).

Figure 2 Logical and physical mappings of the *deterministic* and *best-effort* memory abstractions.

4 System Design

In this section, we first provide a high-level overview of a deterministic-memory based multicore system design (Section 4.1). We then describe necessary small OS and hardware architecture extensions to support the deterministic/best-effort memory abstractions (Section 4.2). Lastly, we describe deterministic memory-aware cache and DRAM management frameworks (Section 4.3 and 4.4, respectively).

4.1 Overview

Figure 2a shows the virtual address space of a task using both deterministic and best-effort memory under our approach. From the point of view of the task, the two memory types differ only in their worst-case timing characteristics. The deterministic memory is realized by extending the *virtual memory system* at the page granularity. Whether a certain page is deterministic or best-effort memory is stored in the task's page table and the information is propagated throughout the shared memory hierarchy, which is then used in allocation and scheduling decisions made by the OS and the memory hierarchy hardware.

Figure 2b shows the system-level (OS and architecture) view of a multicore system supporting the deterministic and best-effort memory abstractions. In this example, each core is given one cache way and a DRAM bank which will be used to serve deterministic memory for the core. One cache way and four DRAM banks are assigned to the best-effort memory of all cores. Here, the highlighted deterministic memory-aware memory hierarchy refers to hardware support for the deterministic memory abstraction.

It is important to note that the support for deterministic memory is generally more *expensive* than that of best-effort memory in the sense that it may require dedicated space, which may be wasted if under-utilized, and predictability focused scheduling, which may not offer the highest performance. As such, to improve efficiency and performance, it is desirable to use as little deterministic memory as possible as long as the desired worst-case timing of real-time tasks can be satisfied.

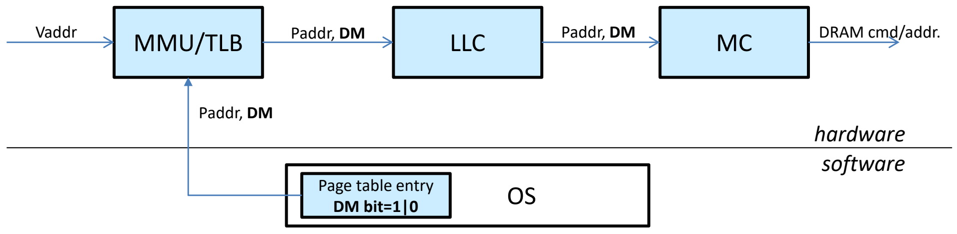


Figure 3 Deterministic memory-aware memory hierarchy: Overview.

4.2 OS and Architecture Extensions for Deterministic Memory Abstraction Support

The deterministic memory abstraction is realized by extending the OS's virtual memory subsystem. Whether a certain page has the deterministic memory property or not is stored in the corresponding page table entry. Note that in most architectures, a page table entry contains not only the virtual-to-physical address translation but also a number of auxiliary attributes such as access permission and cacheability. The deterministic memory can be encoded as just another attribute, which we call a DM bit, in the page table entry. [2] The OS is responsible for updating the DM bits in each task's page tables. The OS provides interfaces for applications to declare and update their deterministic/best-effort memory regions at the page granularity. Any number of memory regions of any sizes (down to a single page) within the application's address space can be declared as deterministic memory (the rest is best-effort memory by default).

In a modern processor, the processor's view of memory is determined by the Memory Management Unit (MMU), which translates a virtual address to a corresponding physical address. The translation information, along with other auxiliary information, is stored in a page table entry, which is managed by the OS. Translation Look-aside Buffer (TLB) then caches frequently accessed page table entries to accelerate the translation speed. As discussed above, in our design, the DM bit in each page table entry indicates whether the page is for deterministic memory or for best-effort memory. Thus, the TLB also stores the DM bit and passes the information down to the memory hierarchy.

Figure 3 shows this information flow of deterministic memory. Note that bus protocols (e.g., AMBA [2]) also should provide a mean to pass the deterministic memory information into each request packet. In fact, many existing bus protocols already support some forms of priority information as part of bus transaction messages [3]. These fields are currently used to distinguish priority between bus masters (e.g., CPU vs. GPU vs. DMA controllers). A bus transaction for deterministic memory can be incorporated into these bus protocols, for example, as a special priority class. The deterministic memory information can then be utilized in mapping and scheduling decisions made by the respective hardware components in the memory hierarchy.

In the following, we focus on cache and DRAM controllers and how the deterministic memory information can be utilized in these important shared hardware resources.

[2] In our implementation, we currently use an unused memory attribute in the page table entry of the ARM architecture; see Section 6 for details.

[3] For example, ARM AXI4 protocol includes a 4-bit QoS identifier AxQOS signal [2] that supports up to 16 different priority classes for bus transactions.

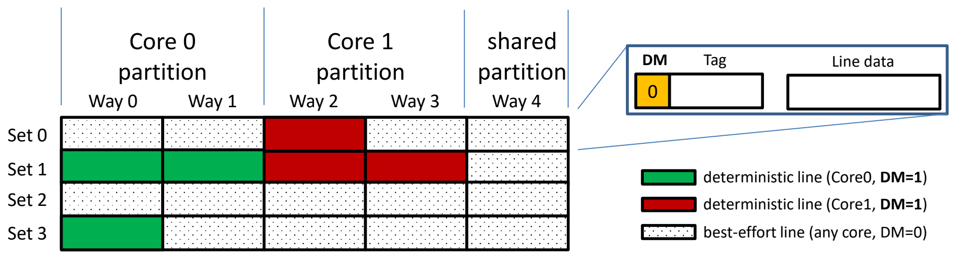


Figure 4 Deterministic memory-aware cache management.

4.3 Deterministic Memory-Aware Shared Cache

In this subsection, we present a deterministic memory-aware shared cache design that provides the same isolation benefits of traditional way-based cache partitioning techniques while achieving higher cache space utilization.

Way-based Cache Partitioning. In a standard way-based partitioning, which is supported in several COTS multicore processors [15, 3], each core is given a subset of cache ways. When a cache miss occurs, a new cache line (loaded from the memory) is allocated on one of the assigned cache ways in order not to evict useful cache lines of the other cores that share the same cache set. An important shortcoming of way-partitioning is, however, that its partitioning granularity is coarse (i.e., way granularity) and the cache space of each partition may be wasted if it is underutilized. Furthermore, even if fine-grain partition adjustment is possible, it is not easy to determine the "optimal" partition size of a task because the task's behavior may change over time or depending on the input. As a result, it is often a common practice to conservatively allocate sufficient amount of resource (over-provisioning), which will waste much of the reserved space most of the time.

Deterministic Memory-Aware Replacement Algorithm. We improve way-based partitioning by taking advantage of the deterministic memory abstraction. The basic approach is that we use way partitioning only for deterministic memory accesses while allowing best-effort memory accesses to use all the cache ways that do not currently hold deterministic cache lines.

Figure 4 shows an example cache status of our design in which two cores share a 4-set, 5-way set-associative cache. In our design, each cache-line includes a DM bit to indicate whether the cache line is for deterministic memory or best-effort memory (see the upper-right side of Figure 4). When inserting a new cache line (of a given set), if the requesting memory access is for deterministic memory, then the victim line is chosen from the core's way partition (e.g., way 0 and 1 for Core 0 in Figure 4). On the other hand, if the requesting memory access is for best-effort memory, the victim line is chosen from the ways that do not hold deterministic cache lines. (e.g., in set 0 of Fugre 4, all but way 2 are best-effort cache lines; in set 1, only the way 4 is best-effort cache line.)

Algorithm 1 shows the pseudo code of the cache line replacement algorithm. As in the standard cache way-partitioning, we assign dedicated cache ways for each core, denoted as $PartMask_i$, to eliminate inter-core cache interference. Note that $DetMask_s$ denotes the bitmask of the set s's cache lines that contain deterministic memory. If a request from core i is a deterministic memory request ($DM = 1$), then a line is allocated from the core's cache way partition ($PartMask_i$). Among the ways of the partition, the algorithm first tries

Algorithm 1: Deterministic memory-aware cache line replacement algorithm.

```
Input  : $PartMask_i$ - way partition mask of Core $i$
         $DetMask_s$ - deterministic ways of Set $s$
Output : $victim$ - the victim way to be replaced.
1  if $DM == 1$ then
2      if $(PartMask_i \wedge \neg DetMask_s) \neq NULL$ then
           // evict a best-effort line first
3          $victim = LRU(PartMask_i \wedge \neg DetMask_s)$
4          $DetMask_s \mid= 1 \ll victim$
5      else
           // evict a deterministic line
6          $victim = LRU(PartMask_i)$
7      end
8  else
       // evict a best-effort line
9      $victim = LRU(\neg DetMask)$
10 end
11 return $victim$
```

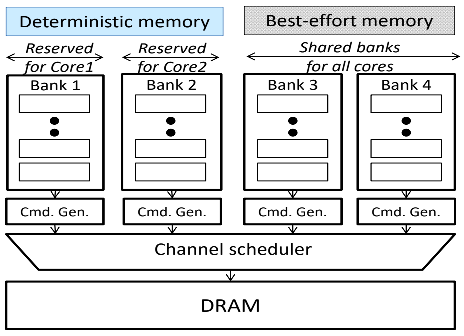
to evict a best-effort cache line, if such a line exists (Line 3-4). If not (i.e., all lines are deterministic ones), it chooses one of the deterministic lines as the victim (Line 6). One the other hand, if a best-effort memory is requested ($DM \neq 1$), it evicts one of the best-effort cache lines, but not any of the deterministic cache lines (Line 9). In this way, while the deterministic cache lines of a partition are completely isolated from any accesses other than the assigned core of the partition, any under-utilized cache lines of the partition can still be utilized as best-effort cache lines by all cores.

Deterministic Memory Cleanup. Note that a core's way partition would eventually be filled with deterministic cache lines (ones with $DM = 1$) if left unmanaged (e.g., scheduling multiple different real-time tasks on the core). This would eliminate the space efficiency gains of using deterministic memory because the deterministic memory cache lines cannot be evicted by best-effort memory requests.

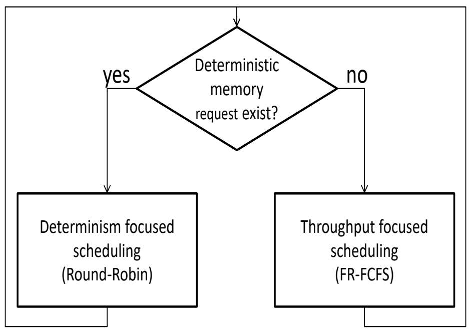
In order to keep only a minimal number of deterministic cache lines on any given partition in a predictable manner, our cache controller provides a special hardware mechanism that clears the DM bits of all deterministic cache lines, effectively turning them into best-effort cache-lines. This mechanism is used by the core's OS scheduler on each context switch so that the deterministic cache-lines of the previous tasks can be evicted by the current task. When the deterministic-turned-best-effort cache-lines of a task are accessed again and they still exist in the cache, they will be simply re-marked as deterministic without needing to reload from memory. In the worst case, however, all deterministic cache lines of a task shall be reloaded when the task is re-scheduled on the CPU.

Note that our cache controller reports the number of deterministic cache lines that are cleared on a context switch back to the OS. This information can be used to more accurately estimate cache-related preemption delays (CRPD) [1].

Guarantees. The premise of the proposed cache replacement strategy is that a core's deterministic cache lines will never be evicted by other cores' cache allocations, hence



(a) MC architecture.


Deterministic memory request exist?
yes
no
Determinism focused scheduling (Round-Robin)
Throughput focused scheduling (FR-FCFS)

(b) Scheduling algorithm.

Figure 5 Deterministic memory-aware memory controller architecture and scheduling algorithm.

preserving the benefit of cache partitioning. At the same time, non-deterministic cache lines in the core's cache partition can safely be used as other cores' best-effort memory requests, hence *minimizing wasted cache capacity* due to partitioning.

Comparison with PRETI. Our DM-aware cache replacement algorithm is similar to several prior mixed-criticality aware cache designs [32, 30, 62]. The most closely related work is PRETI [32], which also modifies LRU to be mixed-criticality-aware. There are, however, several notable differences. First, PRETI uses the thread(task)-id to distinguish critical and non-critical cache accesses, whereas we use MMU, which enables finer, page granularity criticality control. Second, in PRETI, cache-lines reserved for a critical thread can only be released on its termination. In contrast, we provide a DM-cleanup mechanism, which enables efficient reclamation of deterministic memory cache-lines at each context-switch. Third, PRETI's replacement algorithm provides a firm cache space reservation capability [44] in the sense that a real-time task can utilize more than its dedicated private space, whereas our replacement algorithm does not allow such additional cache space utilization. Lastly, while the prior works mainly focus on the cache, our main goal is to provide a unified framework–the deterministic memory abstraction–which can carry information about time-sensitivity of memory space not only in the cache, but also in the OS and throughout the entire memory hierarchy. We will discuss how a traditional DRAM controller can be extended to support deterministic memory abstraction in the following.

4.4 Deterministic Memory-Aware DRAM Controller

In this subsection, we present a deterministic memory-aware DRAM controller design, which, in collaboration with our OS support, provides strong spatial and temporal isolation for deterministic memory accesses while also enables efficient best-effort memory processing.

First, the OS actively controls on which DRAM bank a page frame is allocated. Specifically, the OS reserves a small number of banks for each core to be used as the deterministic memory for the core, while the rest of the banks are used for the best-effort memory of all cores, as shown in Figure 5a (also in Figure 2b). When the OS allocates memory pages of an application task, deterministic memory pages of the task shall be allocated on core-private DRAM banks to eliminate DRAM bank-level inter-core interference [24, 55], while best-effort memory pages are allocated on the shared DRAM banks.

Second, the memory controller (MC) implements a *two-level scheduling* algorithm that first prioritizes deterministic memory requests over the ones for best-effort memory. For deterministic memory requests, we use a round-robin scheduling policy as it offers stronger time predictability [43], while we use first-ready first-come-first-serve (FR-FCFS) policy for scheduling best-effort memory requests as it offers high average throughput [46]. Figure 5b shows the flowchart of the scheduler. Note, however, that strictly prioritizing deterministic memory requests could starve best-effort memory requests indefinitely. Since we assume the existence of a pessimistic worst-case bound for best-effort memory, we limit the maximum number of consecutive processing of deterministic memory requests in case best-effort memory requests exist, in order to achieve tightly bounded worst-case timing for deterministic memory while achieving pessimistic, but still bounded, worst-case timing for best-effort memory.

Our design is inspired by prior mixed-criticality memory controller proposals [26, 22, 55], all of which, like us, apply different scheduling algorithms depending on memory criticality, although detailed designs (and assumptions) are varied. In this work, we particularly use the MEDUSA memory controller design [55] as our baseline, but improve its efficiency by leveraging the DM-bit information passed down to the memory controller. Specifically, in [55], a real-time task has to allocate its entire memory space from the reserved DRAM banks, even when much of its allocated memory is never used in the time-critical part. In contrast, our design can *reduce* the amount of memory allocated in the reserved DRAM banks by only allocating the deterministic memory pages. This allows us to accommodate more real-time tasks with the same amount of reserved DRAM banks.

The necessary changes to support deterministic memory is small. Specifically, the original MEDUSA controller [55] uses a set of memory controller specific hardware registers to identify reserved DRAM banks of the cores. Instead, our modified memory controller design simply uses the DM-bit information in each memory request to determine memory criticality. Other mixed-criticality real-time memory controllers designs [26, 22] also similarly rely on memory-controller-specific hardware registers to identify memory criticality. Thus, we believe they also can be easily augmented to support the deterministic memory abstraction.

4.5 Other Shared Hardware Resources

We briefly discuss other potential deterministic memory-aware shared hardware designs.

As shown in [56], the miss-status-holding-registers (MSRHs) in a shared non-blocking cache can be a significant source of inter-core interference if the number of MSHRs in the shared cache is insufficient to support the memory parallelism of the cores. The contention in MSHRs can be avoided by simply having a sufficient number of MSHRs, as we did in our evaluation setup. But if it is difficult for the reasons discussed in [56], deterministic memory-aware MSHR management can be alternatively considered. For example, one possible DM-aware approach is that reserving some per-core MSHR entries to handle deterministic memory and sharing the rest of MSHR entries for best-effort memory requests from all cores.

Deterministic memory-aware TLB can also be considered. Although a TLB is not typically shared among the cores, it is conceivable to design a DM-aware TLB replacement policy that reserves some TLB entries for deterministic memory that cannot be evicted by access to best-effort memory addresses. Such a policy can be useful to reduce task WCET and CRPD overhead within a core.

5 Timing Analysis

In this section, we show how the traditional response time analysis (RTA) [5] can be extended to account for deterministic and best-effort memory abstractions.

In our system, a real-time task τ_i is represented by the following parameters:

$$\tau_i = \{C_i, T_i, D_i, DM_i, BM_i\} \tag{1}$$

where C_i is the WCET of the τ_i when it executes in isolation; T_i is the period of the task; D_i is the deadline of the task; DM_i represents the maximum number of deterministic memory requests that suffer inter-core interference; BM_i is the maximum number of best-effort memory requests that are subject to inter-core interference.

Note that all these parameters can be obtained in *isolation*. A task is said to execute in isolation if: (1) it executes alone on the assigned core under a given resource partition; and (2) all the other $N_{proc} - 1$ cores are idle or offline.

Note also that in the proposed DM-aware system described earlier, DM_i accounts only a subset of deterministic memory accesses that result in L2 misses because the L2 hit accesses would not suffer inter-core interference. On the other hand, BM_i would represent a subset of best-effort memory accesses that result in L1 misses (not L2 misses). This is because for best-effort memory, L2 cache space is shared, and, in the worst-case, all of them will have to be fetched from the memory controller.

We then can compute τ_i's worst-case memory interference delay I_i as follows:

$$I_i = DM_i \times RD^{dm} + BM_i \times RD^{bm}, \tag{2}$$

where RD^{dm} and RD^{bm} denote the *worst-case inter-core interference delay* of a deterministic and best-effort memory request, respectively.

By our system design, RD^{dm} is small and tightly bounded because accesses to deterministic memory suffer minimal (or zero) inter-core interference at the shared L2 cache and the shared DRAM. On the other hand, RD^{bm} will be substantially higher and highly pessimistic because we have to pessimistically assume access to best-effort memory will always miss the L2 cache and suffer high queuing delay at the DRAM controller.

Traditional RTA analysis can then be performed by finding the first value of k such that $R_i^{(k+1)} = R_i^{(k)}$ (task is schedulable) or such that $R_i^{(k)} > D_i$ (task is not schedulable), given that $R_i^{(0)} = C_i + I_i$ and that $R_i^{(k+1)}$ is calculated as:

$$R_i^{(k+1)} = C_i + I_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil \cdot (C_j + I_j), \tag{3}$$

where $hp(i)$ is the set of all the tasks with priority higher than τ_i.

The major benefit of our approach is its flexibility. For example, a pure COTS multicore system may provide high performance but, doesn't provide isolation guarantees. Therefore, all access to shared resource may need to be assumed to suffer highly pessimistic worst-case inter-core interference delay (e.g., RD^{bm} above) because no isolation is guaranteed. On the other hand, a fully time-predictable hardware architecture [37, 54, 53] may provide strong timing predictability with a small tight worst-case inter-core interference delay (e.g., RD^{dm} above), but not high performance and efficiency. In contrast, the flexibility of our approach enables hardware designs that optimize differently depending on the memory type, which in turn enables analyzable and efficient multicore systems.

6 Prototype Implementation

In this section, we provide implementation details of our prototype, which is based on Linux 3.13 kernel and gem5 [7] full-system simulator. First, we briefly review the ARMv7 architecture on which our implementation is based (Section 6.1). We then describe our modifications

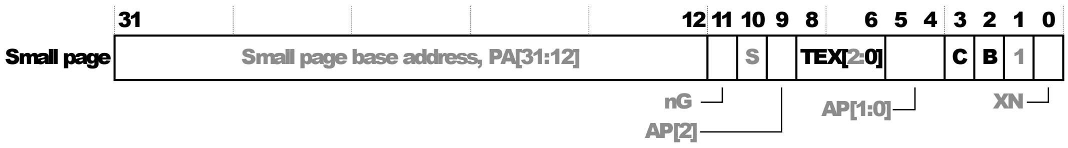


Figure 6 Small descriptor format for 2^{nd} level page table entry in ARMv7-A family SoCs [3].

to the Linux kernel to support the deterministic memory abstraction (Section 6.2). Lastly, we describe the hardware extensions on the gem5 simulator (Section 6.3).

6.1 ARM Architecture Background

We use the ARMv7-A [3] architecture because it is well supported by the gem5 simulator. The ARMv7 architecture defines four primary memory types and several memory-related attributes such as cache policy (write-back/write-through) and coherence boundaries (between cores or beyond). Up to 8 different combinations are allowed by the architecture. Each page's memory type is determined by a set of bits in the corresponding 2^{nd}-level page table entry. Figure 6 illustrates the structure of a page table entry (PTE).

In the figure, the bits `TEX[0]`, `C` and `B` are used to define one of the 8 memory types. The property of each memory type is determined by two global architectural registers, namely Primary Region Remap Register (PRRR) and Normal Memory Region Register (NMRR) [4].

6.2 Linux Extensions

We have modified Linux kernel 3.13 to support deterministic memory.

At the lowest level, we define a new memory type that corresponds to the deterministic memory. The default ARM Linux uses only 6 out of 8 possible memory types of ARMv7, leaving two undefined memory types. For deterministic memory, we define one of the unused memory types as the deterministic memory type, by updating PRRR and NMRR registers at boot time. A page is marked as deterministic memory when the corresponding page table entry's memory attributes point to the deterministic memory type.

At the user-level, we extend Linux's ELF (Executable and Linkable Format [10]) loader and the `exec` system call implementation. We currently use a special file extension to inform the ELF loader whether to mark the entire memory address or a subset of task's memory pages as deterministic memory. For fine-grained control, the virtual page numbers which might be marked as deterministic memory are currently hard-coded in the kernel source and a subset of them is selected based on the arguments passed to the `exec` system call. In the future, we will use Linux kernel's `debugfs` interface to efficiently communicate page information. Also, the ELF header of a program binary can be used instead to encode the virtual page numbers.

Within the Linux kernel, a task's virtual address space is represented as a set of *memory regions*, each of which is represented by a data structure, `vm_area_struct`, called a VMA descriptor. Each VMA descriptor contains a variety of metadata about the memory region, including its memory type information. Whenever a new physical memory block is allocated (at a page fault), the kernel uses the information stored in the corresponding VMA descriptor

[4] The hardware behaves as described only when the so -called “TEX remapping” mechanism is in use. TEX remapping can be controlled via a configuration bit (TRE) in the System Control Register (SCTLR). The Linux kernel enabled TEX remapping by default.

to construct the page table entry for the new page. We add a new flag `VM_DETMEM` to indicate the deterministic memory type in a VMA descriptor. When a page fault happens on accessing a memory address, if the `VM_DETMEM` flag of the memory region corresponding to the address is set, or the address falls within one of the virtual page numbers hard-coded in the kernel then OS sets the `TEX[0]`, `C` and `B` bits in allocating the page for the address to mark that it is a deterministic memory page.

Note that the above code changes are minimal. In total, we only have added/modified less than 200 lines of C and assembly code over 12 files in the Linux kernel source tree. Furthermore, because most changes are in page table descriptors and their initialization, no runtime overhead is incurred by the code changes.

We then have applied the PALLOC patch [63], which replaces the buddy allocator to support DRAM bank-aware page allocation. We further extend the PALLOC allocator to support deterministic memory. Specifically, we extend PALLOC's `cgroup` interface to declare a subset of banks to be used as private banks for the cgroup's deterministic memory pages and another subset of banks to be used for best-effort memory pages.

6.3 Gem5 Extensions

We have modified the gem5 full-system simulator as follows.

MMU and TLB. The deterministic memory type information stored in the page table is read by the MMU and passed throughout the memory hierarchy. When a page fault occurs, the MMU performs the page table walk to determine the physical address of the faulted virtual address. In the process, it also reads other important auxiliary information such as memory attribute and access permission from the page table entry and stores them into a TLB entry in the processor. The deterministic memory attribute is stored alongside with the other memory attributes in the TLB entry. Specifically, we add a single bit in the gem5's implementation of a TLB entry to indicate the deterministic memory type. As a reference, Cortex-A17's TLB entry has 80 bits and a significant fraction of the bits are already used to store various auxiliary information [4] or reserved for future use. Thus, requiring a single bit in a TLB entry does not pose significant overhead in practice. We also extend the memory request packet format in the gem5 simulator to include the deterministic memory type information. In this way, the memory type information of each memory request can be passed down through the memory hierarchy. In real hardware, bus protocols should be extended to include such information. As discussed earlier, existing bus protocols such as AXI4 already support the inclusion of such additional information in each bus packet [2].

Cache Controller. The gem5's cache subsystem implements a flexibly configurable non-blocking cache architecture and supports standard LRU and random replacement algorithms. Our modifications are as follows. First, we extend gem5's cache controller to support a standard way-based partitioning capability [5]. The way partition is configured via a set of programmable registers. When a cache miss occurs, instead of replacing the cache line in the LRU position, the controller replaces the LRU line among the configured ways for the core. The way-based partitioning mechanism is used as a baseline. On top of the way-based partitioning, we implement the proposed deterministic memory-aware replacement and cleanup algorithms (Section 4.3).

[5] https://github.com/farzadfch/gem5-cache-partitioning

Table 2 Simulator configuration.

Core	Quad-core, out-of-order, 2 GHz, IQ: 96, ROB: 128, LSQ: 48/48
L1-I/D caches	Private 16/16 KiB (2-way), MSHRs: 2(I)/6(D)
L2 cache	Shared 2 MiB (16-way), LRU, MSHRs: 56, hit latency: 12
DRAM Controller	Read buffer: 64, write buffer: 64, open-adaptive page policy
DRAM module	LPDDR2@533MHz, 1 rank, 8 banks

DRAM Controller. Gem5's memory controller subsystem supports a standard FR-FCFS algorithm [18]. We have extended the memory controller subsystem to support the two-level scheduling algorithm described in [55]. The two-level scheduler is modified to leverage the DM bit passed to the memory controller as part of each memory request bus transaction. Also, to prevent starvation of best-effort memory requests, we limit the maximum consecutive deterministic memory request processing to 30 when one or more best-effort memory requests are in the memory controller's queue.

7 Evaluation

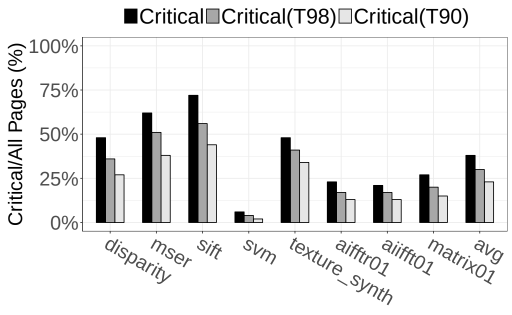
In this section, we present evaluation results to support the feasibility and effectiveness of the proposed deterministic memory-aware system design.

System Setup. For OS, we use a modified Linux kernel 3.13, which implements the modifications explained in Section 6.2 to support the deterministic memory abstraction. For hardware, we use a modified gem5 full system simulator, which implements the proposed deterministic memory support described in Section 6.3. The simulator is configured as a quad-core out-of-order processor (O3CPU model [16]) with per-core private L1 I/D caches, a shared L2 cache, and a shared DRAM. The baseline architecture parameters are shown in Table 2. We use the `mlockall` system call to allocate all necessary pages of each real-time application at the beginning so as to avoid page faults during the rest of program's execution. In addition, we enabled the kernel configuration option `NO_HZ_FULL` to reduce unnecessary scheduler-tick interrupts.

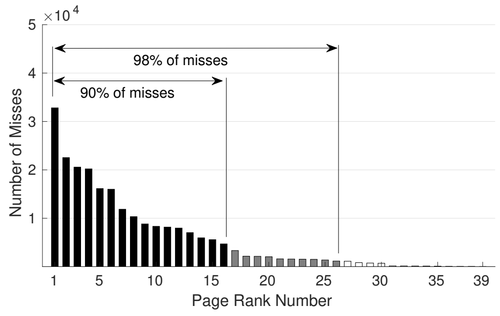
7.1 Real-Time Benchmark Characteristics

We use a set of EEMBC [14] automotive and SD-VBS [57] vision benchmarks (input: sim) as real-time workloads. We profile each benchmark, using the gem5 simulator, to better understand memory characteristics of the benchmarks.

Figure 7a shows the ratio between the number of accessed pages within the main loop and the number of all accessed pages of each benchmark; the pages accessed in the loop are denoted as *critical pages*. To further analyze the characteristics of the critical pages, we profiled L1 cache misses of each critical page to see which pages contribute most to the overall L1 cache misses. *Critical(T98)* shows the ratio of "top" critical pages which contribute to 98% of the L1 cache misses. The same is for *Critical(T90)* except that 90% of the L1 cache misses are considered. As can be seen in the figure, only 38% of all pages, on average, are critical pages, and this number can be as low as 6% (*svm.*) This means that the rest of the pages are accessed during the initialization and other non-time-critical procedures. This ratio is further reduced to 23% of the touched pages if 90% of L1 cache misses are considered.



(a) Critical pages among all touched pages.



(b) *svm* L1 miss distribution of critical pages.

Figure 7 Space and temporal characteristics of application memory pages. Critical pages refer to the touched pages within the main loop of each benchmark.

Note that in our system setup, the private L1 cache misses are directed to the shared L2 cache, which is shared by all cores. Thus, those pages that show high L1 misses likely contribute most to the WCET of the application because they can suffer from high inter-core interference due to contention at the shared L2 cache and/or the shared DRAM. Figure 7b shows how we determine top critical pages for the *svm* benchmark. We rank all pages based on the number of L1 cache misses of each page. In case of *svm*, the top 26 and 16 pages account for 98% and 90% of misses of all critical pages. These pages are 4% and 2% of all the touched pages, respectively, as shown in Figure 7a. This suggests even among the critical pages, certain pages contribute more to WCET than the rest of the critical pages.

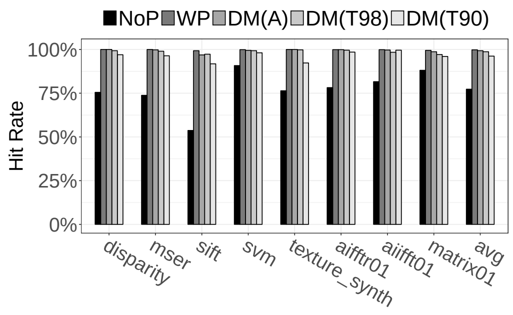
The results show that selective, fine-grained application of deterministic memory can significantly reduce WCETs while minimizing resource waste.

7.2 Effects of Deterministic Memory-Aware Cache

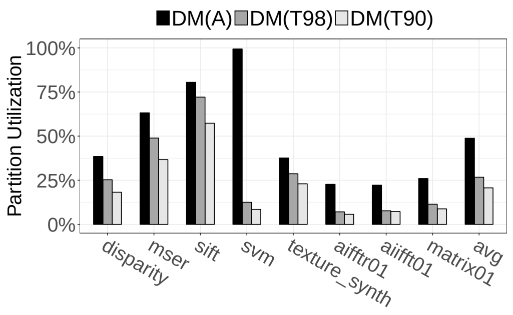
In this experiment, we study the effectiveness of the proposed deterministic memory-aware cache. The basic experimental setup is that we run a real-time task on Core 3 and three instances of a memory intensive synthetic benchmark (*Bandwidth* with write memory access pattern from the IsolBench suite [56]) as best-effort co-runners on Core 0 through 2. Note that the working-set size of the best-effort co-runners is chosen so that the sum of all co-runners is equal to the size of the entire L2 cache. This will increase the likelihood to evict the cache lines of the real-time task if its cache lines are not protected.

We evaluate the system with 5 different configurations: *NoP, WP, DM(A), DM(T98) and DM(T90).* In NoP, the L2 cache is shared among all cores without any restrictions. In WP, the L2 cache is partitioned using the standard way-based partitioning method, where 4 dedicated cache ways are given to each core. In DM(A), the entire address space of the real-time task is marked as deterministic memory, while in DM(T98) and DM(90), only the pages which account for 98% and 90%of the L1 misses, respectively, of the task's critical pages are marked as deterministic. In all DM configurations, each core is given 1/4 of the cache ways for the core's deterministic memory.

Note that, in this experiment, the results for DM(A) will be similar to that of PRETI [32], because, in both systems, a dedicated cache space is guaranteed to a real-time task's entire memory space, while the presence of memory-intensive best-effort co-runners would prevent the real-time task under PRETI from utilizing additional cache space.



(a) L2 hit rate.



(b) Cache partition usage of DM approaches.

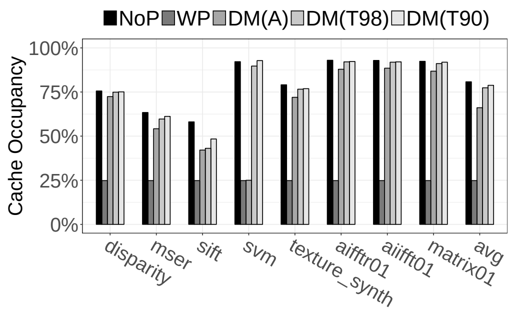
Figure 8 L2 hit-rate and cache space usage (deterministic memory only) of real-time tasks.

Effects on Real-Time Tasks. Figure 8a compares the L2 hit rates of real-time tasks for each system configuration. First, in NoP, the L2 hit rates are low (e.g., 54% for *sift*) because the cache lines of the real-time benchmarks are evicted by the co-running *Bandwidth* benchmarks. In WP, on the other hand, all benchmarks show close to 100% hit rates. This is because the dedicated private L2 cache space (4 out of 16 cache ways = 512KB) is sufficient to hold the working-sets of the real-time benchmarks, which cannot be evicted by the co-runners. The hit rates are also close to 100% in DM(A) because the co-runners are not allowed to evict any of the cache lines allocated for the real-time tasks as their entire memory spaces (thus their cache-lines in the L2) are marked as deterministic memory. In DM(T98) and DM(90), not all pages are marked as deterministic memory. As the result, the co-runners can evict some of the best-effort cache lines of the real-time tasks and this in turn results in slight reduction in the hit rates.

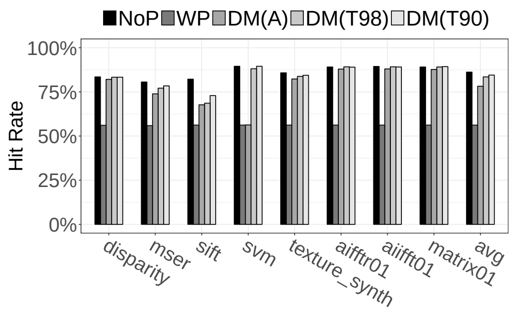
Next, for all DM configurations, we measure the fraction of deterministic memory cache-lines in a real-time task's cache partition by checking *DM* bit in the cache lines in the instrumented gem5 simulator. Figure 8b shows the percentage of the cache lines allocated by the deterministic memory cache lines. On average, only 49%, 27%, and 21% of cache-lines are deterministic memory cache-lines for DM(A), DM(T98), and DM(T90), respectively. Note that when the conventional way partitioning is used (in WP), the unused cache space in the private cache partition is essentially wasted as no other task can utilize it. In the deterministic memory-aware cache, on the other hand, the best-effort tasks can use the non-DM cache lines in the cache partition. Thus, the hit rate of the best-effort tasks can be improved as more cache space will be available to them. This effect will be shown in the following experiment.

Effects on Best-Effort Tasks. To study the effect of deterministic memory-aware cache on realistic best-effort tasks (as oppose the synthetic ones used above), we designed an experiment with the *bzip2* benchmark from SPEC2006 as the best-effort task running on Core 0, and 3 instances of a real-time task running on Core 1 through 3. We chose *bzip2* based on the following selection criteria: 1) It must frequently access the shared cache; 2) It must be sensitive to extra cache space (i.e. the hit rate shall be improved if more cache space is given to the benchmark). The *bzip2* meet both requirements according to a memory characterization study [20] by Intel, which is also confirmed in our simulation setup.

Figure 9 shows the results. Inset (a) shows the percentage of cache space used by *bzip2* for each real-time task pairing, while inset (b) shows its hit rates. Note that in WP, *bzip2* can only use 25% of cache space (512kB out of 2MB), as this is the size of its private cache



(a) Cache space occupied by *bzip2*.



(b) *bzip2* hit rate.

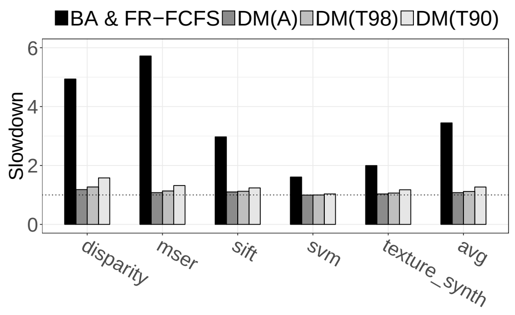
Figure 9 Cache usage and hit rate impact of DM-aware cache to the best-effort task (*bzip2*).

partition. On the other hand, in a deterministic memory-aware cache, *bzip2* can allocate more lines from the private partitions of the other cores which are not marked as deterministic memory cache lines. Consequently, the average hit rate is improved by 39%, 49%, and 50% in DM(A), DM(T98), and DM(T90), respectively, compared with the rate in WP. Note also that more cache lines are allocated by *bzip2* in DM(T98) and DM(T90) compared to DM(A) because more best-effort cache lines can be available for *bzip2* in these configurations. The best-effort cache lines of each core's cache partition are shared among all of the cores, including the core that runs *bzip2* and those that run the real-time tasks. We include the result for NoP to show how much cache space *bzip2* can allocate if there is no restriction. These numbers can also be seen as the upper-bound cache space that *bzip2* can allocate in the deterministic memory-aware cache. By comparing the cache occupancy in DM(90) and NoP (i.e., "free-for-all" sharing), we see that using deterministic memory-aware cache, *bzip2*'s cache space occupancy is close to what we see in NoP.

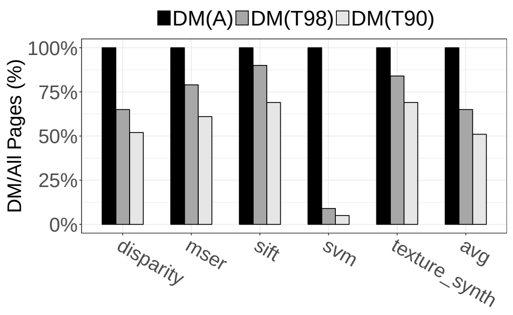
7.3 Effects of Deterministic Memory-Aware DRAM Controller

We evaluate the deterministic memory-aware DRAM controller, using SD-VBS benchmark suite (input: CIF). Note that we increase the input size of the SD-VBS benchmarks to ensure that the working-sets of the benchmarks do not fit in the L2 cache and the memory accesses have to go to the main memory. On the other hand, because the EEMBC benchmarks are cache-fitting and their working-set size cannot be adjusted, we remove them from this experiment. We then re-profile the SD-VBS benchmark with the new inputs, following the method described in 7.1, to determine the critical pages.

The basic setup is the same as in 7.2: We schedule a real-time task on Core 0, while co-schedule three instances of the Bandwidth benchmark as co-runners on Core 1 to 3. The working-set size of the Bandwidth benchmark is configured to be 2x larger than the L2 cache size to induce lots of competing DRAM accesses. We repeat the experiment in the following configurations. In *DM(A)*, *DM(T98)*, and *DM(T90)*, the cache configurations are the same as in 7.2. In addition, each core is given a private DRAM bank for deterministic memory in the DM configurations. The remaining four DRAM banks are shared among the cores for best-effort memory. With the DM-aware OS allocator support described in 4.4, the deterministic memory blocks are allocated on the per-core private banks, and the best-effort regions are allocated on the shared banks. In *BA and FR-FCFS*, the FR-FCFS algorithm is used to schedule the memory accesses to the DRAM, and no OS-level DRAM bank control is applied (i.e., default buddy allocator).



(a) Measured slowdown.



(b) Deterministic memory utilization.

Figure 10 Performance and deterministic memory space impacts of DM-aware DRAM.

Figure 10a shows the normalized slowdown results for different system configurations. Note first that real-time tasks can suffer a significant slowdown in BA&FR-FCFS (by up to 5.7X), while all DM-aware configurations suffer much fewer slowdowns thanks to the two-level scheduling algorithm of our DM-aware memory controller design. Figure 10b shows the ratio between the number pages marked as deterministic and all the pages touched by each real-time task. In *DM(A)*, all the pages of each benchmark are marked as deterministic memory, while in *DM(T90)* only 51% of pages, on average, are marked as deterministic as more pages are allocated in best-effort DRAM banks. This space saving is achieved at the cost of slight execution time increase in real-time benchmarks. These results show how the number of deterministic pages can be used as a parameter to make a trade-off between resource utilization and isolation performance.

8 Related Work

Time-predictable hardware architecture. Time-predictable hardware architecture has long been studied in the real-time community. Edwards and Lee proposed the PRET architecture, which promoted the idea of making time as a first-class citizen in computer architecture [13]. A PRET machine [37] provides hardware-based isolation–featuring a thread interleaved pipeline, scratchpad memory [6] and a bank-privatized PRET DRAM controller [45]–to support strong timing predictability and repeatability. FlexPRET improves the efficiency of PRET with a flexible hardware thread scheduler that guarantees hardware isolation of hard real-time threads while allowing soft real-time threads to efficiently utilize the processor pipeline [67]. T-CREST [48], MERASA [54] and parMERASA [53] projects also have investigated time-predictability focused core architecture, cache, cache coherence protocol, system-bus, and DRAM controller designs [49, 23, 47, 21, 42, 43, 33, 34]. There are also many other proposals, which focus on improving timing predictability of each individual shared hardware component–such as time predictable shared caches [61, 62, 32], hybrid SPM-cache architecture [65], and predictable DRAM controllers [60, 17, 29, 12]. In most proposals, the basic approach has been to provide space and time partitioning of hardware resources to each critical real-time task or the cores that are designated to execute such tasks. Thus, CPU-centric abstractions such as task priority and core/task id are commonly used information sources, which are utilized by these hardware proposals in managing the hardware resources. However, when it comes to managing memory related hardware resources, these CPU-centric abstractions can be too coarse-grained, which make efficient resource management difficult. This is because neither all tasks are time-critical (and thus requires hardware isolation support), nor all memory blocks of a critical task are necessarily time-critical, as we have shown in Section 7.1.

Memory address based real-time architecture designs. The basic idea of using physical memory address in hardware-level resource management has been explored in several prior works. Kumar et al. proposed a criticality-aware cache design, which uses a number of hardware range registers to declare critical memory regions. Its Least Critical (LC) cache replacement algorithm then prioritize the cache-lines of critical memory regions over others to ensure predictable cache performance for a single-core, fixed-priority preemptive scheduled system setup [30]. Kim et al. similarly declare critical memory regions using a set of hardware range registers to distinguish memory-criticality at the DRAM controller level [26]. While our approach is also based on memory address based criticality determination, our deterministic memory abstraction is designed to be utilized by the entire memory hierarchy whereas the prior works focused on a single individual hardware resource management. Furthermore, a key contribution of our approach is that, in our approach, memory criticality is determined at the page granularity by utilizing memory management unit (MMU), which enables more flexible and fine-grained (page-granularity) memory criticality control. In contrast, the prior works may be limited by the number of available hardware range registers in declaring critical memory regions. As such, our MMU-based approach is compatible with high-performance processors and general-purpose OSs such as Linux, whereas the prior works primarily focus on MMU-less processors and RTOSs. We would like to note, however, that our MMU-based deterministic memory abstraction can be integrated into and leveraged by these prior works. The deterministic memory abstraction provides a general framework for the entire memory-hierarchy and thus is complementary to the prior works.

OS-level shared resource management. In many OS-level resource management approaches, MMU has been a vital hardware component that the OS leverages for implementing certain memory management policies for real-time systems. Page-coloring is a prime example that has been used to partition shared cache [35, 36, 66, 50, 11, 59, 39, 25], DRAM banks [63, 38, 51] and even TLB [41] by selecting certain physical addresses (cache color, DRAM bank, etc.) in allocating pages. However, in most OS-level resource management approaches, shared resources are allocated at the granularity of task or core, which is too coarse-grained and therefore can result in resource under-utilization problems. Furthermore, these OS-level resource management approaches have fundamental limitations because they generally cannot directly influence important resource allocation and scheduling decisions done by the underlying hardware due to the lack of a generalized abstraction that allows such cross-layer communication. We address these limitations by proposing the deterministic memory abstraction, which enables close collaboration between the OS and the underlying hardware components in the memory hierarchy to achieve efficient and predictable resource allocation and scheduling. To the best of our knowledge, we are first to propose to encode each individual memory page's time criticality in the page's page table entry, which is then passed through the entire memory hierarchy to enable system-wide, end-to-end memory-criticality-aware resource management.

9 Conclusion and Future Work

In this paper, we proposed a new memory abstraction, which we call *Deterministic Memory*, for predictable and efficient resource management in multicore. We define deterministic memory as a special memory space where the platform–OS and hardware architecture–guarantees small and tightly bounded worst-case access timing.

We presented OS and architecture extensions to efficiently support the deterministic memory abstraction. In particular, we presented a deterministic memory-aware cache design that leverages the abstraction to improve the efficiency of shared cache without losing isolation benefits of traditional way-based cache partitioning. In addition, we proposed a deterministic memory-aware DRAM controller which effectively reduces the necessary core-private DRAM bank space while still providing good isolation performance. We implemented the proposed OS extension on a real operating system (Linux) and implemented the proposed architecture extensions on a cycle-accurate full-system simulator (gem5).

Evaluation results show the feasibility and effectiveness of deterministic memory based cross-layer resource management. Concretely, by using deterministic memory, we achieved the same degree of strong isolation while using 49% less cache space, on average, than the conventional way-based cache partitioning method. Similarly, we were able to reduce required private DRAM bank space while achieving comparable isolation performance for DRAM intensive real-time applications, compared to a baseline real-time DRAM controller.

We are currently working on implementing the proposed architecture extensions on a FPGA using an open-source RISC-V based multicore platform [52]. We also plan to develop methodologies and tools to identify "optimal" deterministic memory blocks that maximize the overall schedulability.

References

1 Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Syst. Symp. (RTSS)*, 48(5):499–526, 2012.

2 ARM. *AMBA AXI and ACE Protocol Specification*, 2013.

3 ARM. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R Edition*, 2014.

4 ARM. *Cortex™-A17 Technical Reference Manual, Rev: r1p1*, 2014.

5 N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

6 R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Int. Symp. Hardware/Software Codesign (CODES+ISSS)*, pages 73–78. ACM, 2002.

7 N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Comput. Architecture News*, 2011.

8 Alan Burns and Robert Davis. Mixed criticality systems - A review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

9 Certification Authorities Software Team. CAST-32A: Multi-core Processors (Rev 0). Technical report, Federal Aviation Administration (FAA), November 2016.

10 TIS Committee. Executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

11 X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. In *European Conf. Comput. Syst. (EuroSys)*. ACM, 2011.

12 Leonardo Ecco and Rolf Ernst. Improved dram timing bounds for real-time dram controllers with read/write bundling. In *Real-Time Systems Symposium (RTSS)*, pages 53–64. IEEE, 2015.

13 S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conf. (DAC)*, 2007.

14 EEMBC benchmark suite. www.eembc.org.

15 Freescale. *e500mc Core Reference Manual*, 2012.

16 Gem5: O3CPU. http://gem5.org/O3CPU.

17 S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Design, Automation and Test in Europe (DATE)*, 2013.

18 A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. Udipi. Simulating DRAM controllers for future system architecture exploration. In *Int. Symp. Performance Analysis of Syst. and Software (ISPASS)*, 2014.

19 J.L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Architecture News*, 34(4):1–17, 2006.

20 Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf, 2010.

21 Javier Jalle, Jaume Abella, Eduardo Quinones, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. AHRB: A high-performance time-composable AMBA AHB bus. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 225–236. IEEE, 2014.

22 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *Real-Time Syst. Symp. (RTSS)*, pages 207–217. IEEE, 2014.

23 Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Real-Time Networks and Systems (RTNS)*, pages 55–64. ACM, 2013.

24 H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. (Raj) Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, 2014.

25 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Real-Time Syst. (ECRTS)*, pages 80–89. IEEE, 2013.

26 Hokeun Kim, David Bromany, Edward Lee, Michael Zimmer, Aviral Shrivastava, Junkwang Oh, et al. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 317–326. IEEE, 2015.

27 Namhoon Kim, Bryan C Ward, Micaiah Chisholm, Cheng-Yang Fu, James H Anderson, and F Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 1–12. IEEE, 2016.

28 O. Kotaba, J. Nowotsch, M. Paulitsch, S. Petters, and H Theiling. Multicore in real-time systems temporal isolation challenges due to shared resources. In *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Syst.*, 2013.

29 Y. Krishnapillai, Z. Wu, and R. Pellizzoni. ROC: A Rank-switching, Open-row DRAM Controller for Time-predictable Systems. In *Euromicro Conf. Real-Time Syst. (ECRTS)*, 2014.

30 NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. Cache design for mixed criticality real-time systems. In *Computer Design (ICCD)*, pages 513–516. IEEE, 2014.

31 Robert Leibinger. Software architectures for advanced driver assistance systems (ADAS). In *Int. Workshop on Operating Syst. Platforms for Embedded Real-Time Applicat. (OSPERT)*, 2015.

32 Benjamin Lesage, Isabelle Puaut, and André Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Real-Time and Network Systems (RTNS)*, pages 171–180. ACM, 2012.

33 Yonghui Li, Benny Akesson, and Kees Goossens. Dynamic command scheduling for real-time memory controllers. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 3–14. IEEE, 2014.

34 Yonghui Li, Benny Akesson, and Kees Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems*, pages 1–55, 2015.

35 J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled cache predictability for real-time systems. In *Real-Time Technology and Applicat. Symp. (RTAS)*. IEEE, 1997.

36 J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Comput. Architecture (HPCA)*. IEEE, 2008.

37 I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Comput. Design (ICCD)*. IEEE, 2012.

38 L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 367–376. ACM, 2012.

39 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*. IEEE, 2013.

40 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Euromicro Conf. Real-Time Syst. (ECRTS)*, 2014.

41 Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via TLB coloring. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 3–13. IEEE, 2015.

42 M. Paolieri, E. Quiñones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Comput. Architecture News*. ACM, 2009.

43 M. Paolieri, E. Quiñones, J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Syst. Letters, IEEE*, 1(4):86–90, 2009.

44 R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking (MNCN)*, January 1998.

45 J. Reineke, I. Liu, H.D. Patel, S. Kim, and E.A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2011.

46 S. Rixner, W. J Dally, U. J Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *ACM SIGARCH Comput. Architecture News*, volume 28, pages 128–138. ACM, 2000.

47 J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Syst. Symp. (RTSS)*, pages 49–60, 2007.

48 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

49 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W Probst, Sven Karlsson, Tommy Thorn, et al. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Syst.*, volume 18, pages 11–21, 2011.

50 L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Int. Symp. Microarchitecture (MICRO)*. IEEE, 2008.

51 N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Sci. and Eng. (CSE)*, pages 685–692. IEEE, 2013.

52 The berkeley out-of-order RISC-V processor code repository. `https://github.com/ucb-bar/riscv-boom`.

53 Theo Ungerer, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, Joao Fernandes, Pavel G Zaykov, Zlatko Petrov, B Boddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA–Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *Digital System Design (DSD)*, pages 363–370. IEEE, 2013.

54 Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. Merasa: Multicore execution of hard real-time applicat. supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. `doi:10.1109/MM.2010.78`.

55 P. Valsan and Heechul Yun. MEDUSA: A predictable and high-performance DRAM controller for multicore based embedded systems. In *Cyber-Physical Syst., Networks, and Applicat. (CPSNA)*. IEEE, 2015.

56 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*. IEEE, 2016.

57 Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego vision benchmark suite. In *Int. Symp. Workload Characterization (ISWC)*, pages 55–64. IEEE, 2009.

58 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Syst. Symp. (RTSS)*, pages 239–243. IEEE, 2007.

59 B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conf. Real-Time Syst. (ECRTS)*, 2013.

60 Z. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Real-Time Syst. Symp. (RTSS)*, 2013.

61 J. Yan and W. Zhang. Time-predictable L2 cache design for high-performance real-time systems. In *Embedded and Real-Time Computing Syst. and Applicat. (RTCSA)*, pages 357–366. IEEE, 2010.

62 Jun Yan and Wei Zhang. Time-predictable multicore cache architectures. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 3, pages 1–5. IEEE, 2011.

63 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, 2014.

64 H. Yun and G. Yao. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, 2013.

65 W. Zhang and Y. Ding. Hybrid spm-cache architectures to achieve high time predictability and performance. In *Application-Specific Syst., Architectures and Processors (ASAP)*, pages 297–304. IEEE, 2013.

66 X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European Conf. Comput. Syst. (EuroSys)*, 2009.

67 Michael Zimmer, David Broman, Chris Shaver, and Edward Lee. FlexPRET: A processor platform for mixed-criticality systems. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 101–110. IEEE, 2014.

Worst-case Stall Analysis for Multicore Architectures with Two Memory Controllers

Muhammad Ali Awan
CISTER Research Centre and ISEP, Porto, Portugal
muaan@isep.ipp.pt
https://orcid.org/0000-0001-5817-2284

Pedro F. Souto
University of Porto, Faculty of Engineering and CISTER Research Centre, Porto, Portugal
pfs@fe.up.pt
https://orcid.org/0000-0002-0822-3423

Konstantinos Bletsas
CISTER Research Centre and ISEP, Porto, Portugal
ksbs@isep.ipp.pt
https://orcid.org/0000-0002-3640-0239

Benny Akesson
Embedded Systems Innovation, Eindhoven, the Netherlands
benny.akesson@tno.nl
https://orcid.org/0000-0003-2949-2080

Eduardo Tovar
CISTER Research Centre and ISEP, Porto, Portugal
emt@isep.ipp.pt
https://orcid.org/0000-0001-8979-3876

Abstract

In multicore architectures, there is potential for contention between cores when accessing shared resources, such as system memory. Such contention scenarios are challenging to accurately analyse, from a worst-case timing perspective. One way of making memory contention in multicores more amenable to timing analysis is the use of memory regulation mechanisms. It restricts the number of accesses performed by any given core over time by using periodically replenished per-core budgets. Typically, this assumes that all cores access memory via a single shared memory controller. However, ever-increasing bandwidth requirements have brought about architectures with multiple memory controllers. These control accesses to different memory regions and are potentially shared among all cores. While this presents an opportunity to satisfy bandwidth requirements, existing analysis designed for a single memory controller are no longer safe.

This work formulates a worst-case memory stall analysis for a memory-regulated multicore with two memory controllers. This stall analysis can be integrated into the schedulability analysis of systems under fixed-priority partitioned scheduling. Five heuristics for assigning tasks and memory budgets to cores in a stall-cognisant manner are also proposed. We experimentally quantify the cost in terms of extra stall for letting all cores benefit from the memory space offered by both controllers, and also evaluate the five heuristics for different system characteristics.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Real-time operating systems, Computer systems organization → Real-time system architecture

Keywords and phrases multiple memory controllers, memory regulation, multicore

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.2

Supplement Material ECRTS Artifact Evaluation approved artifact available at
https://dx.doi.org/10.4230/DARTS.4.2.5

Funding This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (CEC/04234).

1 Introduction

The strong trend towards increasing integration in hardware for embedded real-time systems has led to multicores becoming mainstream platforms of choice for such systems. Multicores have significant advantages in terms of computing power, energy usage and weight over single-cores. Yet, one issue with multicores is that worst-case timing analysis becomes more complicated. In particular, the fact that multiple cores contend for the same shared system resources (buses, caches, memory) must be accounted for [8].

Focusing specifically on the problem of main memory contention, we note various research efforts [21, 22, 15, 10, 5, 11, 13, 20, 14, 3] that employ *memory regulation* to make the memory access patterns of the different cores more amenable to worst-case timing analysis. Under memory regulation schemes, each core gets an associated periodically-replenished memory access budget. When a core attempts to issue more memory accesses than its budget, it gets temporarily stalled, until the next replenishment.

However, engineering practice forges ahead and analysis has to catch up. In recent years, in response to memory bandwidth often becoming a performance bottleneck, multicore chips that integrate, not one, but two memory controllers, have become commercially available. In such platforms, both controllers are accessible by all cores, with little to no difference in latency. Examples include various multicore processors from the NXP QorIQ series [16], ranging from the P5020 with 2 cores to the P4080 with 8 cores. For existing approaches to apply to systems with multiple controllers, one could statically map cores to memory controllers and apply the analyses to each partition independently. This simple approach efficiently reduces contention between cores. Still, it may be hard to find a partition such that no tasks depend on data from the memory space of the other memory controller. Core-to-controller partitioning also reduces flexibility in bandwidth allocation, as a partition's bandwidth requirements must be met by just the associated memory controller. In cases when no such partitions can be found, there are currently no good solutions, because existing approaches can be *unsafe* when applied to platforms with two controllers. The reason is that the worst-case memory access pattern for each controller in isolation will not necessarily lead to the worst-case stall, as we demonstrate in Section 5. This reality motivated the present work, whose main contributions are the following:

First, we show via counter-examples that existing techniques for upper-bounding the memory stall, conceived for memory-regulated architectures with a single memory controller, are not necessarily safe in the presence of multiple controllers. Our second and more important contribution is new worst-case memory stall analysis for architectures with two memory controllers, shared by all cores. This analysis, which presumes fixed task-to-core partitioning and fixed-priority scheduling, can then be integrated to the schedulability analysis for the system. Finally, we explore five different stall-cognisant heuristics for combined memory-bandwidth-to-core assignment and task-to-core assignment and evaluate their performance in terms of schedulability via experiments with synthetic task sets capturing different system characteristics. These experiments also highlight the performance implications of having fully shared memory controllers vs. partitioning the controllers to different cores, in cases when the latter arrangement would be viable from the application perspective (i.e., no data sharing across memory domains).

Next, in Section 2, we discuss related work. Section 3 defines our system model and Section 4 discusses some relevant existing results from the single-controller case. Section 5 contains our analysis. Section 6 describes five proposed stall-cognisant task-to-core assignment heuristics. Section 7 provides an experimental evaluation of our analysis and heuristics in terms of theoretical schedulability using synthetic task sets. Section 8 concludes the paper.

2 Related work

Several software-based approaches for mitigating memory interference in multi-core platforms [21, 22, 15, 10, 5, 11, 3] have been proposed in recent years. These approaches consider a periodic server implemented in software that manages the memory budgets of the cores. This is combined with run-time monitoring through performance counters that keep track of the number of memory accesses and with an enforcement mechanism that suspends tasks whenever they exhaust their budget. Our work is similar to these, as it exploits such a memory throttling mechanism to enforce budgets on memory requests.

The memory regulation techniques used to mitigate the interference on shared memory controllers introduce new stalls and the existing analyses are unsafe unless adapted to account for them. Some efforts in this direction exist for partitioned fixed-priority scheduling [21, 13] and hierarchical scheduling in [5]. Mancuso et al. [13], under their Single-Core Equivalence framework [18], addressed the problem of fixed-priority partitioned schedulability on a multicore. They employ the periodic software-based memory regulation mechanism MemGuard [22] to ensure that each core gets an equal share of memory bandwidth in each regulation interval (or period) and stalls until the end of the regulation period once the budget has been depleted. Such stalls, resulting from the memory regulation together with contention stalls are integrated into the schedulability analysis in [13].

Even if equal sharing of memory bandwidth is simple and facilitates porting applications from a single-core to multi-core platforms (by making the analysis akin to that for a single-core), it is inefficient when the memory requirements of the applications on different cores are diverse. Yao et al. [20], and Pellizzoni and Yun [17] generalise the arrangement along with the analysis to uneven memory budgets per core. The former approach considers round-robin memory arbitration, whereas the latter proposes a new analysis for First-Ready First Come First Served memory scheduling. Recently, Mancuso et al. [14] improved their memory stall analysis by considering the exact memory bandwidth distribution on other cores. However, all these approaches are designed to work with a *single memory controller* and are unsafe with more than one memory controller. The reason is that the worst-case memory access pattern for each controller in isolation no longer necessarily leads to the worst-case stall, as we show in Section 5. In contrast, our work provides a worst-case memory stall analysis for a memory-regulated multicore platform with two memory controllers and incorporates this stall analysis in the schedulability analysis for fixed-priority partitioned preemptive scheduling. We also present five memory bandwidth allocation and task-to-core assignment heuristics.

To summarise, existing works on memory regulation rely on an assumption of a single memory controller. Here, we expand the state-of-the-art by proposing memory stall analysis, when each core can access two controllers, facilitating data sharing among applications and allowing more flexible use of bandwidth. We allow uneven distribution of the memory bandwidth of each controller to available cores. Each core is scheduled under fixed-priority preemptive scheduling, assuming a round-robin memory arbitration policy on both controllers.

3 System Model

We consider a platform with m identical cores (P_1 to P_m) and 2 memory controllers on the same chip, both uniformly accessible by all cores. The sets of memory regions accessible by the two controllers are non-overlapping. Examples of platforms with 2-8 identical cores and two memory controllers include NXP QorIQ P-series P4040, P4080, P5020 and P5040 [16].

Assume a set of n sporadic tasks, τ_1 to τ_n. Each task has a minimum interarrival time T_i, a deadline $D_i \leq T_i$, and a worst-case execution time (WCET) of C_i. Like Yao et al. [20], we assume that CPU computation and memory access do not overlap in time. Each task can access memory via both controllers. Therefore, $C_i = C_i^e + C_i^{m1} + C_i^{m2}$, where C_i^e is the worst-case CPU computation time and C_i^{m1} and C_i^{m2} are the worst-case total memory access times of a task via each respective controller in isolation.

The tasks are partitioned to the cores (no migration) and fixed-priority scheduling is used. For the memory controllers and their interconnects, we assume a round-robin policy [22, 20]. The last-level cache (furthest from the cores) is either private or partitioned to each core. Like Yao et al. [20], we assume that access to main memory is regulated, e.g., by Memguard [22] or in hardware. We also require performance monitoring counters to count the number of memory accesses issued to each controller from each core. As in [20], we assume each memory access takes a constant time L. This allows us to specify P and C_i^e, C_i^{m1} and C_i^{m2} as multiples of L. Our model is agnostic w.r.t. the points in time when memory accesses may occur within the activation of a task and hence imposes no particular programming model.

Memory accesses are regulated as follows. Each core i has a *memory access budget* $Q1_i$ for memory controller 1, which is the maximum allowed memory access time (measured in multiples of L) via that controller, within a *regulation period* of length P. Likewise, it has a budget $Q2_i$ for controller 2. These budgets are set at design time and may be different. A core i that consumes its memory access budget for a given memory controller within a regulation period is *stalled* until the start of the next regulation period[1]. Regulation periods on all cores are synchronised. The *memory bandwidth share* of core i on controller 1 is $b1_i = \frac{Q1_i}{P}$. Similarly for $b2_i$ and controller 2. By design, $\sum_i b1_i \leq 1$ and $\sum_i b2_i \leq 1$, i.e., the bandwidth of any controller is not overcommitted.

4 Relevant existing results from the single-controller case

We now summarise some existing results from [20], for a similar, albeit single-controller, system, in order to later show why those do not apply, and new analysis is needed.

The technique in [20] calculates a worst-case stall term for each task, which is added to the right hand side of the standard worst-case response time (WCRT) recurrence relation for fixed priorities. For ease of presentation, the authors assume that there is a single task running on the core under consideration. Later on, for the case when many tasks are assigned to a core, they explain how to equivalently model the considered task τ_i and all higher-priority tasks as a single synthetic task, in order to apply their stall analysis and derive the worst-case stall term for τ_i. Below, we similarly assume a single task per core.

A memory request may stall either (i) because of requests from other cores, contending for the memory controller simultaneously (a case of **contention stall**) or (ii) because the issuing core has exhausted its budget for the current regulation period (a **regulation stall**).

[1] On practical grounds, we assume that a core is stalled immediately after the Q^{th} memory access in a regulation period via the respective controller is served. Yao et al [20], more generously, assume that it is stalled immediately before attempting a $(Q+1)^{th}$ access within the same regulation period.

Yao et al. identify worst-case patterns for memory accesses and computation within a single regulation period, characterised by maximum stall with the fewest memory accesses. Next, they use these patterns as main "building blocks" for the worst-case pattern for the entire task activation, over multiple regulation periods. In more detail:

Case $b_i \leq 1/m$ (regulation dominant). If $b_i \leq 1/m$, i.e., if the task's bandwidth share is "fair" at most, then a task incurs worst-case stall when all its memory accesses are clustered at the start of its activation, before any computation. Another pessimistic assumption is that the task is released just after a regulation stall, so it waits for $(P - Q_i)$ until the next regulation period. The task will incur a stall of $(P - Q_i)$ within each of the next $\lfloor \frac{C_i^m}{Q} \rfloor$ regulation periods; whether this is entirely due to a regulation stall or partially also due to contention from other cores is irrelevant. Afterwards, any remaining memory accesses (which are too few to trigger a regulation stall), can each incur a worst-case contention stall of (m-1), i.e., one contending access from each other core due to round robin arbitration.

Case $b_i > 1/m$ (contention dominant). In this case, the smallest number of memory accesses per period a core must issue to get the maximum stall is $RBS \overset{\text{def}}{=} \frac{P_i - Q_i}{m-1}$, and occurs when the remaining budget is shared evenly among the other cores. From the assumption of the case, $b_i > 1/m$, it follows that $RBS < Q_i$. Therefore, the worst-case pattern for one regulation period involves $c_i^m = RBS$ accesses, each suffering a maximum contention stall of $(m-1)$, for a total stall of $P - Q_i$. This leaves $Q_i - RBS$ time units not filled by memory accesses or respective stalls. These are filled with computation; if memory accesses were added instead, they would incur no stall. To bound the stall for the entire task activation, this pattern is applied to as many regulation periods as possible. Two subcases exist: either memory accesses or computation will run out first.

Due to space constraints, we refer to [20] for details. Meanwhile, some insights driving Yao's analysis, for single-controller systems, are codified via the following lemmas from [20]:

▶ **Lemma 1.** *Considering the stall of a core due to memory regulation alone, the worst-case memory access pattern of one task is when all accesses within the task are clustered, and the stall is upper bounded by $P - Q_i$ for each regulation period P.*

▶ **Lemma 2.** *If the memory is not overloaded and the regulation periods are the same and synchronized, the stall due to inter-core memory contention alone on each core i with assigned budget Q_i is upper-bounded by $P - Q_i$ for every regulation period P.*

▶ **Lemma 3.** *Considering the contention stall alone, the maximum stall for core i with budget Q_i is obtained when the remaining budget $P - Q_i$ is evenly distributed among all other cores and they generate the maximum amount of accesses.*

5 Analysis

In this section, we formulate the main contribution of this paper: a stall analysis for multicores with two memory controllers, which leverages on Yao et al [20] stall and schedulability analysis for multicores with a single memory controller. First, we look at Lemmas 1 to 3 and Yao's analysis in general, and examine what holds over from [20] and what does not. For readability, we omit the core (task) index, since it is implied. Table 1 summarizes the symbols used.

Table 1 Symbols used in the analysis.

$Q1, Q2$	memory budget on controllers 1 and 2, respectively
C^{m1}, C^{m2}	maximum number of memory accesses via controllers 1 and 2, respectively
C^e	worst-case computation time
P	regulation period
m	number of cores
$b1, b2$	core memory bandwidth shared on controllers 1 and 2, respectively
$RBS1, RBS2$	remaining budget share on controllers 1 and 2, respectively
c^{m1*}, c^{m2*}	worst-case number of accesses per period in contention-dominant case
$K1^*$	number of regulation periods of phase 1 in contention-dominant case
$\hat{C}^e, \hat{C}^{m1}, \hat{C}^{m2}$	task computation parameters after phase 1 (in contention-dominant case)
$\Delta\rho^*$	worst-case reduction in regulation stalls w.r.t. maximum regulation stalls in the third case (regulation is dominant only for one controller)
ΔC^e	additional "computation" added to contention-only phase by reducing the number of regulation stalls by 1
ΔC_c^{m2*}	additional number of contention stalls **required** when moving ΔC^e to ensure that the total stall is larger with one less regulation stall on controller 1
$\Delta C_c^{m2}(max)$	maximum number of additional contention stalls obtained by moving ΔC^e to the contention-only phase
$\Delta C_c^{m2}(min)$	minimum number of additional contention stalls obtained by moving ΔC^e to the contention-only phase
$r^m = \frac{C^{m2}}{C^{m1}}$	ratio of memory accesses to each controller
$C_{\bar{c}}^{m2}$	number of memory accesses via controller 2 **without** contention
$single()$	worst-case single controller stall according to Yao's analysis, ignoring the regulation stall at the beginning of the execution

5.1 What holds over from Yao's analysis and what does not

When we have multiple controllers, with an assigned memory budget Qj for each, Lemma 1 can be generalized as follows:

▶ **Lemma 4.** *Considering the stall of a core due to memory regulation alone* ***on controller j, with budget Qj****, the worst-case memory access pattern of one task is when all accesses* ***via controller j*** *within the task are clustered, and the stall is upper bounded by $P - Qj$ for each regulation period P.*

A corollary of this lemma is that the regulation stall on controller j is maximum when there are no memory accesses to a second controller in that period. Note also that a core can only regulation-stall on at most one memory controller in a given regulation period.

With multiple controllers Lemmas 2 and 3 apply to *each controller separately.* Furthermore, because a core may access memory via multiple controllers in a single regulation period, a consequence of Lemma 2 is the following:

▶ **Lemma 5.** *If the memory is not overloaded and the regulation periods are the same and synchronized, the stall due to inter-core memory contention alone on each core i with assigned budget Qj_i on controller j is upper-bounded by $\min\left(\sum_j (P - Qj_i), \frac{P}{m} \cdot (m-1)\right)$ for every regulation period P.*

When there are multiple memory controllers, the maximum contention stall may occur when there are accesses via more than one controller. The first argument to the min operator in

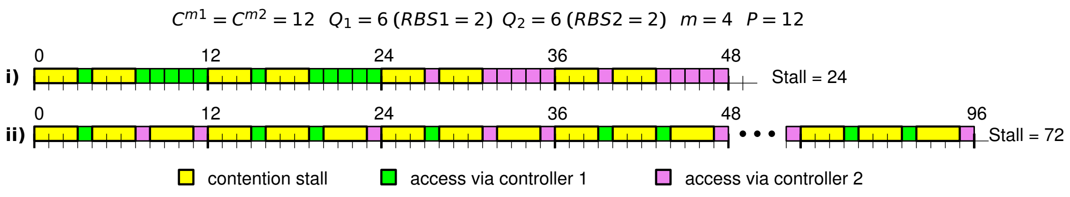


Figure 1 As shown in this example, the worst-case total stall is when there are memory accesses via more than one controller in the same regulation period.

the above expression sums up the contention stall from each controller according to Lemma 2. The second argument expresses the fact that no more than P/m accesses (irrespective via which controller) can all suffer the worst-case per-access contention stall of $(m-1)$ because of round robin arbitration. Both terms independently bound the contention stall.

When there are multiple shared controllers and we try to upper-bound the stall over multiple regulation periods, Yao's analysis may not be safe, i.e., it may underestimate the worst-case stall, as illustrated by the example of Figure 1. Execution i) has the worst-case stall, according to Yao's stall analysis, when in a regulation period all memory accesses are via the same controller. In each period, the first two memory accesses suffer the maximum stall. However the remaining 4 memory accesses suffer no stall, because the maximum stall in every regulation period is 6, $P - Qi$, and it occurs in the first two memory accesses of the respective regulation period. Execution ii) shows the worst-case stall when there are accesses via both controllers in the same period. In each period, we have 2 memory accesses *via each controller* and each of these accesses suffers the maximum contention stall, $m-1$. This is because the contention stall on accesses via one controller does not affect the contention stall on accesses via the other controller. Thus, in execution ii) all memory accesses suffer the maximum contention stall, whereas in execution i) only a third does.

5.2 Two-controller Task Stall Analysis

Having shown the need for a new analysis, we consider several cases depending on the values of $b1$ and $b2$. Some entail sub-cases. More specifically, we consider 3 cases:

1. $b1 \leq \frac{1}{m} \wedge b2 \leq \frac{1}{m}$
2. $b1 > \frac{1}{m} \wedge b2 > \frac{1}{m}$
3. remaining cases, i.e. $(b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}) \vee (b1 > \frac{1}{m} \wedge b2 \leq \frac{1}{m})$

5.2.1 Case 1: $b1 \leq \frac{1}{m} \wedge b2 \leq \frac{1}{m}$

In this case, for each controller, the worst case occurs when there is a regulation stall, as shown in [20]. By Lemma 4, the following execution suffers the worst-case stall. In a first phase, there is the longest sequence of consecutive periods with regulation stalls on controller 1, followed by a second phase consisting of the longest sequence of consecutive periods with regulation stalls on controller 2. Finally, there is a third phase with the remaining memory accesses via each controller, $C^{mi} \bmod Qi$, that suffer the maximum contention stall per memory access, $m-1$, and any computation. Because in each of the two first phases all memory accesses are via a single controller, we can use Yao's stall analysis to compute an upper bound on the stall in each of these phases. The upper bound of the total stall can then be computed by adding the upper bounds for each phase. I.e.:

$$\begin{aligned}Stall =& single(C^m = \left\lfloor \frac{C^{m1}}{Q1} \right\rfloor \cdot Q1, C^e = 0, Q = Q1, P = P, m = m) \\ &+ single(C^m = \left\lfloor \frac{C^{m2}}{Q2} \right\rfloor \cdot Q2, C^e = 0, Q = Q2, P = P, m = m) \\ &+ (C^{m1} \bmod Q1 + C^{m2} \bmod Q2) \cdot (m - 1) \end{aligned} \tag{1}$$

where $single()$ is the stall based on Yao's (single controller) stall analysis for the respective set of parameter values [20].

5.2.2 Case 2: $b1 > \frac{1}{m} \wedge b2 > \frac{1}{m}$

In this case, according to Yao's analysis, for each controller, the worst case occurs when there is maximum contention stall in a regulation period with the minimum number of memory accesses. However, as shown in Figure 1, in this case the worst-case stall may occur when a task accesses memory via different controllers in the same regulation period. Therefore, the worst-case memory access pattern of a task in this case has 3 phases, as illustrated in Figure 2 i):

Phase 1. In this phase, every regulation period incurs the maximum contention stall. This phase terminates when the task runs out of memory accesses via some controller, and therefore cannot sustain the maximum contention stall any more. In Figure 2 i), this phase spans the two first periods, and, in each period, there are $RBS1$ and $RBS2$ memory accesses via the respective controller.

Phase 3. In this phase, all accesses are via a single controller. This phase may not exist, if the task runs out of memory accesses via both controllers in the same regulation period. In Figure 2 i), this is the 4th and last period and has memory accesses only via controller 1.

Phase 2. This "middle" phase may also not exist, but if it exists, it has only one regulation period. In this phase, we have memory accesses via both controllers, but either there are not enough memory accesses via at least one of the controllers to ensure the maximum contention stall in that period, or there is not enough execution to fill the complete period. In Figure 2 i), this is the 3rd period, and has only one memory access via controller 2.

According to Lemma 5, there are two main cases for the maximum contention stall in a regulation period. We analyse each of these cases separately.

5.2.2.1 Sub-case 1: $(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$

In this case, the maximum contention stall in a regulation period occurs when a task performs $RBS1$ memory accesses via controller 1 and $RBS2$ memory accesses via controller 2. Therefore, the maximum stall per period is $(RBS1+RBS2)\cdot(m-1) = (P-Q1)+(P-Q2)$. Because the task is non preemptive and $(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$, by the definition of the sub-case, there is a "hole" of size $P - (RBS1 + RBS2) \cdot m$ that must be filled with execution, i.e. either computation or memory accesses. An execution in which computation fills as many of these holes as possible suffers the maximum stall, because any additional memory accesses in these periods suffer no contention stall. This will minimize the number of memory accesses without contention in Phase 1, increasing the number of memory accesses in latter phases, and possibly their stall. Similar reasoning can be applied to Phase 2, as well.

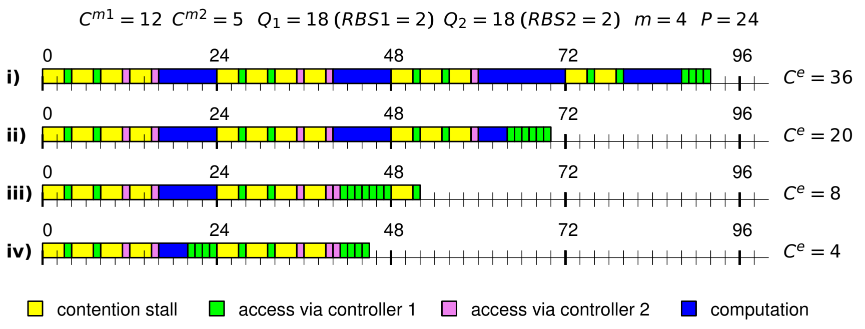


Figure 2 Example execution patterns with worst case stall, for the contention-dominant case when $(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$.

Figure 2 illustrates an execution pattern that leads to the worst-case stall, based on the above observations. In execution i) there is enough computation to fill in the holes in Phases 1 (the first two periods) and 2. However, there is not enough computation to ensure that all memory accesses suffer contention: in the 4th and last period, which belongs to Phase 3, there are 4 memory accesses via controller 1 that do not suffer any contention. In execution ii) there is not enough computation to fill the holes in Phase 2, and therefore, we have 6 memory accesses via controller 1 in Phase 2, the 3rd period, that do not suffer any contention, and there is no 3rd Phase. In execution iii) there is no Phase 2, because all memory accesses via controller 2 are used to fill the holes in Phase 1. Phase 3 consists only of a single memory access via controller 1. Finally, in execution iv) there is not enough computation, and Phase 1, like Phase 2, has only one period, and there is no Phase 3.

It can be shown, by case analysis, that in any of these executions swapping any computation or memory access in one regulation period with computation or memory accesses in later regulation periods does not lead to an increase in the total stall, and therefore the execution pattern shown suffers the maximum stall. The following stall analysis is based on the execution pattern shown in Figure 2.

In order to reuse the analysis in other cases below, let c^{m1*} and c^{m2*} be the minimum values of c^{m1} and c^{m2}, respectively, that maximize the contention stall in a regulation period, assuming that any holes are filled with computation. Note that by Lemma 5, it must be $c^{m1*} \leq RBS1$ and $c^{m2*} \leq RBS2$. In this sub-case, they are $RBS1$ and $RBS2$, respectively.

In our analysis, we consider Phase 1 separately from the remaining phases, if any.

Phase 1 stall. In Phase 1, the contention stall in every regulation period is maximum and equal to $(c^{m1*} + c^{m2*}) \cdot (m - 1)$. The total stall in this phase is:

$$Stall1 = K1^* \cdot (c^{m1^*} + c^{m2^*}) \cdot (m - 1) \tag{2}$$

$$\text{where: } K1^* = min\left(\left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor, \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor, \left\lfloor \frac{C^e + C^{m1} + C^{m2}}{P - (c^{m1*} + c^{m2*}) \cdot (m - 1)} \right\rfloor\right) \tag{3}$$

is the number of regulation periods in Phase 1. Indeed, to sustain maximum contention stall in every regulation period of Phase 1, the task must have both:

1. Enough memory accesses via controller 1, i.e. $K1^* \leq \left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor$.
2. Enough memory accesses via controller 2, i.e. $K1^* \leq \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor$.
3. Enough execution, since when a core is not stalled it must be either computing or accessing memory, i.e. in every Phase 1 period a task must execute for $P - (c^{m1*} + c^{m2*}) \cdot (m - 1)$.

Therefore, $K1^* \leq \left\lfloor \frac{C^e+C^{m1}+C^{m2}}{P-(c^{m1*}+c^{m2*})\cdot(m-1)} \right\rfloor$.
We use the minimum of these 3 values, because this is the largest possible number of periods in Phase 1 and, as argued above, this leads to the worst-case stall.

Remaining stall. Without loss of generality, let $\left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor \geq \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor$, i.e. controller 2 runs out of memory accesses entirely in Phase 2 the latest. (The other case is symmetric.)

To analyse the stall in Phases 2 and 3, if any, we consider the stall of each controller separately. Since memory accesses via controller 2 occur only in Phase 2 (which has at most one regulation period) and not in Phase 3, the contention stall on controller 2 can be upper bounded by $min(\hat{C}^{m2}, RBS2) \cdot (m-1)$, where $\hat{C}^{m2}$ is the number of memory accesses via controller 2 in Phase 2, if any. Observe that these memory accesses and respective stall can be taken into account as computation in the analysis of the stall of memory accesses via controller 1, in Phase 2. Furthermore, in Phase 3, if any, all memory accesses are via controller 1, only. Therefore, we apply Yao's stall analysis to compute the stall of memory accesses via controller 1 in Phases 2 and 3, if they exist.

So, to complete analysis of this case, we compute $\hat{C}^{m2}$, as well as parameters for Yao's single controller stall analysis. Since in the latter we consider the remaining memory accesses via controller 2, $\hat{C}^{m2}$, and respective stall, if any, as computation, C^e is obtained by adding to that value the remaining computation, $\hat{C}^e$, i.e. the task computation that was not performed in Phase 1. Finally, the value of C^m to use in the single controller analysis is the number of memory accesses via controller 1 that were not performed in Phase 1, $\hat{C}^{m1}$, if any. Thus,

$$\begin{aligned} Stall =& Stall1 + min(\hat{C}^{m2}, RBS2) \cdot (m-1) \\ & + single(C^e = \hat{C}^{m2} + min(\hat{C}^{m2}, RBS2) \cdot (m-1) + \hat{C}^e, \\ & C^m = \hat{C}^{m1}, Q = Q1, P = P, m = m) \end{aligned} \tag{4}$$

where $Stall1$ is given by (2). Next, we derive the expressions for $\hat{C}^e, \hat{C}^{m1}$ and $\hat{C}^{m2}$.

In every Phase 1 period a task must execute, i.e. either compute or access memory, when it is not stalled. Thus, in addition to the $c^{m1*} + c^{m2*}$ memory accesses that lead to the maximum stall in a regulation period, a task may have to execute for the remaining time: $P - (c^{m1*} + c^{m2*}) \cdot m$. As we have argued, the total stall will be maximum in executions where computation fills as many of these "holes" as possible. Thus:

$$\hat{C}^e = max\left(0, C^e - K1^* \cdot \left(P - (c^{m1*} + c^{m2*}) \cdot m\right)\right) \tag{5}$$

If there is enough computation to fill all these holes, $C^e \geq K1^* \cdot \left(P - (c^{m1*} + c^{m2*}) \cdot m\right)$, then $\hat{C}^{m1} = C^{m1} - K1^* \cdot c^{m1*}$ and $\hat{C}^{m2} = C^{m2} - K1^* \cdot c^{m2*}$.

If there is not enough computation to fill all these holes, then the remaining holes, $K1^* \cdot (P - (c^{m1*} + c^{m2*}) \cdot m) - C^e$, will be filled with memory accesses. Thus, the total number of memory accesses that will occur in the remaining phases, if any, is:

$$\begin{aligned} \hat{C}^m &= C^{m1} + C^{m2} - K1^* \cdot (c^{m1*} + c^{m2}) - (K1^* \cdot (P - (c^{m1*} + c^{m2*}) \cdot m) - C^e) \\ &= C^{m1} + C^{m2} - (K1^* \cdot (P - (c^{m1*} + c^{m2*}) \cdot (m-1)) - C^e) \end{aligned} \tag{6}$$

To determine $\hat{C}^{m1}$ and $\hat{C}^{m2}$, we distinguish two cases, depending on the value of $K1^*$.

If $K1^* = \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor \left(\leq \left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor\right)$, then an execution that has at least $min(C^{m1} - K1^* \cdot c^{m1*}, RBS1, \hat{C}^m)$ controller 1 memory accesses in the first period of the remaining phases, will suffer maximum stall, because all these memory accesses suffer maximum contention stall. The first bound is the number of memory accesses not used to ensure maximum stall in Phase 1, the second bound is the maximum number of accesses via controller 1 that can suffer maximum stall in a regulation period, and the third bound is the number of memory accesses in the remaining phases. This ensures that controller 2 runs out of memory accesses before controller 1, as shown in Figure 2 iii). Thus the number of memory accesses via controller 2 in Phase 2 is $\hat{C}^{m2} = min\left(\hat{C}^{m} - min(C^{m1} - K1^{*} \cdot c^{m1*}, RBS1, \hat{C}^{m}), C^{m2} - K1^{*} \cdot c^{m2*}\right)$ i.e. the number of memory accesses via controller 2 in Phase 2 is the number of memory accesses not used to fill the holes in Phase 1, discounted by the minimum number of memory accesses via controller 1 that suffer maximum contention in Phase 2, and upper-bounded by the maximum number of controller 2 memory accesses that are not necessary to ensure maximum stall in Phase 1. Finally, $\hat{C}^{m1} = \hat{C}^{m} - \hat{C}^{m2}$.

If $K1^{*} = \left\lfloor \frac{C^{e}+C^{m1}+C^{m2}}{P-(c^{m1*}+c^{m2*})\cdot(m-1)} \right\rfloor$, there is not enough execution to complete the $K1^{*}$+1st regulation period, if any – the execution has at most one regulation period after Phase 1.

In this case, the total stall is maximum in executions where the number of contention stalls in the last period is maximum. However, there cannot be more than $RBS1$ ($RBS2$) contention stalls on controller 1 (2, respectively) in this period. Like in the previous sub-case, an execution with at least $min(C^{m1} - K1^{*} \cdot c^{m1*}, RBS1, \hat{C}^{m})$ controller 1 memory accesses in Phase 2, guarantees that controller 2 runs out of memory accesses no later than controller 1, and suffers maximum stall, because all these memory accesses suffer maximum contention stall. Thus, the expressions we derived for $\hat{C}^{m1}$ and $\hat{C}^{m2}$ in the previous sub-case, are also valid for this one. Summarizing, we get the following expressions:

$$\hat{C}^{m2} = \begin{cases} C^{m2} - K1^{*} \cdot c^{m2*}, & \text{if } C^{e} \geq K1^{*} \cdot \left(P - (c^{m1*} + c^{m2*}) \cdot m\right) \\ min(\hat{C}^{m} - min(C^{m1} - K1^{*} \cdot c^{m1*}, RBS1, \hat{C}^{m}), C^{m2} - K1^{*} \cdot c^{m2*}), & \text{o.w} \end{cases} \tag{7}$$

$$\hat{C}^{m1} = \begin{cases} C^{m1} - K1^{*} \cdot c^{m1*} & \text{if } C^{e} \geq K1^{*} \cdot \left(P - (c^{m1*} + c^{m2*}) \cdot m\right) \\ \hat{C}^{m} - \hat{C}^{m2} & \text{otherwise} \end{cases} \tag{8}$$

5.2.2.2 Sub-case 2: $(P - Q1) + (P - Q2) \geq \frac{P}{m} \cdot (m - 1)$

In this case (by the definition of RBS), $RBS1 + RBS2 \geq \frac{P}{m}$, and therefore it is possible to guarantee maximum contention stall in a period, without any computation or memory accesses without contention. To ensure the maximum stall, the memory accesses should be distributed in a "balanced" way so that both controllers run out of memory access at more or less the same time, thus ensuring that all C^{m} memory access suffers the maximum stall.

Let c^{m1*} and c^{m2*} be the number of memory accesses via controllers 1 and 2 per regulation period that maximize the contention stall in a period. The goal is then to ensure:

$$\frac{C^{m1}}{c^{m1*}} = \frac{C^{m2}}{c^{m2*}} \Rightarrow c^{m2*} = \frac{C^{m2}}{C^{m1}} \cdot c^{m1*} \Rightarrow c^{m2*} = r^{m} c^{m1*}, \text{where: } r^{m} \overset{\text{def}}{=} \frac{C^{m2}}{C^{m1}} \tag{9}$$

Without loss of generality, assume $r^{m} < 1$; the other case is symmetrical. Then it must be:

$$c^{m1*} + c^{m2*} = \frac{P}{m} \Rightarrow (1 + r^{m}) \cdot c^{m1*} = \frac{P}{m} \Rightarrow c^{m1*} = \frac{P}{m \cdot (1 + r^{m})} \tag{10}$$

$$c^{m2*} = r^{m} \cdot c^{m1*} \Rightarrow c^{m2*} = r^{m} \cdot \frac{P}{m \cdot (1 + r^{m})} \tag{11}$$

We now consider three sub-cases:

Sub-case $c^{m1*} \leq RBS1 \wedge c^{m2*} \geq 1$. In this case it is possible to ensure that all memory accesses suffer the maximum contention stall, even without any computation. Thus:

$$Stall = (C^{m1} + C^{m2}) \cdot (m-1) \tag{12}$$

Note that even though c^{m1*} or c^{m2*} may be fractional, these are average values. This means that in an execution with worst-case stall, the number of memory accesses via any controller may not be the same across all the regulation periods. However, there is an execution such that $c^{m1} + c^{m2} = \frac{P}{m}$, in all but possibly the last regulation period, and $c^{m1} \leq RBS1$ and $c^{m2} \leq RBS2$ in every regulation period.

Sub-case $c^{m1*} > RBS1$. In this case, both controllers would run out of computation at the same time only if the number of memory accesses via controller 1 exceeded $RBS1$, and therefore there would be memory accesses without any contention. An execution following the pattern illustrated in Figure 2, with $c^{m1*} = RBS1$ and $c^{m2*} = min(\frac{P}{m} - RBS1, RBS2)$ will have the worst-case stall, and therefore we can apply the analysis in Section 5.2.2.1.

Sub-case $c^{m2*} < 1$. In this case, both controllers would run out of computation at the same time only if there are some periods without memory accesses via controller 2. An execution following the pattern illustrated in Figure 2, with $c^{m2*} = 1$ and $c^{m1*} = min(\frac{P}{m} - 1, RBS1)$ will have the worst-case stall, and therefore we can apply the analysis in Section 5.2.2.1.

5.2.3 Case 3: $(b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}) \vee (b1 > \frac{1}{m} \wedge b2 \leq \frac{1}{m})$

In this case, executions with the maximum number of regulation stalls do not always lead to the worst-case stall. This is shown in Figure 3. In execution i), all memory accesses via controller 1 are clustered, causing two regulation stalls on controller 1, in the first two regulation periods. All the memory accesses via controller 2, occur in the third regulation period. Of these, only the first two suffer the maximum contention stall. The remainder suffer no contention, because the memory budget of the remaining cores, $P - Qi$, is exhausted by the stalls of the first 2 memory accesses. In execution ii), there is one memory access via controller 1 in each period, and thus there is no regulation stall on controller 1, but each of these accesses suffers the maximum contention stall. Furthermore, in each of the first 3 periods, there are 2 memory accesses via controller 2, each of which suffers the maximum contention stall. Thus all memory accesses via both controllers suffer the maximum contention stall, and the total stall for execution ii) exceeds that of execution i). This is counter-intuitive, because the contention stall by accesses via controller 1 in execution ii), 12, is smaller than the regulation stall, 20, caused by the same number of accesses via controller 1 in execution i). However, this loss is more than compensated by the contention stall in execution ii) of the 4 memory accesses via controller 2 that suffer no contention stall in execution i). I.e., although we are trading off a regulation stall, $P - Qi$, for contention stalls, presumably with maximum contention stall, $Qi \cdot (m-1) < P - Qi$, we may also be adding stall to memory accesses via the second controller that previously suffered no stall.

Depending on whether $b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}$ or $b1 > \frac{1}{m} \wedge b2 \leq \frac{1}{m}$, there are two sub-cases. Because they are symmetrical, we analyse only the former.

5.2.3.1 Sub-case 3.1: $b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}$

Figure 3 shows that the maximum number of regulation stalls does not always lead to the worst-case stall. Furthermore, it can be shown that the total stall is maximum if there are no memory accesses via the second controller in periods with a regulation stall. Thus, the

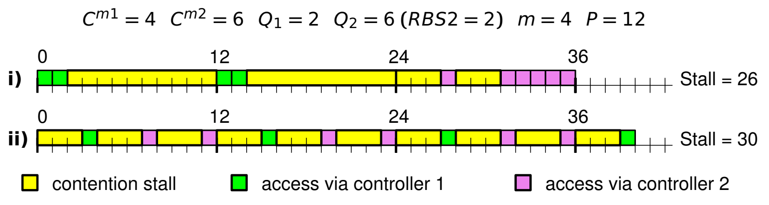


Figure 3 Maximizing the number of regulation stalls may not lead to the worst-case stall.

Algorithm 1 Compute stall for each task.

```
Input: Parameters: $C^{m1}$, $C^{m2}$, $m$, $C^e$, $Q1$, $Q2$ and $P$ (omitting task's index for simplicity)
Output: Stall
 1: $b1 = \frac{Q1}{P}$, $b2 = \frac{Q2}{P}$, $RBS1 = \frac{P-Q1}{m-1}$, $RBS2 = \frac{P-Q2}{m-1}$ and $C = C^e + C^{m1} + C^{m2}$
 2: if ($b1 \le \frac{1}{m} \;\wedge\; b2 \le \frac{1}{m}$) then        ▷ Regulation stall is dominant for both controllers
 3:     Stall = Equation (1)
 4: else if ($b1 > \frac{1}{m} \;\wedge\; b2 > \frac{1}{m}$) then        ▷ Contention stall is dominant for both controllers
 5:     if ($(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$) then
 6:         $c^{m1*} = RBS1$, $c^{m2*} = RBS2$
 7:         Compute Stall with Algorithm 2
 8:     else        ▷ $(P - Q1) + (P - Q2) \ge \frac{P}{m} \cdot (m - 1)$
 9:         $r^m = \frac{C^{m2}}{C^{m1}}$, $c^{m1*}$ = Equation 10, $c^{m2*}$ = Equation 11
10:         if ($r^m < 1$) then
11:             if ($c^{m1*} \le RBS1 \wedge c^{m2*} \ge 1$) then
12:                 Stall = Equation 12
13:             else if ($c^{m1*} > RBS1$) then
14:                 $c^{m1*} = RBS1$, $c^{m2*} = min(RBS2, \frac{P}{m} - RBS1)$
15:                 Compute Stall with Algorithm 2
16:             else        ▷ $c^{m2*} < 1$
17:                 $c^{m1*} = min(RBS1, \frac{P}{m} - 1)$ $c^{m2*} = 1$
18:                 Compute Stall with Algorithm 2
19:             end if
20:         else        ▷ $r^m \ge 1$: symmetric of previous case, swap indices
21:         end if
22:     end if
23: else        ▷ Regulation stall is dominant for only one controller
24:     if ($b1 \le \frac{1}{m} \;\wedge\; b2 > \frac{1}{m}$ ) then
25:         Compute $\Delta\rho^*$ using Algorithm 3
26:         Stall = Equation 13
27:     else        ▷ $b2 \le \frac{1}{m} \;\wedge\; b1 > \frac{1}{m}$: symmetric of previous case
28:     end if
29: end if
30: return Stall += ($P - \min(Q1, Q2)$)        ▷ This adds the stall when the task arrives.
```

Algorithm 2 Compute stall for contention dominant case.

```
Input: Parameters: $c^{m1*}$, $c^{m2*}$, $C^{m1}$, $C^{m2}$, $m$, $C^e$, $Q1$, $Q2$ and $P$ (omitting task's index)
Output: Stall
 1: $b1 = \frac{Q1}{P}$, $b2 = \frac{Q2}{P}$, $RBS1 = \frac{P-Q1}{m-1}$, $RBS2 = \frac{P-Q2}{m-1}$ and $C = C^e + C^{m1} + C^{m2}$
 2: $K1^*$ = Equation 3 ,
 3: Stall1 = Equation 2
 4: $\hat{C}^e$ = Equation 5, $\hat{C}^{m1}$ = Equation 8, $\hat{C}^{m2}$ = Equation 7
 5: Stall23 = $single(C^e = \hat{C}^{m2} \cdot m + \hat{C}^e, C^m = \hat{C}^{m1}, Q = Q1, P = P, m = m)$
 6: return Stall = Stall1 + $min(\hat{C}^{m2}, RBS2) \cdot (m - 1)$ + Stall23        ▷ Equation 41
```

following memory access pattern with two phases leads to the worst-case stall: in the first phase, there is a number, possibly 0, of consecutive periods with regulation stalls; in the second phase, the contention-only phase, there is a number of consecutive periods, possibly only 1, with contention stalls only. Thus, the problem of finding the worst-case stall reduces to that of determining the number of regulation stalls that maximizes that stall. Actually, to simplify the mathematical expressions, we use the difference, $\Delta\rho^*$, between this number and the maximum number of regulation stalls, $\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor$. The total stall can then be determined using Yao's stall analysis:

$$\begin{aligned} &Stall = single(Q = Q1, C^m = C^{m1} - C^{m1} \bmod Q1 - \Delta\rho^* Q1, C^e = 0) \\ &+ ((C^{m1} \bmod Q1) + \Delta\rho^* \cdot Q1) \cdot (m-1) \\ &\qquad + single(Q = Q2, C^m = C^{m2}, C^e = C^e + ((C^{m1} \bmod Q1) + \Delta\rho^* \cdot Q1) \cdot m) \end{aligned} \tag{13}$$

where, for computing the stall on the memory accesses via controller 2 in the second phase, we account the memory accesses via controller 1 in the second phase and respective contention stalls as computation, assuming that each of them suffers the maximum contention stall under round-robin, $m-1$. Algorithms 1 and 2 detail the case analysis that we have described so far in this section. In the following, we determine the value of $\Delta\rho^*$.

We consider two main sub-cases depending on whether there is enough computation, including residual memory accesses via controller 1, to ensure that every memory access via controller 2 suffers maximum contention.

5.2.3.2 Sub-case 1: Enough computation

If $C^e \geq \left\lfloor \frac{C^{m2}}{RBS2} \right\rfloor \cdot (P - m \cdot RBS2) - (C^{m1} \bmod Q1) \cdot m$, then every memory access in the contention-only phase suffers maximum contention, and therefore the total stall is maximum when the number of regulation stalls is maximum, i.e. $\Delta\rho^* = 0$.

5.2.3.3 Sub-case 2: Not enough computation

In this case, as illustrated in Figure 3, if there are memory accesses in the contention-only phase that suffer no contention, the worst-case stall may occur when the number of regulation stalls is not maximum.

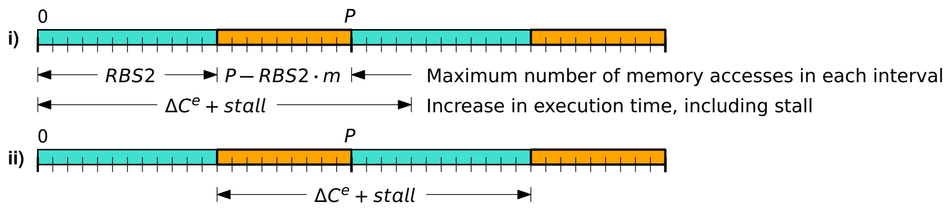
When the number of regulation stalls is decremented by one, the regulation stall reduction by $P - Q1$ is partially compensated by an increase of the contention stall via controller 1 by $Q1 \cdot (m-1)$. If the increase in contention stall via controller 2, $\Delta stall_c^2$ is such that:

$$\Delta stall_c^2 > \Delta stall_c^{2^*} \stackrel{\text{def}}{=} P - Q_1 - Q_1 \cdot (m-1) = P - Q_1 \cdot m \tag{14}$$

then reducing the number of regulation stall leads to a larger total stall. In other words, the total stall will be worse if the increase in the number of memory accesses with maximum stall, ΔC_c^{m2}, satisfies the following inequality:

$$\Delta C_c^{m2} > \Delta C_c^{m2^*} \stackrel{\text{def}}{=} \frac{\Delta stall_c^{2^*}}{m-1} = \frac{P - Q_1 \cdot m}{m-1} \tag{15}$$

Like in the analysis in Section 5.2.2, to compute the stall on memory accesses via controller 2, we can view the memory accesses via controller 1 and respective contention stall as computation. Thus, we need to determine ΔC_c^{m2} when the computation in the contention-only phase increases by $\Delta C^e = Q_1 \cdot m$. The challenge is that this value, ΔC_c^{m2}, may not be constant. I.e., when we increase the computation by $\Delta C^e = Q_1 \cdot m$, ΔC_c^{m2} may have different values depending on other parameter values.



0
P
i)
RBS2
P − RBS2 · m
Maximum number of memory accesses in each interval
ΔC^e + stall
Increase in execution time, including stall
0
P
ii)
ΔC^e + stall

Figure 4 Upper (i) and lower (ii) bounds on ΔC_c^{m2}.

Our solution is to compute the maximum and minimum values of ΔC_c^{m2}, $\Delta C_c^{m2}(max)$ and $\Delta C_c^{m2}(min)$, respectively, and then finding $\Delta\rho^*$ by case analysis, as described below.

When we increase the computation of the contention-only phase by ΔC^e, the total execution of that phase, including any contention, will increase at least by that much. This execution can replace memory accesses via controller 2 that did not have any contention, i.e memory accesses in excess of $RBS2$ accesses per period, which can then be shifted towards the end of the execution. ΔC_c^{m2} will be maximum if the shifted memory accesses are added to a regulation period with no memory accesses via controller 2, up to a limit of $RBS2$ memory accesses per regulation period, as shown in Figure 4 i). Thus, in this case, as a result of adding ΔC^e memory accesses we get:

$$\begin{aligned}\Delta C_c^{m2}(max) = RBS2 \cdot &\left\lfloor \frac{\Delta C^e}{RBS2 + P - RBS2 \cdot m} \right\rfloor \\ &+ min(RBS2, \Delta C^e \bmod (RBS2 + P - RBS2 \cdot m)) \\ &= RBS2 \cdot \left\lfloor \frac{\Delta C^e}{Q2} \right\rfloor + min(RBS2, \Delta C^e \bmod Q2) \end{aligned} \quad (16)$$

The first term corresponds to the number of additional periods with $RBS2$ memory accesses. (Note that ΔC^e is used both to shift memory accesses via controller 2, and to fill the "hole" in the remaining of the period, $P - RBS2 \cdot m$.) The second term corresponds to the number of memory accesses in the last incomplete regulation period, if any: essentially, the memory accesses that can be replaced with the remaining of ΔC^e that was not used for the additional full periods, upper-bounded by $RBS2$.

On the other hand, ΔC_c^{m2} will be minimum, if, before adding ΔC^e, the execution ended immediately after the $RBS2$ accesses with contention. This is shown in Figure 4 ii). In this case, the analysis is similar to the one above, and therefore we can also use (16), except that rather than using ΔC^e, we need to use $max(\Delta C^e - (P - RBS2 \cdot m), 0)$, because the remainder of the period at which the execution ended needs to be filled with "computation" before an earlier memory access via controller 2 without contention stall can experience the maximum contention stall by shifting it towards the end of the execution.

We can now distinguish there sub-cases, depending on the relative values of ΔC_c^{m2*}, $\Delta C_c^{m2}(max)$ and $\Delta C_c^{m2}(min)$.

Sub-case $\Delta C_c^{m2*} \geq C_c^{m2}(max)$. In this case, the increase in the number of memory accesses with contention cannot make up for the eliminated regulation stall, so $\Delta\rho^* = 0$.

Sub-case $\Delta C_c^{m2*} < C_c^{m2}(min)$. In this case, the increase in the number of memory accesses with contention suffices to make up for the eliminated regulation stall. Therefore, the worst-case stall increases as we reduce the number of regulation stalls until one of the following 3 cases occurs: 1) there are no more regulation stalls; 2) there are not enough memory

Algorithm 3 Compute $\Delta\rho^*$.

```
Input: Parameters: $C^{m1}$, $C^{m2}$, $m$, $C^e$, $Q1$, $Q2$ and $P$ (omitting task index for simplicity)
Output: $\Delta\rho^*$
 1: $RBS1 = \frac{P-Q1}{m-1}$, $RBS2 = \frac{P-Q2}{m-1}$ and $C = C^e + C^{m1} + C^{m2}$
 2: $\Delta C^e = m \cdot Q1$
 3: $\Delta C_c^{m2}(max)$ = Equation 16
 4: $\Delta C_c^{m2}(min)$ = Equation 16, but replacing $\Delta C^e$ with $\max(\Delta C^e - (P - m \cdot RBS2), 0)$
 5: $\Delta C_c^{m2*} = \left\lfloor \frac{P-m\cdot Q1}{m-1} \right\rfloor$
 6: if $(C^e \geq \left\lfloor \frac{C^{m2}}{RBS2} \right\rfloor \cdot (P - m \cdot RBS2) - (C^{m1} \bmod Q1) \cdot m)$ then
 7:     $\Delta\rho^* = 0$                                        ▷ There is enough "computation"
 8: else if ( $\Delta C_c^{m2}(max) \leq \Delta C_c^{m2*}$ ) then            ▷ Which implies $\Delta C_c^{m2}(min) \leq \Delta C_c^{m2*}$
 9:     $\Delta\rho^* = 0$                                ▷ Maximize regulation stalls on Controller one
10: else if ( $\Delta C_c^{m2}(min) > \Delta C_c^{m2*}$ ) then            ▷ Which implies $\Delta C_c^{m2}(max) > \Delta C_c^{m2*}$
11:     $\Delta\rho^* = 0$
12:     $stall = single(Q = Q2, C^m = C^{m2}, C^e = C^e + (C^{m1} \bmod Q1) \cdot m)$
13:     $R = C^{m2} + C^e + (C^{m1} \bmod Q1) \cdot m + stall$
14:     $C_{\bar{c}}^{m2} = C^{m2} - \left\lfloor \frac{R}{P} \right\rfloor \cdot RBS2 - min\left(\left\lfloor \frac{R \bmod P}{m} \right\rfloor, RBS2\right)$
15:     while $\left( C_{\bar{c}}^{m2} > \Delta C_c^{m2*} \wedge \Delta\rho^* < \left\lfloor \frac{C^{m1}}{Q1} \right\rfloor \right)$ do
16:         $\Delta\rho_t^* = \Delta\rho^* + 1$
17:         $\hat{C}^{m1} = C^{m1} \bmod Q1 + \Delta\rho_t^* \cdot Q1$           ▷ Accesses via controller 1 in second phase
18:         $stall = single(Q = Q2, C^m = C^{m2}, C^e = C^e + \hat{C}^{m1} \cdot m)$
19:         $R = C^{m2} + C^e + \hat{C}^{m1} \cdot m + stall$
20:         $C_{\bar{c}t}^{m2} = max\left(C^{m2} - \left\lfloor \frac{R}{P} \right\rfloor \cdot RBS2 - min\left(\left\lfloor \frac{R \bmod P}{m} \right\rfloor, RBS2\right), 0\right)$
21:         if $\left(\hat{C}^{m1} - min\left(Q1 - 1, max\left(0, \left\lfloor \frac{R \bmod P}{m} \right\rfloor - RBS2\right)\right) \leq (Q1 - 1) \cdot \frac{R}{P}\right)$ then ▷ Enough
    reg. periods to ensure that there is no reg. stall in periods with accesses via both controllers.
22:             $\Delta\rho^* = \Delta\rho_t^*, C_{\bar{c}}^{m2} = C_{\bar{c}t}^{m2}$
23:         else break
24:         end if
25:     end while
26: else                                                  ▷ $\Delta n_c^2(min) \leq \Delta n_c^{2*} < \Delta n_c^2(max)$
27:     $\Delta\rho(max) = 0, stall(max) = 0$                              ▷ Variables for maximum stall
28:     for $\Delta\rho^* = 0$ to $\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor$ do                                  ▷ Do exhaustive search
29:         $\hat{C}^{m1} = C^{m1} \bmod Q1 + \Delta\rho_t^* \cdot Q1$
30:         $stall = single(Q = Q2, C^m = C^{m2}, C^e = C^e + \hat{C}^{m1} \cdot m)$ ▷ Cont. stall on both controllers
31:         $R = C^{m2} + C^e + \hat{C}^{m1} \cdot m + stall$                ▷ Duration of contention-only phase
32:         if $stall + \left(\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor - \Delta\rho^*\right) \cdot (P - Q1) > \text{stall(max)}$
    $\wedge \left(\hat{C}^{m1} - min\left(Q1 - 1, max\left(0, \left\lfloor \frac{R \bmod P}{m} \right\rfloor - RBS2\right)\right) \leq (Q1 - 1) \cdot \left\lfloor \frac{R}{P} \right\rfloor\right)$ then
33:             $stall(max) = stall + \left(\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor - \Delta\rho^*\right) \cdot (P - Q1)$
34:             $\Delta\rho^*(max) = \Delta\rho^*$
35:         end if
36:     end for
37:     $\Delta\rho^* = \Delta\rho^*(max)$
38: end if
39: return $\Delta\rho^*$
```

accesses via controller 2, ΔC_c^{m2*}, without the maximum contention stall, to compensate for the loss in the regulation stall; or 3) the number of memory accesses via controller 1 in at least one period of the second phase exceeds $Q1 - 1$, in which case we would have a regulation stall, and therefore there would be no reduction in the number of regulation stalls.

Algorithm 4 Sensitivity analysis to reclaim memory bandwidth from both controllers.

```
Input: $b1$, $b2$, $m$, $\Delta$ (threshold to stop the algorithm)) and $\tau$
Output: Minimum required memory bandwidth of both controllers
 1: $b^1_{min} = 0$, $b^1_{max} = b1$, $b^2_{min} = 0$, $b^2_{max} = b2$
 2: while $(b^1_{max} - b^1_{min} > \Delta \vee b^2_{max} - b^2_{min} > \Delta)$ do
 3:     for each controller $j \in \{1, 2\}$ do
 4:         if $(b^j_{max} - b^j_{min} > \Delta)$ then
 5:             $X^j = \lfloor \frac{b^j_{min} + b^j_{max}}{2} \rfloor$
 6:             if $(j == 1)$ then
 7:                 Schedulability = MultiControllerSchedulabilityAnalysis($X^j$, $b^2_{max}$, $m, \tau$)
 8:             else
 9:                 Schedulability = MultiControllerSchedulabilityAnalysis($b^1_{max}$, $X^j$, $m, \tau$)
10:             end if
11:             if (Schedulability == true) then $b^j_{max} = X^j$
12:             else $b^j_{min} = X^j$
13:             end if
14:         end if
15:     end for
16: end while
17: return $\{b^1_{max}$ and $b^2_{max}\}$
```

Because ΔC^{m2} varies, we do not know a closed form expression for the number of regulation periods to reduce. Thus, we use the iterative procedure shown in Algorithm 3. We hence start with $\Delta\rho^* = 0$ and keep increasing it by one until one of the above 3 stop conditions is satisfied. Specifically, while there are still enough memory accesses via controller 2 without maximum contention stall, $C^{m2}_{\bar{c}}$, and there is still one regulation stall (line 15), $\Delta\rho^*$ is tentatively increased by one. In each iteration, we tentatively compute the total stall using Yao's analysis with the appropriate parameters (line 18) and the number of memory accesses via controller 2 that suffer no contention (line 20), for the tentative value of $\Delta\rho^*$. If the number of memory accesses via controller 1 in all periods of the contention-only phase (line 21) does not exceed $Q1 - 1$, then the tentative values become definitive (line 22), and the algorithm loops again, otherwise it exits the loop and terminates.

All other cases, i.e. $C^{m2}_c(min) \leq \Delta C^{m2*}_c < C^{m2}_c(max)$. In this case, the total stall sometimes increases when the number of regulations stalls decreases by one and sometimes it does not. Thus in this case, our approach to find the value of $\Delta\rho^*$ is to compute the stall for every possible value of $\Delta\rho^*$ and pick the one that leads to the maximum stall. Algorithm 3, lines 27-37, details the computation of $\Delta\rho^*$ in this case.

5.3 Schedulability analysis

Until now, we assumed one task per core. When many tasks are assigned to a core, the task in consideration and those of higher priority can be modelled by one synthetic task, using the approach in [20], and schedulability analysis can be performed as summarized in Section 4.

6 Bandwidth Allocation and Task-to-core Assignment Heuristics

We propose 5 heuristics for allocating tasks and memory bandwidth of both controllers to the cores. They are evaluated in terms of system schedulability. We use Audsley's algorithm [1] to assign task priorities, even if it is no longer necessarily optimal in the presence of stalls.

Even. The total memory bandwidth of both controllers is equally distributed among all cores. Subject to this even share, the task-to-core assignment is performed using first-fit.

Uneven. Initially, this heuristic also distributes both controller's bandwidth evenly among cores and employs the first-fit for task-to-core assignment. However, instead of declaring failure whenever a task does not fit on any core, it sets that task aside, and moves on to consider the next task. Any tasks that remain unassigned after considering all tasks, are handled in-order as follows. Each core's memory bandwidth from both controllers is "trimmed" to the minimum value that preserves schedulability, via the sensitivity analysis of Algorithm 4, explained later in this section. Let the total reclaimed bandwidth from all cores be $B1$ and $B2$ from controllers 1 and 2, respectively. A second round of first-fit tries to assign the remaining tasks, assuming that the bandwidth of the target core i is increased by $B1$ and $B2$ for controllers 1 and 2, respectively. Upon successfully assigning such a task, we trim anew the target cores's memory budgets via sensitivity analysis, adjust the available reclaimed budgets and move on to the next task in a similar manner.

Greedy-fit. Initially, the total memory bandwidth of both controllers is assigned to the first core and the task-set is iterated over once (in a given order) to assign the maximum number of tasks to this core; if a task does not fit, we try the next one. Afterwards, the spare bandwidth from each controllers on this core is reclaimed via sensitivity analysis, and is fully assigned to next core. And so on, until all tasks are assigned or the cores run out.

Humble-fit. Similar to greedy-fit, except that when a task assignment fails, we move to the next core (attempting no more task assignments on the current one).

Memory-fit. Initially, $b1_i = b2_i = 0$, for every core i, where bx_i is the allocated memory bandwidth of controller x on core i. Each task is assigned to the core i that requires the least increase to $b1_i + b2_i$ to accommodate it, subject to existing task assignments.

"Uneven" explores a larger solution space than "Even. "Greedy-fit" and "Humble-fit" aggressively optimise for processing capacity use foremost. Conversely, "Memory-fit" optimises for bandwidth instead. Hence, all heuristics sample the solution space in different ways.

Sensitivity analysis. Algorithm 4 presents the sensitivity analysis that trims the unused memory bandwidth from both controllers and outputs the least required memory bandwidth from each controller. This sensitivity analysis, used for bandwidth optimisation, is an adaptation of binary interval search ([19, 2]). It gives both controllers an equal chance to preserve their bandwidth in a round-robin fashion. By comparison, completely optimizing one controller followed by the second one, may lead to an imbalanced approach, hence avoided.

7 Evaluation

Experimental Setup. We developed a Java tool for our experiments. Its first module generates the synthetic task sets and sets up a platform with the given input parameters. A second module performs task-to-core allocation and feasibility analysis with two controllers.

We generate the task-set with a given target $U = x \cdot m : x \in (0, 1]$ using UUnifast-discard algorithm [6, 9] for unbiased distribution of task utilisations. The task-set size is given as input. Task periods are log-uniform-distributed, in the range 10-100 ms. We assume implicit deadlines, even if our analysis also holds for constrained deadlines. The WCET of a task is derived as $C_i = U_i \cdot T_i$. The total memory accesses of each task are randomly selected in the range $[0, \Gamma \cdot C_i]$, with memory intensity factor $\Gamma \in (0, 1]$ user-defined. The total memory

Table 2 Overview of Parameters.

Parameters	Values	Default
Number of cores (m)	$\{4, 8, 12, 16\}$	4
Task-set size (n)	$\{8, 16, 24, 32, 40, 48\}$	16
Regulation period (P)	$\{1us, 10us, 100us, 1ms\}$	$100us$
Inter-arrival time (T_i)	$10ms$ to $100ms$	N/A
Nominal utilisation ($\bar{U} = \frac{U}{m}$)	$\{0.1 : 0.01 : 1\}$	N/A
Memory intensity (Γ)	$\{0.1 : 0.1 : 1\}$	0.5

accesses are randomly divided between the two memory controllers. By default the task-set is sorted in descending order of utilisation. For each set of input parameters, we generate 1000 task-sets. We use independent pseudo-random number generators for the utilisations, minimum inter-arrival times/deadlines, memory accesses and reuse their seeds [12]. Table 2 summarises all parameters, with default values underlined. We observed that size of the regulation period has no effect on the schedulability ratio.

To avoid having hundreds of plots, in each experiment we vary only one parameter, with others conforming to the defaults from Table 2 and present the results as plots of *weighted schedulability*. This performance metric, adopted from [4], condenses what would have been three-dimensional plots into two dimensions. It is a weighted average that gives more weight to task-sets with higher utilisation, which are supposedly harder to schedule. Specifically, using notation from [7], let $S_y(\tau, p)$ represent the result (0 or 1) of the schedulability test y for a given task-set τ with an input parameter p. Then $W_y(p)$, the weighted schedulability for that test y as a function p, is $W_y(p) = \sum_{\forall\tau} \left(\bar{U}(\tau) \cdot S_y(\tau, p)\right) / \sum_{\forall\tau} \bar{U}(\tau)$. Here, $\bar{U}(\tau) \overset{\text{def}}{=} U(\tau)/m$ is the system utilisation, normalised by the number of cores m.

No other stall analysis with two controllers exists in the literature to compare with. We therefore compare our approach against a system where the two controllers are partitioned among cores that can only make requests to their assigned controller. The benefit of such partitioning is that it roughly cuts contention in half. On the other hand, tasks assigned to one controller cannot access data addressable by the other controller.

For the comparison, half the cores are assigned to each controller. Since each core accesses only one controller, the feasibility of the tasks assigned to it can be tested with Yao's analysis [20]. We adapt the task-to-core assignment heuristics and bandwidth allocation schemes presented in Section 6 for the partitioned case: The even heuristic equally divides a controller's bandwidth among its associated cores. Similarly, in the uneven heuristic, the readjustment of the controllers bandwidth is performed only among the controller's associated cores. In the greedy-fit/humble-fit, all bandwidth of a given controller is only assigned to its first associated core with an objective to maximise the number of tasks assigned to it. The trimmed-off bandwidth from this controller is assigned to its remaining associated cores. If the task is not feasible on the cores associated to the first controller, its feasibility is next checked on the set of cores associated with the second controller. In the memory-fit, a task is assigned to the core with the lowest bandwidth requirement of its controller. We use Yao- and MC- prefixes to denote the partitioned and our approach, respectively, followed by the name of the heuristic (even, uneven, greedy-fit, humble-fit and memory-fit).

Results. Figure 5 presents the weighted schedulability for different number of cores for both systems with partitioned and shared controllers (our approach) using the proposed heuristics. The first important result is that all heuristics under partitioning perform better

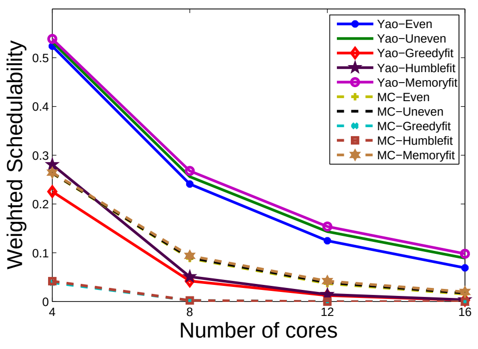


Figure 5

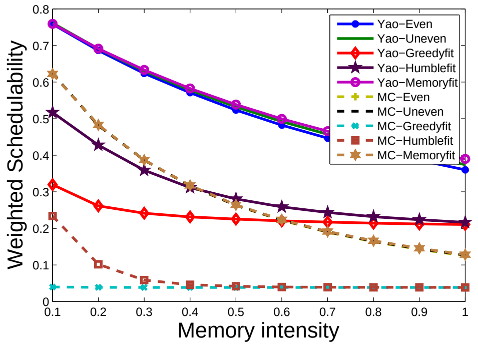


Figure 6

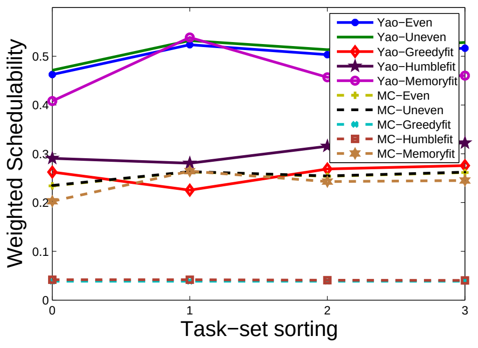


Figure 7

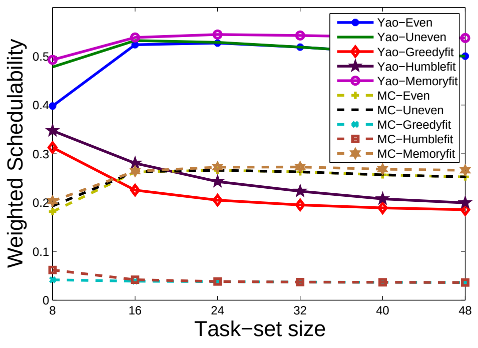


Figure 8

than their corresponding heuristic under shared controllers, which is due to the stall being roughly cut in half in the former approach. This difference ranges around 10% − 30% in absolute terms of weighted schedulability. Of course, this expected result applies only when there are no dependencies across partitions. However, in many systems, there is always some sharing/communication of data among tasks and this might make such partitioning impossible. In other cases, a single controller cannot deliver enough bandwidth. This may become more frequent in the future, as applications getting more demanding. Therefore safe analysis for predictable access to both controllers, like the one proposed here, is needed.

In terms of heuristics, memory-fit, uneven, even, humble-fit and greedy-fit is the descending ordered list w.r.t. weighted schedulability ratio. The memory-fit heuristic, which optimises the use of memory bandwidth, performing best, implies that memory bandwidth is typically the scarce resource for the given set of parameters. The uneven and even heuristics are more balanced in terms of bandwidth and processing capacity distribution and hence, perform close to memory-fit. Humble-fit and greedy-fit are too aggressive in construction to optimise the use of processing capacity at the cost of memory resources and hence underperform the other heuristics in a memory-scarce setup. Greedy-fit manages the memory resources comparatively better than humble-fit and hence, outperforms it. Yet, if the applications are compute-intensive and the system is not scarce w.r.t. memory resource, the heuristics that optimise for processing resources may become handy and outperform their counterparts.

With more cores, the contention from other cores increases and hence, the schedulability of the system decreases. Figure 6 presents the effect of memory intensity over the proposed heuristics. Obviously, higher memory intensity increases the contention on the shared controllers, consequently decreasing the schedulability. We also compared the effect of the task indexing over the different heuristics as shown in Figure 7. The numbers 0, 1, 2 and 3 on the X-axis correspond to task-set ordering w.r.t. descending order of deadlines, utilisation, total memory requests and memory density (i.e. total memory requests divided by the T_i), respectively. Task-set indexing w.r.t. utilisation benefits the memory-fit, even and uneven heuristics. Figure 8 shows that task-set size has very limited effect on the memory-fit, uneven and even approaches and they scale well when that increases. Conversely, the performance of humble-fit and greedy-fit degrade with greater task-set sizes due to their aggressive optimisation of processor usage at the expense of memory bandwidth.

8 Conclusion

This paper demonstrated that worst-case memory stall analyses for single-memory-controller multicores with memory regulation are unsafe if applied to multicores with multiple memory controllers. We overcome this limitation by proposing a new memory stall analysis for multicore platforms with two memory controllers that captures the cases where all cores can access both controllers. We also proposed five memory allocation heuristics, each specialising in optimising processing capacity and/or memory bandwidth. The experimentally quantified cost of allowing all cores to flexibly access the memory space of two controllers is $10 - 30\%$ in terms of weighted schedulability. Results further show that the proposed memory-fit heuristic performs well if bandwidth is scarce. The even and uneven heuristics are suitable for balanced systems, while greedy-fit and humble-fit are handy for compute-intensive systems.

References

1 N. C. Audsley. On priority asignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

2 M. A. Awan, K. Bletsas, P. F. Souto, and E. Tovar. Semi-partitioned mixed-criticality scheduling. In *Proceedings of the 30th International Conference on the Architecture of Computing Systems (ARCS 2017)*, pages 205–218, 2017. doi:10.1007/978-3-319-54999-6_16.

3 Muhammad Ali Awan, Pedro Souto, Konstantinos Bletsas, Benny Akesson, and Eduardo Tovar. Mixed-criticality scheduling with memory bandwidth regulation. In *Proceedings of the 55th IEEE/ACM Conference on Design Automation and Test in Europe (DATE 2018)*, March 2018.

4 A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of the OSPERT*, pages 33–44, 2010.

5 M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory-bus contention. *ACM SIGBED Review*, 10(3):35–42, 2013. doi:10.1145/2544350.2544354.

6 E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Journal of Real–Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.

7 A. Burns and R. I. Davis. Adaptive mixed criticality scheduling with deferred preemption. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS 2014)*, pages 21–30, Dec 2014. doi:10.1109/RTSS.2014.12.

8 D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, pages 39–48, June 2013.

9 R. I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 398–409, Dec 2009. doi:10.1109/RTSS.2009.31.

10 J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 151–159, June 2014. `doi:10.1109/SIES.2014.6871199`.

11 Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjodin. Multi-core composability in the face of memory-bus contention. In *Proceedings of the 20th IEEE Real-Time Technology and Applications Symposium (RTAS 2014)*, 2014.

12 Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

13 R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 174–183, July 2015. `doi:10.1109/ECRTS.2015.23`.

14 Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

15 J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, pages 109–118, 2014. `doi:10.1109/ECRTS.2014.20`.

16 NXP. QorIQ Layerscape Processors Based on Arm Technology, 2018. URL: `www.nxp.com/products/processors-and-microcontrollers/applications-processors/qoriq-platforms/p-series`.

17 R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, pages 97–108, April 2016. `doi:10.1109/RTAS.2016.7461339`.

18 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, Bradford Richard, et al. Single core equivalent virtual machines for hard real—time computing on multicore processors. Technical report, Univ. of Illinois at Urbana Champaign, 2014.

19 Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Åkesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Journal of Real–Time Systems*, 50(5-6):680–735, 2014.

20 G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *IEEE Transactions on Computers*, 65(2):601–614, Feb 2016.

21 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 299–308, July 2012. `doi:10.1109/ECRTS.2012.32`.

22 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, pages 55–64, April 2013. `doi:10.1109/RTAS.2013.6531079`.

HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET Estimates in Multicore Real-Time Systems

Pedro Benedicte[1]
Barcelona Supercomputing Center and Universitat Politècnica de Catalunya, Barcelona, Spain
pbenedic@bsc.es
https://orcid.org/0000-0003-1670-7783

Carles Hernandez[2]
Barcelona Supercomputing Center, Barcelona, Spain
carles.hernandez@bsc.es
https://orcid.org/0000-0001-5393-3195

Jaume Abella[3]
Barcelona Supercomputing Center, Barcelona, Spain
jaume.abella@bsc.es
https://orcid.org/0000-0001-7951-4028

Francisco J. Cazorla[4]
Barcelona Supercomputing Center and IIIA-CSIC, Barcelona, Spain
francisco.cazorla@bsc.es
https://orcid.org/0000-0002-3344-376X

Abstract

High-performance processors have deployed multilevel cache (MLC) systems for decades. In the embedded real-time market, the use of MLC is also on the rise, with processors for future systems in space, railway, avionics and automotive already featuring two or more cache levels. One of the most critical elements for MLC is the write policy that not only affects several key metrics such as performance, WCET estimates, energy/power, and reliability, but also the design of complexity-prone cache coherence protocol and cache reliability solutions. In this paper we make an extensive analysis of existing write policies, namely write-through (WT) and write-back (WB). In the context of the real-time domain, we show that no write policy is superior for all metrics: WT simplifies the design of the coherence and reliability solutions at the cost of performance, WCET, and energy; while WB improves performance and energy results, but complicates cache design. To take the best of each policy, we propose Hybrid Write Policy (HWP) a low-complexity hardware mechanism that reconciles the benefits of WT in terms of simplifying the cache design (e.g. coherence solution) and the benefits of WB in improved average performance and WCET estimates as the pressure on the interconnection network increases. Guaranteed performance results show that HWP scales with core count similar to WB. Likewise, HWP reduces cache energy usage of WT, to levels similar to those of WB. These benefits are obtained while retaining the reduced coherence complexity of WT, in contrast to high coherence costs under WB.

2012 ACM Subject Classification Computer systems organization → Parallel architectures, Computer systems organization → Embedded systems, Computer systems organization → Real-time systems, Computer systems organization → Dependable and fault-tolerant systems and networks

[1] Spanish Ministry of Education, Culture and Sports under the FPU grant FPU15/01394
[2] MINECO and FEDER funds through grant TIN2014-60404-JIN
[3] MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717
[4] European Research Council under EU's H2020 research/innovation programme (grant No. 772773)

Keywords and phrases multilevel caches, real-time systems, multicores, WCET

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.3

Acknowledgements This work was partially funded by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P and the HiPEAC Network of Excellence. It also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773).

1 Introduction

High-performance processors ubiquitously deploy several levels of cache (e.g. IBM POWER 9, Intel Core i7-based systems, and the ARM A Series). This emanates from the positive impact that MLC have on overall system performance. However, MLC design is delicate, not only because it involves high complexity when dealing with coherence, inclusion, and miss policies (among other issues); but also because it can affect key metrics like cycle time, energy/power (and hence temperature), and reliability.

In the real-time domain, the increase in computation needs of critical software across all domains, is driving system designers towards the use of multicores, which necessarily carry the use of MLC systems. For instance, the ARM big.LITTLE (automotive), the Aeroflex Gaisler LEON4 (space), and the NXP T2080 (railway and avionics) architectures comprise two or more levels of cache, the last of which is shared between cores. In addition to average performance, MLC impacts noticeably other metrics specially sensitive to real-time systems: worst-case execution time (WCET) estimates, i.e. guaranteed performance; hardware reliability – particularly critical in the space domain and other harsh environments; and complexity that affects compliance with safety standards (e.g. ISO26262 [17]).

The (cache) write policy determines how writes to lower (L1) cache levels, those closer to the cores, are handled. Under write-through (WT), write operations are performed in the lower cache and are forwarded to the higher (L2) cache level so that both caches hold consistent data. With write back (WB), write operations are only performed in the lower level cache, and the update of the next level is postponed until the cache lines containing the dirty data is evicted from the lower level cache. The write policy impacts the write-miss policy (write-allocate or not write-allocate), the cache coherence solution (e.g. in snooping-based protocols the write miss policy determines – together with the inclusivity protocol – the set of actions to take on a read/write to local and global data), and the reliability solution (e.g. WT usually requires low-overhead parity in lower level caches and ECC in higher level caches, whereas WB requires ECC in dL1 to keep the reliability of data not backed up in L2). Due its remarkable impact on the overall MLC cache design, the write policy affects metrics as important as guaranteed performance, energy/power, and reliability.

Interestingly, each write policy offers a different trade-off among the different metrics and MLC complexity. Hence, the design of the write policy requires finding a balance between them. The latter goes beyond a simple high-performance and real-time classification. Instead, for a given area (e.g. real-time), the particular application domain defines the relevance of each metric and hence, the write policy to use. For instance, in the space domain, due to exposure to radiation, hardware reliability plays a much more important role than in railway. Likewise, performance is much more relevant in automotive, where performance needs are expected to increase by 100x in coming years [4], than in space. In this line, we make the following main contributions:

1. We make an in-depth analysis of both write policies, WT and WB, with emphasis on those metrics of relevance for real-time systems. WT simplifies coherence since most updated data is always in L2, and reliability since the more costly ECC is only needed in L2 with only parity being used in dL1. However, as the pressure on the interconnection (NoC) increases – as a result of integrating more cores – the contention on the NoC generated by writes under WT greatly reduces guaranteed performance (i.e. increases WCET estimates). Further, WT increases energy consumption as each write accesses the NoC and the larger L2. With WB, each write to dL1 does not result in accessing the NoC, with considerable energy consumption reduction; and exceptional WCET reductions. Yet, WB complicates coherence and reliability, increasing cache complexity.
2. We propose Hybrid Write Policy (HWP), a low-overhead mechanism that takes advantage of the good properties of each policy. Building on WT, we attack its average and guaranteed performance issues, with a mechanism that builds on shared/private data classification hardware and applies WT to shared data and WB to private data. HWP removes write-through operations on private data, which in general are the most accessed data, while keeping it for shared data, so cache coherence can be managed as in pure WT caches. At hardware level, in the the Memory Management Unit (MMU) or Memory Protection Unit (MPU), HWP incurs negligible cost for tracking whether memory pages are shared or private and other page properties such as read/write permissions.
3. We evaluate WCET estimation, reliability, energy consumption and coherence cost of HWP. Our results show that for those scenarios in which tasks have limited data sharing, HWP delivers performance similar to WB. Even when the percentage of shared data is as high as 40% HWP remains competitive in all evaluated metrics (other works estimate the percentage of shared data in multiprocessor programs ranges from 25% [13] to 17% [15]). Overall, our design has a simplicity comparable to WT in terms of coherence, while achieving average/guaranteed performance and energy consumption comparable to WB.

The rest of this paper is structured as follows. Section 2 introduces basic concepts of MLC. Section 3 shows some of the main tradeoffs in the design of the cache write policy. Section 4 details our proposal (HWP) in terms of average and guaranteed performance, energy, reliability and coherence control. Section 5 provides empirical evidence of HWP benefits on our evaluation framework. Section 6 presents the most relevant related works. Section 7 summarizes the main conclusions of this work.

2 Background

When designing a multilevel cache hierarchy, see the illustrative example in Figure 4, there are several design choices to be made, which are not independent of each other but quite tightly correlated. In addition to the write policy we have.

With write allocate (WA), on a write miss data is fetched into cache, as it is the case for read misses, and once fetched, the write operation occurs. With no-write allocate (nWA), on a write miss the write operation is simply forwarded to the next cache level (or memory). Both WT and WB can use either of these write-allocation policies, but we only consider WB-WA and WT-nWA caches, since they are the most common choices. Though, our analysis can be extended to other combinations.

The *inclusivity* of the lower cache levels into the upper cache levels (those closer to memory), imposes that all contents in the lower level cache are also included in the upper level cache. Hence, whenever a cache line is evicted from the upper level cache, all cache lines in the lower level cache holding any of the contents of the cache line evicted in the upper level cache, are also evicted. Under an *exclusivity* approach, cache lines can be stored only in one of the two levels involved. When a new cache line is fetched by the processor, it is typically fetched into the lower level and removed from the upper level. When a cache line is evicted from the lower level it is moved up to the next level. Finally, *non-inclusive* caches are those where no constraint is imposed on whether cache lines are stored in upper or lower cache levels. This is a common choice for instruction caches since they are typically read-only and thus, cache lines can be simply removed on an eviction.

Snooping and directory-based approaches are the most commonly used ones for implementing cache coherence in multicores. For a moderate number of cores, snooping is in general the preferred mechanism because it is faster and much easier to implement and verify [28]. We use it as reference mechanism in this paper. We also assume a bus interconnect, although other interconnect networks would also benefit from our solution. Under snooping, writes can be handled in two different ways: write invalidate or write update. We focus on the write-invalidate MESI (Modified, Exclusive, Shared, Invalid) protocol as one of the most common that also supports write-back caches. We also cover a simple valid/invalid (V/I) protocol, used for WT caches. Under MESI when a snoop write request arrives to cache, the cache invalidates its own copy, if the cache has it. MESI distinguishes between data that is shared (i.e. exists a copy of the same data in another dL1), exclusive (it only exists a copy in the local dL1 and is clean), modified data (i.e. the data only exists in the local dL1 and it is dirty), and invalid data. Coherence is often implemented on top of inclusive caches, so that coherence can be checked in L2 and, only on a read/write request from a core that hits in L2, the dL1 caches of the other cores might be accessed. Under WT with a simple V/I, coherence is completely managed in L2 and, upon a shared cache line write request, it is immediately invalidated in the dL1 caches of the other cores and the data is delivered right away. When dL1 caches are WB and use MESI, on an L2 match a complex process is initiated to invalidate the corresponding dL1 lines, which may be dirty. This stalls the requesting core, with the L2 not accepting further requests until the current one is resolved. This occurs when the potentially dirty line in the dL1 of another core is written back to L2 and invalidated. The fact that accesses may be multi-cycle and non-pipelined to manage coherence imposes the use of complex logic. This may increase design cost and additional power, while significantly affecting critical circuit paths and limit operation frequency. Alternatively, the coherence protocol can be handled at each dL1 cache and the interconnect. With this approach dL1 caches snoop the bus to monitor the activity from the other cores and cores have to expose its activity to the interconnect. This removes the need for using inclusive caches but comes at the expense of an increase in power and complexity in the on-chip interconnect. For instance, in the absence of a shared medium ensuring the in-order delivery of core/memory transactions is difficult [29].

Error correction codes such as single error correction double error detection (SECDED) are inherently complex mechanisms that introduce some delay to encode/decode data. When used in dL1 caches, SECDED can increase cache latency. To prevent so, complex logic is put in place to recover a correct state if data is delivered to upper cache levels unchecked. For L2 caches, SECDED can be more complex since their impact on performance – for instance by making cache access to take an extra cycle – have relatively lower impact than for dL1 caches. When there is no need to correct errors in the dL1 cache, a simple parity mechanism can be used instead of SECDED.

Table 1 Percentage of stores executed by the EEMBC Automotive and MediaBench suites.

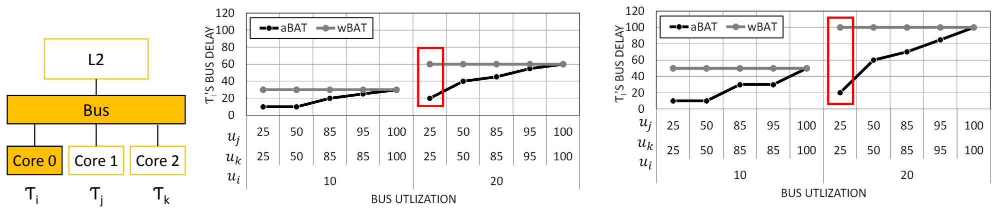
EEMBC	%	EEMBC	%	MediaBench	%	MediaBench	%
`a2time`	5%	`matrix`	3%	`adpcm.d`	13%	`mesa.m`	12%
`aifftr`	18%	`pntrch`	0%	`adpcm.e`	14%	`mesa.o`	14%
`aifirf`	8%	`puwmod`	12%	`epic.d`	6%	`mesa.t`	9%
`aiifft`	18%	`rspeed`	14%	`epi.e`	5%	`mpeg2.d`	10%
`basefp`	2%	`tblook`	6%	`g721.d`	8%	`pegwit.d`	6%
`bitmnp`	11%	`ttsprk`	4%	`g721.e`	9%	`pegwit.e`	6%
`cacheb`	16%			`gsm.d`	3%	`pgp.d`	5%
`canrdr`	15%			`gsm.e`	3%	`pgp.e`	13%
`idctrn`	8%			`jpeg.d`	6%	`rasta`	8%
`iirflt`	7%			`jpeg.e`	10%		

3 Tradeoffs in the Design of Cache Write Policy

MLC are one of the main hardware blocks in a multicore architecture devoted to improve performance and reduce the energy/power profile of applications. MLC aim at rapidly and efficiently satisfying data/instruction requests coming from the cores, while maintaining the coherence (i.e. the particular value returned on a read), consistency (i.e. when data is available), reliability (physical integrity) and more recently security (i.e. protection against unwanted/unauthorized actions). The cache write policy, which handles write operations, is at the core of the complexity of MLC since it has a direct impact on the design of other policies. In this section we analyze the impact of WT and WB policies on reliability, inclusivity, and coherence choices. We also analyze their impact on performance (average and guaranteed), reliability, and energy/power. For the latter, the results obtained from several controlled experiments are used as supporting argument.

3.1 Write-Through (WT)

Under WT, each store operation is sent to the L2 so it uses the NoC, which can significantly increase the pressure on it. In the core, the store buffer decouples the commit (finalization) of the stores so that they do not block the pipeline. To that end, once a store reaches the commit/writeback stage, it updates dL1 and in parallel it is placed in the FIFO store buffer allowing the execution to continue. The store is forwarded to L2 when it reaches the head of the store buffer and there is available NoC bandwidth. The store buffer can significantly mitigate the impact of stores in single-core architectures, but rapidly becomes insufficient in multicore. This is better illustrated with an example: let us assume that a bus connects dL1 and L2 caches and each store operation uses it for k cycles. As long as the frequency of stores is (on average) below $1/k$, they will not significantly affect processor performance – unless they are bursted which we do not assume in this simple example. However, in a multicore architecture with N_c cores, as soon as the pressure in the bus increases, the actual duration of a store becomes $k \times N_c$, i.e. $k \times (N_c - 1)$ cycles of contention and k cycles for the bus access. In this scenario, stores become a performance issue as soon as their density reaches $1/(k \times N_c)$. As an illustrative example, Table 1 shows the percentage of store operations executed by EEMBC and Mediabench benchmarks, see Section 5 for more details on the experimental setup. The average percentage of stores is 9%. Further $k \times N_c \in [15, .., 20]$ – and hence it is higher than 1/9, for multicores with 4-8 cores. To make things worse, the percentage of memory operations is growing in emerging data-intensive real-time applications,



(a) Setup (b) Same duration ($l^i = l^j = l^k$) (c) Different duration ($2 \cdot l^i = l^j = l^k$)

Figure 1 τ_i aBAT and wBAT as a function of its load and its contenders' (τ_j and τ_k) load.

e.g. applications in cars managing data coming from different sensors such as radar, LIDAR, and stereo cameras. Intuitively, this problem can be alleviated by using a crossbar between the dL1 and the L2, at the expense of increased hardware cost. However, this would just shift the problem from the bus to the L2 itself, since L2 access latency is longer than that of the crossbar. Further, to preserve coherence, each store must be allowed to reach any part of the entire L2 cache, which defeats any attempt to mitigate the problem by partitioning the cache space.

The impact of WT on average performance due to NoC contention magnifies for guaranteed performance, causing inflated WCET estimates. This comes from the fact that worst-case time allowances must be done in the WCET estimates to factor in the impact of NoC contention. In general, no assumption can be made on how the requests of the different running tasks are interleaved in the use of the bus. The exception to this are some static timing analysis techniques that keep track of the worst-time when each request from each core can be issued, and hence are able to exactly determine how requests overlap in the access to shared resources [22][14][21]. This, of course, comes at a significant cost, including the increasing effort of making a cycle-accurate model of the MLC system and processor, and increased analysis computation time. Further, this analysis, despite producing (in general) tighter WCET estimates, makes them non time-composable so that any shift in any task requires performing the WCET estimation for all tasks. Hence, to increase time composability and reduce costs, worst-case assumptions are made on how tasks' request are aligned [25, 8, 19]. This is better illustrated with an example. Let us assume a bus connecting the L2 to 3 cores (respectively executing tasks τ_i, τ_j, τ_k) and all bus requests using the bus for the same duration l (shown in Figure 1 (a)). The best overlapping scenario for average Bus Access Time (aBAT) happens when requests of the task under analysis (τ_i) and the contender tasks (τ_j, τ_k, ...) do not overlap as long as the bus utilization of all tasks is below 100%, and when the utilization goes over 100% the minimum overlap happens. For instance if a τ_i uses the bus for 20% of the time and τ_j for 90% of the time, τ_i gets affected only 10% of its time. The worst overlapping scenario for bus access time (wBAT) is assuming that requests from τ_i arrive in the same cycle as the requests from the contender tasks, but τ_i systematically gets the lowest priority. Figure 1(b) shows how worst-case BAT gets much more affected than average BAT due to contention for different scenarios of bus utilization of τ_i and its contenders. We see that wBAT is significantly affected even for low bus utilization. For instance, for utilization $u_i = 20\%$, $u_j = 25\%$, $u_k = 25\%$ for τ_i, τ_j, and τ_k respectively (see red rectangle in Figure 1(b)), τ_i suffers no delay in the best case aBAT and in the worst case it goes to 60% (a 2.4 increase). Further, typically store operations take shorter than load operations accessing the cache (no need to wait for a response), which translates into a scenario in which τ_i requests take shorter than its contenders' request. We see in Figure 1(c), for a scenario in which τ_i requests take half of its contenders, that the impact of contention

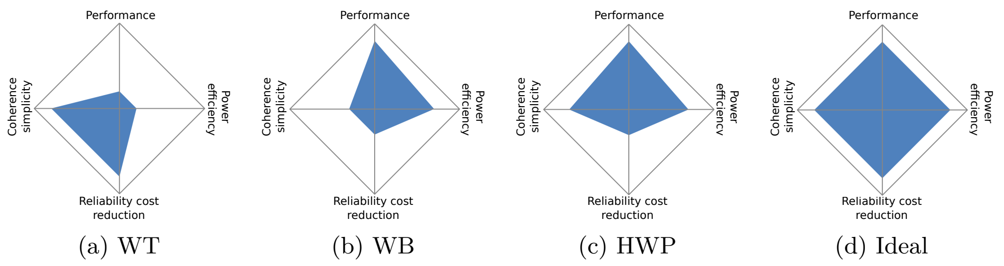


Figure 2 Visual comparison of the WT, WB and HWP for the different metrics discussed.

on WCET estimates increase. For instance, for utilization $u_i = 20\%$, $u_j = 25\%$, $u_k = 25\%$ for τ_i, τ_j, and τ_k respectively (see red rectangle in Figure 1(c)), τ_i suffers no delay in aBAT but a 100% in the wBAT.

Continuous store accesses to the L2 cause performance and WCET degradation but can also increase power consumption. Updating values with WT policy implies accessing the bus and L2, even if the core updating the values is the only consumer of this data. This can have a significant impact on the overall power consumption. For example, when running `a2time` from the EEMBC automotive benchmark suite in our reference processor setup (see Figure 4 and Section 5.1), the 14% of the energy consumption comes from the bus and L2.

Under WT, reliability can be handled with reduced overhead. A usual tradeoff consists of using only parity for error detection in dL1 caches, and (usually) apply it at double-word level, that is, using 1 parity bit for 64 data bits (8 bytes). This results in low overhead of around 1.6% (1/64). Furthermore, the operations needed to compute the parity (XOR) can be carried out in parallel and hence are unlikely to affect cycle time. On a parity error, however, hardware support is needed to squash the execution of the instruction that obtained erroneous data and following instructions. On completion, error-free data is fetched from L2 and execution resumes. Alternatively, parity can be checked before delivering the data to remove the need of squash logic. However, this would likely increase cache latency since XOR gates to compute the parity bit may easily need an extra cycle. WT parity-protected dL1 caches are used in combination with SECDED-protected L2 caches. The latter is achieved with ECC that carries an inherent area and logic for its implementation. Typically, SECDED requires 8 code bits per 64 data bits (so $\approx 12.5\%$ extra bits), with negligible impact on L2 performance, since although an additional cycle may be needed to deliver data corrected, this operation is fully-pipelined. Hence, L2 latency may increase by 1 cycle, thus slightly increasing the latency of dL1 read misses, which are generally scarce, but without affecting L2 throughput. Note that on the event of detection of an error in dL1 in a given cache, it is simply discarded and data is fetched from upper cache levels since a correct copy of the data exists in L2 or beyond.

dL1 WT caches simplify coherence management. In particular, dL1 WT caches are made inclusive L2. As a result, when shared data exists in data dL1 (dL1), up-to-date copies of the data is also present in L2. Hence, coherence can be managed in L2 and, upon shared data modifications, the corresponding cores' dL1 caches receive (infrequent) invalidation requests. With WT caches a simple invalidation protocol (V/I)) is enough.

Overall, WT can negatively affect average and worst performance – the latter more intensely– and energy. On the positive side, it can be used with low-overhead coherence and reliability solutions. These properties are summarized in Figure 2(a) in a qualitative manner, with Figure 2(d) showing the ideal scenario.

3.2 Write-Back (WB)

For low core counts, the small average performance improvement of WB over WT does not compensate its additional validation and design costs. However, as the number of cores of multicore real-time systems increases, WB becomes more attractive.

WB significantly reduces the number of bus and L2 accesses compared to WT. Furthermore, since worst-case contention is quite proportional to the number of accesses, WCET estimates are typically much lower with WB than with WT.

WB access count reduction to shared resources decreases the power consumed by those resources. In our setup, the bus and L2 accounts on average for 13% of the system energy, and hence reducing its utilization translates into a non-negligible energy reduction. Also, the need for higher reliability in the data dL1 cache (dL1) increases the power used by the system due to the extra bits and logic needed to implement, for instance, SECDED codes. Finally, since invalidation operations due to shared data accesses may require invalidating dirty lines in dL1, this may cause extra energy consumption to write data back to L2.

When WB is used in the dL1, the data most frequently updated/sensible can be spread between multiple caches (the different dL1 caches and L2). In this scenario, error detection in dL1 and error correction in L2 is not enough, since some data is only updated in dL1 and, upon an error, it could be detected but not corrected. In this case, there are two possible implementations of ECC in dL1, each one with its advantages and drawbacks:

- Under **Data delivery after correction** data is read from dL1, then ECC checked (and eventually data corrected), and finally data is delivered. Unfortunately, checking the ECC code increases access latency by 1 cycle. While such operation can be pipelined, thus not increasing dL1 utilization, the effective latency for data read increases.
- Under **Data delivery before correction** data is read from dL1 and delivered as if it was error-free. In parallel, ECC is checked and, upon an error detection, the affected instruction and subsequent ones need to be squashed. Then, the execution can be resumed using the corrected data. While such process has negligible impact in performance (radiation errors occur only sporadically), the logic for squashing instructions and resuming execution may be complex. However, such logic is analogous to that of WT caches when operating with parity.

With WB caches V/I is not enough because data can be in another state apart from valid or invalid, namely, modified state. Because of this, we will use MESI (an enhancement over MSI) for WB caches. Maintaining cache coherence in multicores with WB dL1 caches requires frequent accesses to other cores' dL1 caches to verify whether shared data is there and, eventually, retrieve them (if dirty) or invalidate them (if the ongoing access is a write or data is not dirty). Depending on the inclusivity of the cache system, we find two possible scenarios (exclusive caches are infrequent so we do not discuss them here):

- **Inclusive**. In an inclusive cache system (the most convenient solution) the updated data is in dL1 or L2, but L2 has all the tags. This means that all coherence requests can go to L2, and only upon a match ask dL1 for the data it needs.
- **Non-inclusive**. If the system is non-inclusive, there is no unique cache that "knows" where all the data is. This means that any request for data has to be communicated to all caches (all the private dL1 and L2), and any cache can answer with the data. This complicates the coherence protocol design. Hence, we disregard this option.

Either case, whenever some data is requested and the L2 experiences a hit on shared data, it must stall the request and block further L2 accesses. Then, the corresponding dL1 caches deliver the data if dirty. Since dirtiness in dL1 caches is not known a priori by the

Table 2 Commercial processors and their characteristics.

Processor	Cores	Freq.	L1 WT?	L1 WB?
ARM Cortex R5	1-2	160MHz	Yes, ECC/parity	Yes, ECC/parity
ARM Cortex M7	1-2	200MHz	Yes, ECC	Yes, ECC
Freescale PowerQUICC	1	250MHz	Yes, ECC	Yes, parity
Freescale P4080	8	1.5GHz	No	Yes, ECC
Cobham LEON 3	2	100MHz	Yes, parity	No
Cobham LEON 4	4	150MHz	Yes, parity	No

L2 cache, it must remain blocked long enough to allow the dirty data to be read from the corresponding dL1 and be sent to L2. Then, the L2 can update its contents, deliver the data and hence, serve the request. However, the complexity of the logic to manage all this process synchronously and across multiple cycles and components may affect critical circuit paths, which can carry a reduction of the operating frequency.

Figure 2(b) presents in a graphical manner the assessment we have done on WB. We can see that while WT is better in reliability and coherence simplicity, it performs worse on performance (both average and worst-case) and power.

3.3 Cache Write Policy in Some Commercial Architectures

To better illustrate the quandary chip vendors face when selecting the write policy, we have analyzed the miss policy of several commercial processors[5].

The ARM Cortex R5 [3] is a 1 (or 2) core processor that implements both WB and WT in the dL1 cache, both with parity and ECC. This means that either policy can be selected in a configuration register. The ARM Cortex M7 [1] is a low-performance processor. Like the previous one, it implements both write policies in the dL1 cache, but it only has ECC in the L2 cache. ARM acknowledges that using dL1 ECC may have an impact on operating frequency due to the XOR trees for the ECC when getting the data from the cache. Thus, depending on the particular chip implementation of the ARM IP processor we might have to decrease maximum operating frequency or require two cycles to access the dL1 to support ECC in the dL1 and have the possibility of recovering from errors in the cache when WB is enabled. Hence, despite in general WB caches perform better the strong reliability constraints in safety-critical systems and the associated overheads incurred due to implementing ECC in WB caches makes chip vendors offer the users the possibility to choose between WT/WB according to the needs of their application. However, this forces chip vendors to carry with the effort and responsibility to implement and validate both.

The Freescale PowerQUICC [32] implements WB in the dL1 with parity and the L2 with ECC. This lead to a system where not all cache bit-flips can be recovered. In that respect, Freescale states that the probability of errors is so low that the target application domain should accept the possibility of having “unrecoverable" errors.

The Cobham Gaisler LEON3 [12] is a dual-core running at 100MHz, with a 5 stage in-order pipeline. It is designed for critical real-time systems, and implements WT in the dL1 cache, so that reliability can be handled in L2 with more robust ECC. The LEON4 [10] comprises with 4 cores running at 150MHz with a 7 stage pipeline. It has the same critical real-time systems scope as its predecessor, and the same write policies in the dL1.

[5] Core and frequency numbers have been obtained from specific processor implementations [36, 35, 26].

WT has been widely implemented in the last level of private caches (mainly in dL1) due to its simplicity (no need for reliability and simple coherence) and its acceptable single-core performance. However, in future multi- and many-cores, the increased number of accesses to shared resources will cause a dramatic increase in average execution time and the WCET estimates. WB caches have performance and energy consumption benefits over WT in mid to high core count processors. However, this performance comes at a complexity cost in the coherence protocol mainly and, to a lower extent, in the reliability mechanisms.

4 Hybrid Write Policy (HWP)

HWP low-overhead approach addresses WT average and guaranteed performance issues while reducing overheads w.r.t. WB. HWP eliminates the additional cost of coherence for WB caches and, simultaneously, keeps WT operations limited to a small fraction of write operations so that efficiency is close to that of WB caches, see Figure 2(c).

In order to reach its goals HWP builds on the following observations. First, cache coherence management with WB caches is costly and complex because cache lines accessed may reside dirty in local dL1 caches. Second, private data is not affected by cache coherence, so conceptually it is irrelevant whether such data is dirty or not in dL1 caches. And third, a significant percentage of memory data is only accessed by one processor (also in parallel applications) and, thus, does not require keeping coherence (e.g. 75% of the access are reported as private in [13] and around 83% in [15]).

From those observations, we design a new policy (HWP) that manages private data as in WB caches and shared data as in WT caches. With HWP, memory contents are classified at page granularity as either shared or private, which has been shown to be a very convenient granularity for private/shared data classification [15, 6]. In particular, as long as a page contains any shared data, it is (pessimistically) classified as shared. Otherwise, it is classified as private. On a write to shared data, HWP writes it through to L2 cache (a la WT). Meanwhile write operations to private pages are not propagated to L2 (a la WB), hence decreasing contention in the access to L2.

Next, we discuss the key characteristics and implementation details of HWP, with emphasis on how to classify data as private or shared (and the appropriate granularity to do so), how to check whether data is shared or private to decide whether to proceed as in a WT or WB cache, how cache coherence needs to be managed, what the reliability implications are, and how contention in the access to L2 is mitigated.

4.1 Data Classification

Orthogonally to HWP, a mechanism is needed to classify data as private or shared. Techniques exist to that end, with some of them [15] already integrated on a real hardware platform (LEON3 processor) and Linux, providing evidence of its feasibility. Interestingly, private/share data classification can be performed at different levels (e.g. cache line size).

Private/shared information can be managed at fine granularity (e.g. cache line level). This would allow a much finer classification but at the cost of higher area and energy overheads [15]. Additionally, performing the shared/private classification at page granularity makes it possible using OS functionality to reduce hardware implementation overheads [6].

Ho et al. [15] and Cuesta et al. [6] show that the most convenient granularity to classify data is page level. With this solution, whenever a piece of data is shared between two cores, the whole page in which the data is is marked as shared. Hence, this solution pessimistically assumes that all data in a shared page is shared. As part of that solution, the information on

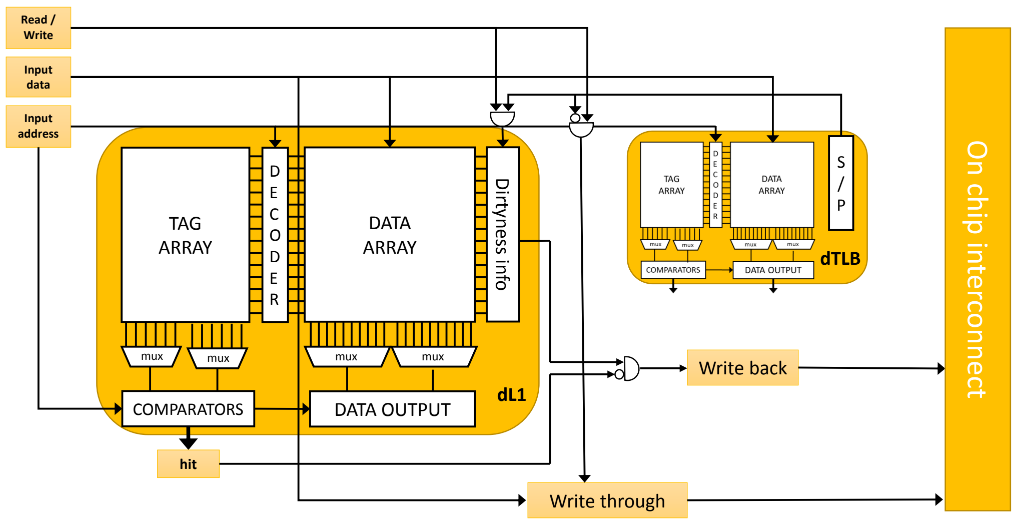


Figure 3 Schematic of HWP cache access protocol.

private/share information can be stored in the Memory Protection Unit (MPU) or Memory Management Unit (MMU) for each page along with other information such as whether pages are user-level or supervisor-level, whether they are read/write or read-only, and whether they are cacheable or not. Such information is often cached in the Translation Lookaside Buffer (dTLB) together with address translation. In most processors dTLBs are accessed in parallel with dL1 caches for fast address translation and for verification of the permissions to read/write in specific memory pages. Hence, they can store private/shared information.

In real-time systems an alternative approach to those hardware approaches is possible with software address space partitioning. Many OS use address spaces (i.e. a range of addresses) to map specific I/O devices. Also RTOS like PikeOS use separate address spaces to implement resource partition. Furthermore, in the automotive domain, AURIX architectures come equipped with caches and different memory types (e.g. flash, ram). From the software side, address ranges are defined to map data/instructions to the desired memory and or to make data cacheable or non-cacheable. Hence, address space can be partitioned assigning a particular address range to shared data.

The main disadvantage of dynamic hardware solutions is that data re-classification is needed. This happens, for instance, when a page is first loaded by one core (hence classified as private) and then accessed by another core (being reclassified as shared). This does not only create predictability issues in real-time systems, but it also adds complexity to HWP, including writing through all data (dL1 lines) of this page in the owner core, while managed those same data with WB in the other cores. This complexity is avoided with the classification based on software address partitioning, *which is the solution we assume in this paper*, without loss of generality.

4.2 Private/Shared Data Management

The way in which data is accessed under HWP varies depending on whether data is shared or private. This is graphically illustrated in Figure 3.

On a load/store access, the dL1 and the dTLB are accessed in parallel. In case of a dTLB miss, it is served first before proceeding with the access, as done regularly in most processors. Note that address translation is typically needed before accessing the L2. Therefore, while

Table 3 Timing of a dL1 hit (dTLB hit) under HWP.

		cycle 1	cycle 2
LOAD		Read dL1, Read dTLB	
STORE	Private	Write dL1, Read dTLB	Update dirtiness bit
	Shared	Write dL1, Read dTLB	Write L2

serialization of dTLB misses and dL1 accesses may be unnecessary for some dL1 hits, dTLB miss rates are usually extremely low, and their occurrence together with dL1 hits is even more unlikely since this can only occur if data from the page has been fetched and sufficient evictions occurred in the dTLB but not in the dL1.

4.2.1 Hit in dL1

In case of a dL1 hit (and dTLB hit), the shared/private information determines whether the line hit needs to be marked as dirty or not. If the line belongs to a private page ($S/P = 0$) and the access is a write operation ($W/R = 1$), the dirtiness bit is set. The separation of data and dirtiness information poses no issue since dirtiness information can be accessed systematically one cycle after, as it is only needed in case of a miss to decide whether the evicted cache line needs being written back. Also, in case of dL1/dTLB hit, if the line belongs to a shared page ($S/P = 1$) and the access is a write operation, data is written through L2 as in a regular WT MLC.

In terms of timing, Table 3 shows the different possible scenarios and their timing. After the processor request, regardless of whether it is a read or a write, both the dL1 and the dTLB are accessed in parallel. In the case of a read, at the end of the first cycle the data is available and is served to the processor. In the case of a write, at the end of the first cycle the dTLB determines whether it is a write to a shared or private page. If the store targets a private page, the dirtiness bit is updated in the dL1 cache in the second cycle, and the request is completed. However, if it is a write to a shared page, a write request is issued to the L2.

4.2.2 Miss in dL1

On a dL1 miss, WT management is performed as for hits. However, if the miss corresponds to a read operation ($W/R = 0$) or the address is private ($S/P = 0$), the line is fetched from L2 and allocated in the dL1 data cache. Note that we assume the usual case where WT implements no-write-allocate (nWA) policy on write misses, whereas WB implements write-allocate (WA). Different write allocate policies could be implemented such as, for instance, WA (or nWA) regardless of the privateness of the data accessed.

In Table 4 we see the different scenarios that can happen with a dL1 miss. In the first cycle, both the dL1 and the dTLB are accessed in parallel. At the end of the cycle, if the line to be evicted is dirty, the dL1 sends a dirty eviction request to the upper level. In the load scenario, the next cycle (3 if the line was dirty, 2 if it was clean) the line is requested to the L2. After n cycles, the cache line arrives to the dL1 and the data is be available. In the case of a store private, it also requests the line to the L2. When the answer comes, it updates the dL1 and update the dirty bit (allocate on private data). Finally, on a store to a shared line, a request for the write is sent to the L2, and no update occurs in dL1 (no allocate for shared data).

Table 4 Timing of a dL1 miss (dTLB hit) under HWP.

		cycle 1	cycle 2	cycle 3	...	cycle n+3
LOAD		Read dL1, Read dTLB	if dirty -> Eviction	Request line L2		Read dL1
STORE	Private	Write dL1, Read dTLB	if dirty -> Eviction	Request line L2		Write dL1, Update dirtiness bit
	Shared	Write dL1, Read dTLB	if dirty -> Eviction	Write L2		

4.3 Non-Functional Metrics

This section makes a qualitative assessment of the benefits of HWP over WT and WB. Quantitative comparisons are carried out in the Section 5.

Under HWP, shared data is consistently stored in L2, making that all shared data in dL1 caches is necessarily non-dirty. As a result, with HWP coherence is managed as in the case of pure WT caches, hence keeping its low- cost and complexity benefits and avoiding the overheads related to WB caches. With HWP V/I is enough, as for WT, because the shared data will always be updated in a single place (L2), so we do not need a Modified state in the dL1 to keep track of who has the most updated data.

Since shared contents are written through to L2, the fraction of dirty dL1 cache contents is smaller than in pure WB caches. Yet some dL1 cache contents can be dirty. Hence, error correction capabilities are still required in dL1, as in the case of pure WB caches. A simple software solution to reduce the associated costs consists in marking the pages storing error-sensitive data as shared. This way the only data that could be lost would be the private one. However, for critical applications, the same reliability technique used in WT (SECDED in dL1) can be used.

Under WT, performance issues relate to contention in the NoC and the L2 due to write-through stores. With HWP, this problem is alleviated, restricting write throughs to stores to shared data. Obviously, the lower the number of accesses to shared data, the lower the number of WT operations, and hence, the lower the contention and the lower the sensitivity to contention. In general, programs are designed to reduce access count to shared data (25% [13] and 17% [15]), which usually carries a serialization of tasks.

In general, power consumption relates to the activity performed (dynamic power) and execution time (static power). By limiting the number of WT operations, dynamic power is reduced drastically w.r.t. pure WT designs. By reducing contention, execution time is also lower than for pure WT designs, thus reducing static power.

Overall, our HWP hybrid cache design offers a globally better tradeoff than WB and WT. This is illustrated in Figure 2(c). As shown, our design offers performance and power close to that of WB, with similar reliability overheads, but much lower complexity for the management of shared data. When compared against WT, coherence management cost is identical, performance and power are much better, and only reliability costs are higher.

5 Evaluation

In this section we quantitatively assess the benefits of HWP over conventional write policies (WT and WB). We use the metrics presented in previous sections, namely, guaranteed and average performance, energy consumption, coherence overhead, and reliability.

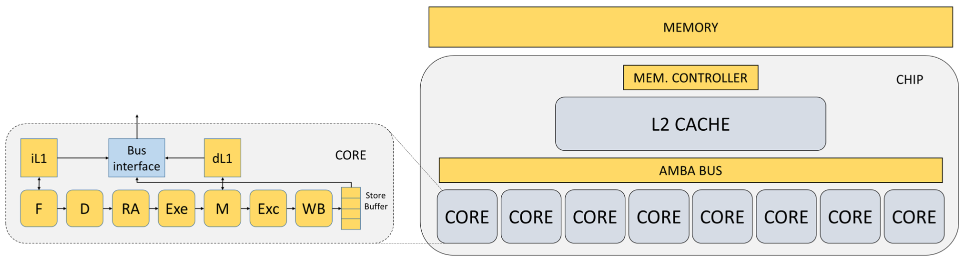


Figure 4 Block diagram of the main elements of our NGMP-based 8-core architecture.

Table 5 Benchmarks in EEMBC Automotive and MediaBench we use in this paper.

suite	List of benchmarks
EEMBC Auto	`a2time`, `aifftr`, `aifirf`, `aiifft`, `basefp`, `bitmnp`, `cacheb`, `canrdr` `idctrn`, `iirflt`, `matrix`, `pntrch`, `puwmod`, `rspeed`, `tblook`, `ttsprk`
Media-Bench	`adpcm.d`, `adpcm.e`, `epic.d`, `epic.e`, `g721.d`, `g721.e`, `gsm.d`, `gsm.e`, `jpeg.d`, `jpeg.e`, `mesa.m`, `mesa.o`, `mesa.t`, `mpeg2.d`, `pegwit.d`, `pegwit.e`, `pgp.d`, `pgp.e`, `rasta`

5.1 Reference Architecture and Benchmarks

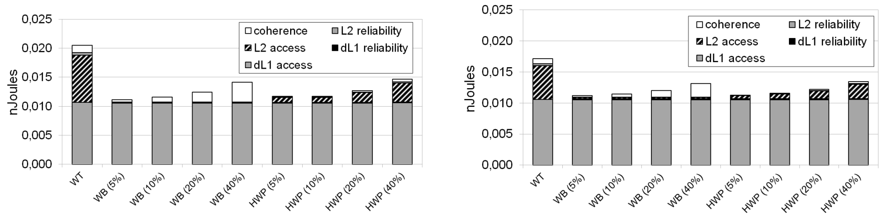
We use a simulation environment based on the cycle-accurate SoCLib [33] framework to model the architecture of the Cobham Gaisler's Next Generation Multipurose Processor (NGMP), as a representative of current bus-connected multicore MPSoC. The main difference lies in that we scale the number of cores from 1 to 8 in our experiments (See Figure 4) to assess the impact on the different metrics, while the NGMP specifically features 4 cores. Each processor implements a SPARC V8 architecture [11]. Each LEON4 core comprises seven stages: fetch (F), decode (D), register access (RA), execution of non-memory operations (Exe), dL1 access (M), Exceptions (Exc) and write back (WB). The execution units are equipped with an integer and a floating-point unit (FPU). Each core has its own private instruction (il1) and data (dL1) caches that are 16KB, 4-way with 32-byte lines. Processors are connected by a shared on-chip round-robin arbitrated AHB processor bus to a shared L2 cache and memory. The shared second level (L2) cache is split among cores, each receiving one way of the L2. All caches use LRU replacement policy.

We evaluate a large subset of the EEMBC automotive [27] suite comprising common critical real-time applications in automotive systems and MediaBench [20] comprising embedded applications such as multimedia and communications. The benchmarks we use from both suites are listed in Table 5. We create several scenarios in which we vary the percentage of accesses targeting the address range for shared data.

5.2 Energy

As presented in Section 3 each cache write policy carries side effects on the write-miss policy, the reliability solution, inclusivity, and the coherence solution. This affects the set of activities carried out by each task, the energy cost of each activity and hence the overall energy profile of each task. Further, the complexity of each write policy varies which affects its 'intrinsic' energy consumption.

We assess the energy usage under each policy using CACTI [24], the state-of-the-art integrated model for cache and memory access time, cycle time, area, leakage and dynamic power consumption, configured with the NGMP cache parameters. With CACTI we break-



(a) `EEMBC` (b) `Mediabench`

Figure 5 Average energy breakdown per cache access for EEMBC and Mediabench.

down the energy usage of each cache access into 5 components: dL1 access, dL1 reliability, L2 access, L2 reliability, and coherence.

We present the average cache access energy consumption, across all EEMBC and Mediabench benchmark suites, in Figures 5 (a) and 5 (b) respectively. The difference across individual programs in each benchmark are not relevant, and hence are not shown. We compare WT, WB and HWP; and for the latter two, we assume three different scenarios depending on the percentage of accesses to shared data: 5%, 10%, 20% and 40%. Note that WT results do not depend on the percentage of shared data, since all writes go to L2.

We observe that the dL1 energy usage for an access is roughly the same for all write policies. The difference in the energy of the dL1 reliability solution is small, with WT having the lowest value due to the use of simple parity instead of ECC (used by WB and HWP).

We also observe that the lowest access energy profile is obtained for WB and HWT. In the case of WB, there are few L2 accesses, since stores do not access L2 every time, while the load access rate to the L2 is relatively low. HWP has a higher L2 access rate than WB since it writes shared data directly to the L2.

On the coherence side, WB has an increased amount of coherence-related messages as the shared data increases. Taking into account all components, WB and HWP consume roughly the same energy per access for a given ratio of accesses to shared data. Both show approximately a 42-50% per access energy reduction (depending on the percentage of shared data) with respect to WT. To sum up, when comparing the different write policies on the energy aspect, HWP has the same reduced energy consumption as WB compared to WT (up to 50%), but without the coherence complexity inherent to WB, as presented in Section 4.

5.3 Guaranteed Performance

WCET estimation is one of the most critical metrics for real-time systems, since it determines the guaranteed performance that the system can deliver. As presented before, WCET estimation is challenged by the use of multicores due to contention delay suffered by tasks.

In order to assess the benefits on WCET estimate reduction of HWP, we have created 1-, 2-, 4- and 8-task workloads, as presented in Table 6. Workloads have been generated using benchmarks from the EEMBC automotive suite (eembc1.X, eembc2.X) and from the MediaBench suite (media1.X, media2.X). Across workloads, the first task in each workload, the one for which WCET estimates are produced, comprise at least one benchmark with at most a 5% of stores, and at least one benchmark with at least a 13% of stores. The rest of the new benchmarks in the workload are selected randomly.

Modeling multicore contention is a concern for timing validation and verification as witnessed by a notable amount of works on the topic, summarized in [9]. Many measurement-based approaches – the most extended industrial practice – build on the availability of

Table 6 Benchmark mixes used to assess WCET estimates under different core counts.

Mix	main	cont1	cont2	cont3	cont4	cont5	cont6	cont7
eembc1.1	bitmnp							
eembc1.2	puwmod							
media1.1	g721.d							
media1.2	jpeg.d							
eembc2.1	bitmnp	a2time						
eembc2.2	puwmod	aifftr						
media2.1	g721.d	adpcm.d						
media2.2	jpeg.d	adpcm.e						
eembc4.1	bitmnp	a2time	matrix	rspeed				
eembc4.2	puwmod	aifftr	idctrn	ttsprk				
media4.1	g721.d	adpcm.d	gsm.d	pegwit.d				
media4.2	jpeg.d	adpcm.e	g721.e	pgp.d				
eembc8.1	bitmnp	a2time	matrix	rspeed	tblook	canrdr	aifirf	aifftr
eembc8.2	puwmod	aifftr	idctrn	ttsprk	basefp	cacheb	tblook	ttsprk
media8.1	g721.d	adpcm.d	gsm.d	pegwit.d	g721.d	pegwit.d	gsm.e	pgp.d
media8.2	jpeg.d	adpcm.e	g721.e	pgp.d	adpcm.e	jpeg.d	gsm.e	adpcm.e

performance monitoring counters (PMCs) [23, 25, 7, 18, 8]. From those we build on [18] since it captures the number of requests each core performs to the shared resources. This results in *partially time composable* WCET estimates, rather than *fully-time composable* ones that result from assuming that every single request of the task under analysis is delayed regardless of the load contenders put on the shared resources.

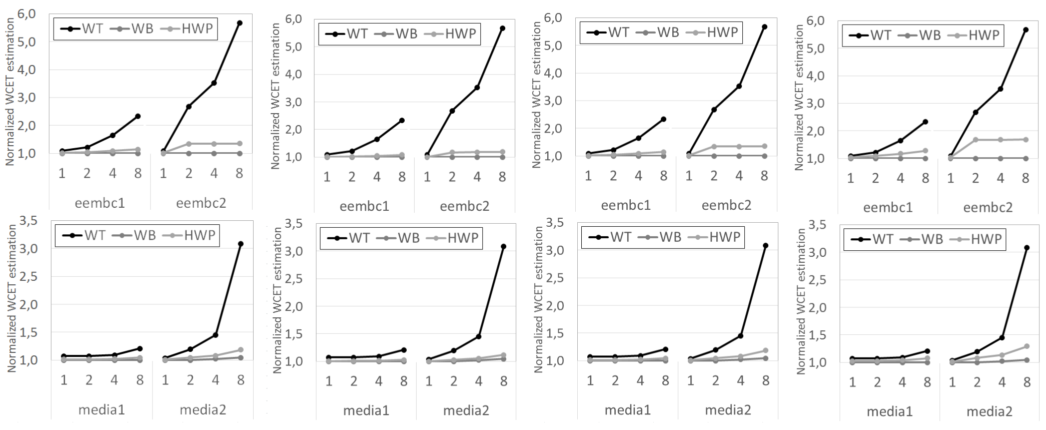
We illustrate the model [18] with a small example comprising one task under analysis or τ_a and a contender task or τ_b. When τ_b has more requests than τ_a, each request of τ_b is assumed to delay the requests of τ_a. The worst-case contention that τ_b can cause on τ_a, i.e. $\Delta_{b\to a}^{cont}$, is computed according to Equation 1, where n_b^t is the number of τ_b requests of type t and lat^t is the latency of that request type. Note that the model makes the worst case assumption of no overlap of requests, so each τ_b's request delays τ_a by its latency, i.e. lat^t.

$$\Delta_{b\to a}^{cont} = \sum_{t\in\mathcal{T}} min(n_a, n_b^t) \times lat^t \qquad (1)$$

In our case the request types are $\mathcal{T} = \{L2h, L2m, s2h, s2m\}$ corresponding to loads hitting and missing in the L2 cache, and stores hitting and missing in the L2 cache respectively, which can be tracked with existing PMCs [11]. The corresponding latency of each of these event type is [18] (in processor cyles): $lat^{L2h} = 9$, $lat^{L2m} = 7$, $lat^{s2h} = 1$, and $lat^{s2m} = 1$.

Note that it does not matter the type of τ_a requests but just it overall number $n_a = \sum_{t\in\mathcal{T}} n_a^t$. That is, the contention τ_a suffers depends on its total number of requests and the number of requests of each type of its contenders (τ_b in this case). The model factors in the case when τ_b has fewer accesses than τ_a that results in some τ_a requests not being delayed by any request from τ_b. The approach presented in Equation 1 for τ_b is followed for all the $Nc-1$ tasks simultaneously running with τ_a, where Nc is the number of cores. The reader is referred to [18] for more details.

Figure 6 shows the WCET estimate obtained for the first task under each cache write policy. WCET estimates are shown as the number of cores varies from 1 to 8. In order to simplify the comparison, all WCET estimates are normalized to the WCET estimate of



(a) 0% shared (b) 10% shared (c) 20% shared (d) 40% shared

Figure 6 Normalized WCET estimate for the first task in the workload under different core counts and percentage of shared data for the different write policies.

the first task when run in isolation under WB. We see that in all cases the tightest WCET estimates are obtained with WB. HWP obtains comparable results to those of WB and much better than those for WT. The latter gets rapidly worse as the core count increases. Note that Figures 6 (a), (b), (c), and (d) are not directly comparable, since for each figure WCET estimates are normalized to that of WB when the task runs on isolation.

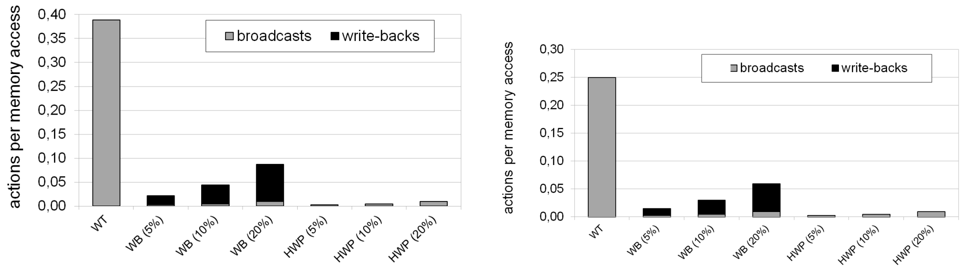
We also see that WT is not affected by the percentage of shared data, since it always updates the L2 regardless if the data is shared or not. WB does not show meaningful variations either, while HWP has small variations (mainly for eembc2 and media2). In all cases HWP is significantly better than WT.

Across all shared-data scenarios for WT we can observe that:

- Mix eembc2 suffers a significant increase in WCET estimates (more than 5x in the 8 core configuration). This is due to the combination of memory instructions the program under analysis executes (30% of all instructions) and the number of stores the competing tasks have (9% of all instructions on average).
- Mixes eembc1 and media2, have lower, yet significant, WCET increases (more than 2x and 3x respectively). This is caused by the combination of the two metrics just mentioned is lower than that of eembc2.
- Finally, media1 has a small WCET estimate increase due to a lower number of memory instructions executed by the main program (23%) and a lower percentage of stores in the challenger tasks (6% on average).

WB is the write policy with lower WCET estimate performance penalty. WB causes a small increase in WCET estimates even when we have 40% of data shared (higher than what is usually found in parallel applications [13, 15]). This is so, because only data requested by other cores is exchange via the bus.

HWP lies in between WT and WB, though it is much closer to WB. For eembc1 and media1, the WCET estimate is remarkably low. This is also contributed by low percentage of memory instructions combined with the low percentage of stores in the challenger tasks. For eembc2 and media2, HWP suffers high increase in WCET estimates in the 40% shared data scenario: despite HWP reduces the pressure on the bus, (i) the high percentage of share data, (ii) the high percentage of memory instructions these benchmark mixes execute (30% and 23% respectively), and (iii) the number of instructions that are stores in the competing



(a) EEMBC (b) Mediabench

Figure 7 Number of broadcasts and write-backs per memory access.

tasks (8 and 9% respectively), cause the pressure on the bus to increase. Yet, HWP stills performs better than WT, specially in high-core setups (4-8), where WT grows to 3-6x and HWP only grows to less than 2x in the worst setups. The penalty difference with WB as the number of cores and shared data increase is mainly due to the fact that all accesses to shared data is sent to the L2 (write-through on shared data), without the need of another core requesting the data. This means that the same core could write several times directly to L2 without another core requesting the data in between.

To sum up, HWP obtains similar WCET estimates to WB, but significantly smaller than WT (up to 5x) in multi-core setups. This difference in WCET estimates increases significantly with the number of cores being used.

5.4 Coherence

The write policy impacts the selection of the coherence solution. With WT caches a simple invalidation protocol V/I is enough, while for WB caches a more complex policy such as MESI is required. For HWP, an invalidation protocol such as the one used in WT is enough.

The potential impact of the coherence protocol, in particular MESI, is two-fold. First, the complexity of its design, implementation, and validation. And second, its impact on performance since the number of messages to exchange between processors and the L2 cache to maintain coherence.

Since the complexity has been qualitatively assessed in Sections 3 and 4, here we focus on the number of messages that will be sent in every coherence protocol as a proxy to coherence performance overheads. In particular we focus on the invalidation messages and the number of write-backs caused because of coherence (not due to cache capacity issues).

Figure 7 shows the average number of coherence messages per memory access for EEMBC (a) and Mediabench (b). We evaluate the 3 write policies: WT, WB and HWP; considering 5%, 10% and 20% of shared accesses in the last two policies. The number of invalidations in WT is high in both benchmark suites because every write access requires that an invalidation message is sent to the bus, since any other private cache can have a copy of the data. For WB and HWP the number of invalidations is much smaller, since the cache directory tracks the core having a copy of each cache line, and only shared data that actually is in private caches will be invalidated.

The other coherence metric we analyze is the number of write-backs related to coherence, which only happen for the WB policy. This occurs when a core c_0 modifies some data in its private dL1 and another core c_1 wants to access that data. Since the L2 knows that c_0 has this data in a Modified state, the L2 asks c_0 to write back the modified data to L2, and then the L2 sends it to c_1. In the WT policy, the L2 always has the most updated values, so there are no write-backs due to coherence. Likewise, HWP only writes back private data, treating shared data like WT, so there are no write-backs due to coherence either.

Note that while the trend for the coherence cost shown in this section is similar to that of energy (Figure 5), the absolute values are not the same. This is so because the energy cost of a write-back is higher than that of an invalidation. As a result, when comparing WT to WB/HWP, the energy consumed in coherence is not that high as the number of messages as shown in this section.

Overall, HWP offers the best of WT and WB in terms of coherence: it generates as little invalidations as WB without the coherence related write-backs of WB.

5.5 Reliability

We assume that caches are able to detect and correct single-bit upsets (SBU), while multi-bit upsets (MBU) may occur when their probability is high enough. We assume that solutions such as word interleaving[6] are applied so that a N-bit MBU becomes N SBUs. Hence, the criteria to assess reliability consists of whether designs are able to detect and correct single-bit errors. Note that such reliability criteria are already implemented in processors targeting the highest criticality levels in the space [11] and automotive [2] domains.

Since in WT all the updated data is always in L2, only parity is required in dL1 to detect single-bit errors given that correct data can be retrieved from L2. WB and HWP allow dirty data in the dL1 cache, and thus they require error correction capabilities in dL1, such as SEC-DED. The L2 cache always implements SEC-DED, since there can be dirty data at this cache level when using all policies.

The difference in the reliability technique used in dL1 has limited impact on area. Parity, used in WT, imposes a 1.6% increase in the number of cells needed (1 bit per 64-bit word), as well as few XOR gates and a comparator. SEC-DEC, used in WB and HWP, increases by 12.5% the number of cells (8 bits per 64-bit word), and also adds extra XOR gates and comparators [16]. Note that the relative area of dL1 cache w.r.t. L2 cache is typically low, and all write policies implement SEC-DED in L2, thus lowering the relative additional cost of SEC-DED vs parity in dL1 when put in the context of the complete cache system.

This section complements the comparison that has been made in Sections 3 and 4.3.

6 Related Work

Relevant related works relate to WB caches and their use in real-time systems, private/shared data classification mechanisms, models for computing WCET estimates for multi-core contention, and the use of other hybrid techniques for high-performance computing.

Due to the recent interest in the use of WB caches for critical real-time systems, mainly due to its potential increase in guaranteed performance, some works [34, 5] have studied static WCET analyses of this write policy, since it is more challenging than for WT caches. Authors in [34] propose an eviction-focused technique, analyzing for each cache miss if it could result in a write-back in order to estimate the WCET. In a more recent work [5], a new method has been proposed to complement the previous work by using a store-focused technique. This method consists in checking whether a store may transform a currently

[6] Interleaving K words at bit level ensures that bits of a given word, and hence protected with the same parity/ECC code, are at a distance of at least K bits.

clean line into dirty, and hence result in a write-back later on. Those techniques can be retargeted to capture HWP to tighten WCET estimates over WB/WT. In this line, previous works [31] also propose new cache systems that take into account shared/private data to improve WCET estimates, but with more radical changes required in the architecture.

Regarding private/shared data classification, different methods and hardware designs based on them have been proposed [13, 15]. Some authors [13] classify the different types of cache access patterns, and use such classification to implement a specific distributed cache design. Authors study the percentage of data that is private, shared/read-only and shared/modified. In [15] the authors propose a dynamic classification of shared and private pages. This technique needs some WB mechanism when a page changes its status from private to shared. While this technique may also improve performance over WT, it also has to deal with the coherence complexity of WB.

WCET estimate computation in multicores has been subject to intense study [23, 25, 7, 18, 8]. In [18, 8], the authors propose techniques for computing partially time composable Execution Time Bounds for bus accesses based on the number of requests the contenders can generate, regardless of when they access the bus. These technique provides tighter WCET estimates than simpler fully time composable models that always assume the worst case on a bus access. We have built on these techniques for WCET estimation.

Techniques for a hybrid approach on coherence management have been studied in high-performance domains [30, 6]. In [30], the authors implement a similar technique to [15] that dynamically changes the status of memory pages from private (default) to shared when they are accessed by more than one core. In [6], the authors propose a similar technique to differentiate private and shared pages at OS level, thus reducing the size of cache directories since they do not need to keep track of private lines. However, in these works there is still a non-trivial coherence mechanism with transient states, while our proposal targets a simpler (static) coherence mechanism.

7 Conclusions

The relentless trend towards the adoption of multilevel caches in real-time systems is a fact, in the line of high-performance systems. Our analysis of the write miss policy shows that WT simplifies coherence and reliability, while WB performs better in performance and energy. From the analysis we propose a new Hybrid Write Policy (HWP) that discriminates among shared and private data to smartly write through dL1 data or keep it dirty in dL1. Experimental results show that HWP results in remarkably better guaranteed performance than WT. HWP results for energy consumption per memory access improve those of WT. In terms of complexity of the coherence protocol, HWP implements a simple Valid/Invalid protocol like WT, compared to the complex MESI protocol used in WB.

References

1 ARM. ARM Cortex-M7 processor. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B_cortex_m7_trm.pdf`.

2 ARM. Arm cortex-r series processors specification. `http://infocenter.arm.com/help/topic/com.arm.doc.set.cortexr/index.html`.

3 ARM. ARM Cortex R5 technical reference manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0460d/DDI0460D_cortex_r5_r1p2_trm.pdf`.

4 ARM. ARM expects vehicle compute performance to increase 100x in next decade. `https://www.arm.com/about/newsroom/`

`arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php`, 2015.

5 T. Blaß, S. Hahn, and J. Reineke. Write-back caches in WCET analysis. In *ECRTS*, 2017.

6 B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *ISCA*, 2011.

7 D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, 2011.

8 E. Díaz, M. Fernández, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and F. J. Cazorla. MC2: Multicore and cache analaysis via deterministic and probability jitter bounding. In *ADA-Europe*, 2017.

9 G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *WCET Workshop*, 2014.

10 Cobham Gaisler. LEON4-N2X data sheet and user's manual. `http://www.gaisler.com/doc/LEON4-N2X-DS.pdf`.

11 Cobham Gaisler. NGMP preliminary datasheet version 2.1. `http://microelectronics.esa.int/gr740/LEON4-NGMP-DRAFT-2-1.pdf`.

12 Cobham Gaisler. UT699 32-bit fault-tolerant SPARC V8/LEON 3FT processor data sheet. `http://www.gaisler.com/doc/gr712rc-datasheet.pdf`.

13 N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 2009.

14 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009.

15 N. Ho, I. I. Ashraf, P. Kaufmann, and M. Platzner. Accurate private/shared classification of memory accesses: a run-time analysis system for the LEON3 multi-core processor. In *DATE*, 2017.

16 M. Y. Hsiao. A class of optimal minimum odd-weight-column SEC-DED Codes. In *IBM Journal of Research and Development*, 1970.

17 International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

18 J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Bounding resource contention interference in the next-generation microprocessor (NGMP). In *ERTS*, 2015.

19 H. Kim, Dionisio de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS*, 2014.

20 Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, 1997.

21 B. Lesage, D. Hardy, and I. Puaut. Shared data caches conflicts reduction for WCET computation in multi-core architectures. In *RTNS*, 2010.

22 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

23 T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald. Methods for modeling resource contention on simultaneous multithreading processors. In *IEEE ICCD*, 2005.

24 N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. In *HP Tech Report HPL-2009-85*, 2009.

25 J. Nowotsch, M. Paulitsch, D.B. Uhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

26 NXP. MPC8245 integrated processor hardware specifications. `https://www.nxp.com/docs/en/data-sheet/MPC8245EC.pdf`.

27 J. Poovey. *Characterization of the EEMBC Benchmark Suite*, 2007.

28 A. Roca, C. Hernandez, M. Lodde, and J. Flich. Area-efficient snoopy-aware NoC design for high-performance chip multiprocessor systems. In *Computers & Electrical Engineering*, 2015.

29 S. Rodrigo, J. Flich, J. Duato, and M. Hummel. Efficient unicast and multicast support for CMPs. In *MICRO*, 2008.

30 A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *PACT*, 2012.

31 M. Schoeberl. Time-predictable cache organization. In *STFSSD*, 2009.

32 Freescale Semiconductor. MPC8548E PowerQUICC III integrated processor hardware specifications. `http://cache.freescale.com/files/32bit/doc/data_sheet/MPC8548EEC.pdf`.

33 SoCLib. The soclib project. `http://www.soclib.fr/trac/dev`.

34 T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *RTSS*, 2010.

35 STMicroelectronics. STM32F756xx datasheet. `http://www.st.com/content/ccc/resource/technical/document/datasheet/fb/d4/56/db/60/61/4f/9c/DM00166114.pdf/files/DM00166114.pdf/jcr:content/translations/en.DM00166114.pdf`.

36 Texas Instruments. TMS570LS09x/07x 16/32-Bit RISC flash microcontroller. `http://www.ti.com/lit/ug/spnu607/spnu607.pdf`.

Compiler-based Extraction of Event Arrival Functions for Real-Time Systems Analysis

Dominic Oehlert
Hamburg University of Technology, Hamburg, Germany
dominic.oehlert@tuhh.de

Selma Saidi
Hamburg University of Technology, Hamburg, Germany
selma.saidi@tuhh.de

Heiko Falk
Hamburg University of Technology, Hamburg, Germany
heiko.falk@tuhh.de

Abstract

Event arrival functions are commonly required in real-time systems analysis. Yet, event arrival functions are often either modeled based on specifications or generated by using potentially unsafe captured traces. To overcome this shortcoming, we present a compiler-based approach to safely extract event arrival functions. The extraction takes place at the code-level considering a complete coverage of all possible paths in the program and resulting in a cycle accurate event arrival curve. In order to reduce the runtime overhead of the proposed algorithm, we extend our approach with an adjustable level of granularity always providing a safe approximation of the tightest possible event arrival curve. In an evaluation, we demonstrate that the required extraction time can be heavily reduced while maintaining a high precision.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Compilers, Mathematics of computing → Integer programming

Keywords and phrases compiler, real-time, event arrival functions, extraction

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.4

1 Introduction and Motivation

The design of safety-critical real-time systems often requires an effective analysis of the worst-case timing behavior in order to determine the compliance of the system to the timing constraints. This usually involves a traditional two-steps approach [3] which consists of a first low-level code analysis to determine the worst-case execution time of every task based on its program structure, followed by a system-level timing analysis to determine the worst-case response time of interfering tasks based on abstract models of the tasks activations and the scheduling policy.

In particular, system-level analysis often makes use of event arrival functions [18, 20, 23, 1] in order to bound the number of accesses to the shared resources and analyze the amount of induced interference. Event streams abstract the notion of traces and describe the possible I/O timing of interfering tasks sharing resources in the system under analysis. System properties are then computed in a compositional way using algebraic techniques where event streams are used to connect components' analyses according to the system's application and communication structure.

The code-level and system-level analysis steps are complementary, however in practice they are often considered separately. Some existing approaches, such as [19, 4] extend code-level analysis to system-level analysis by considering in a multicore system shared cache preemption delays to bound tasks' response times. These methods result in tight upper bounds on the response times. However, they consider a holistic approach for the evaluation of the worst-case execution time which cannot capture all the timing dependencies of interfering tasks. Furthermore, they are hard to scale with the complexity of the system and therefore they cannot be applied to complex hardware architectures involving on-chip interconnects and multi-level memory hierarchies.

On the other hand, system-level performance analysis approaches such as [23, 10] are more scalable and can be applied to analyze in a compositional way complex hardware structures [15]. They take as input for every task the worst-case execution time resulting from the code-level analysis, and *abstract models* of the arrival curves corresponding for instance to a known (periodic) activation pattern but which are very seldom derived using appropriate tools. This leads to harsh overapproximations in terms of timing, as the detailed event arrival curves are not known. Furthermore, the system-level results may even be unsafe due to unsafe event arrival curves resulting from e.g., traces which do not capture the worst-case behavior.

In this paper we present a compiler-based extraction of event arrival curves. Our goal is to bridge the gap between abstract system-level analysis and low-level code analysis. This is in particular very relevant for the analysis of multicore systems where there is a strong correlation between the individual timing of tasks and their cross-core interference [21]. Memory accesses constitute one main example where compared to existing approaches, such as [6, 24] that only consider a maximum total number of memory accesses for each program, arrival curves give a more precise information about the distribution of data accesses during the program execution. This allows to provide a more detailed and accurate analysis of the timing behavior of the system and to ease the integration between the worst-case execution and response time analysis steps. Yet, our proposed approach is not limited to memory accesses as it takes abstract events as an input, enabling various actions to be defined as an event (e.g., function calls).

Several existing work have investigated deriving access patterns by exploiting low-level informations. Li et al. [16] presented a mode-controlled data-flow model of real-time memory controllers. It is capable of deriving a tight worst-case bandwidth (WCBW) estimation for shared SDRAM memories. For this analysis, it is required to describe the dynamic command scheduling used by the memory controller and transaction sequence of the applications via so-called mode sequences.

Jacobs et al. [11] presented an approach for extracting safe upper event arrival curves at the code-level using compiler-based techniques. They proposed a modified version of the *implicit path enumeration technique* (IPET) [17] to find the maximum number of events potentially occurring in a given time interval on any path of the program. This approach is used to model all potential sub-paths implicitly by formulating an integer linear program (ILP). Yet, the presented approach lacks formalisms for critical aspects to ensure safeness (i.e., the resulting arrival curve should not be underapproximated) and tightness (i.e., the level of overapproximation due to the model should be minimal). The modification of the IPET approach is required since the standard approach only covers complete paths through a program. However, it is necessary to explore all possible sub-paths in a program starting and ending at any arbitrary node when deriving an event arrival curve. We extend this approach to also support lower event arrival curves and introduce a variable granularity during the extraction to find a compromise between extraction time and overapproximation.

Beside Jacobs et al. [11], only few existing work have considered IPET-based approaches exploring sub-paths in a program. Altmeyer et al. [2] presented an approach where sub-paths are defined by introducing additional preemption nodes. However, these sub-paths are restricted to the preemption nodes and are forced to terminate there which reduces the number of explored sub-paths. Kleinsorge et al. [14] presented an explicit path analysis which is capable of evaluating arbitrary partial worst-case execution paths. However, due to its nature all existing loops have to be unrolled during the analysis.

Contribution

We present a formal description of a compiler-based extraction of event arrival functions. It builds on the primary approach of Jacobs et al. and extends it in several aspects, e.g., the non-trivial extraction of lower arrival curves and increasing tightness of the arrival curves. The extraction of lower arrival curves is introduced since system-level analyses, such as Real-Time Calculus [23] or SymTA/S [10], partially rely on them as well. Tightness of the event arrival curves is increased by differentiation of loop control types and consideration of minimum loop bounds. For this, we introduce a complete formalized set of equations of the model. As the essential benefit of a compiler-based extraction of event arrival functions lies in its safe- and tightness, it is relevant to formulate the description well to ensure these characteristics. Besides, we provide an algorithm in order to derive a bound on the number of events for all possible time intervals of a program's runtime. The algorithm considers a complete coverage of all possible paths and therefore builds a safe upper bound on the number of events. The execution time of the proposed algorithm depends on the structure of the program but also on the granularity of the considered time intervals and the clustering of events per basic block. Therefore, we relate the extraction time of the proposed algorithm to the granularity and discuss the duality between the considered granularity level and the precision of the derived arrival curves.

The remainder of the paper is structured as follows. In Section 2 we present the system model and how the IPET approach is extended to extract the event arrival curves while providing a full coverage of all execution paths in a program. Section 3 presents our proposed algorithm for the extraction of the event arrival curves and its extension to consider different granularities. Section 4 evaluates our algorithm and confirms our findings. Section 5 concludes the paper.

2 System Model

2.1 Context and Prerequisites

Event arrival functions allow to model the dynamics of a real-time system, even for arbitrarily triggered events. They are generally defined as follows,

▶ **Definition 1** (Event Arrival Functions). Let $\eta_i^+(\Delta t)$ and $\eta_i^-(\Delta t)$ denote for each task i the maximum and minimum number of events issued within a time window of size Δt. Their pseudo-inverse counterparts $\delta^+(n)$ and $\delta^-(n)$, return the maximum/minimum time interval between the first and the last event in any sequence of n event arrivals. The conversion between η and δ functions is straightforward and can be easily derived as explained in [5].

In order to extract event arrival curves using code-level analysis, we consider as input the low-level representation of the program implementing a task annotated with loop bounds and timings. The low-level representation of a program is close to its actual assembly representation, yet still represented by certain data structures to ease the handling. Loop bounds are annotations which indicate the maximum or minimum number of possible iterations of a loop and can be inserted by the user or automatically. Prior to the extraction of the arrival curve, a *worst-case execution time* (WCET) analysis is performed considering no interference from other cores or tasks. The WCET of a program is the worst possible time it needs when it runs in isolation from its start until its termination. Subsequently, a *best-case execution time* (BCET) analysis is also performed. We are not discussing these analyses in further detail, since existing methods are used.

We denote events as actions triggered by an instruction or a sequence of instructions. Most notably this can be a memory access to a shared memory region or an access to an I/O device. However, the model is not restricted to this, since it solely takes as input the maximum and minimum of occurring events per basic block (BB).

2.2 Path Analysis and Event Arrival Functions

We base the extraction of the event arrival curves of a program on the control flow graph (CFG) extracted from its low-level representation. In order to determine the maximum (resp., minimum) number of events in a specific time interval Δt, all possible paths in this CFG have to be considered. Since the number of existing paths grows exponentially with the depth of conditional statements and variable loop bounds, considering all existing paths individually easily becomes infeasible. Jacobs et al. [11] proposed to exploit the so-called implicit path enumeration technique (IPET) as presented by Li and Malik [17]. This technique is typically used to locate the worst-case execution path (WCEP) of a program, over which its WCET occurs.

Using the IPET, a set of integer linear programming (ILP) *flow constraints* is generated to describe the CFG. All possible paths through the task's CFG are then implicitly described by the relation of its basic blocks in the constraints. By setting distinctive conditions, e.g., the first and last basic blocks have to be executed exactly once while maximizing the accumulated time, the WCEP can be found.

Yet, the classical IPET formulation can not be directly applied to the problem of finding maximum number of events during a given time window. This originates from the fact that we do not enforce one full path through the CFG, since we are only interested in sub-paths. Such a sub-path does not need to start at an entrypoint, nor end at an exit block. This way all possible sub-paths, which can be executed in a given time window, need to be considered.

Jacobs et al. introduced a modified IPET-based approach, in which all possible sub-paths are considered. Therefore any basic block can act as a source, whereas any reachable block can be a sink. This way any consecutive path, starting and ending at an arbitrary basic block, can be chosen by the ILP solver in order to find the sub-path over which the maximum number of events with respect to a given time interval are present.

In the sequel, we present the underlying basic model based on the previous work. A set of linear inequations is set up to describe the CFG of a program. The objective function is set to maximize the number of events on a to be chosen sub-path of the CFG, whereas the timing of this sub-path is not allowed to exceed a user-given constant.

For the upcoming we use the following notational conventions. Lower case italic Latin letters like a will be used for ILP variables. Upper case italic Latin letters like A represent constants inside the ILP model. Table 1 contains all ILP variables used in the paper. Unless otherwise stated, all ILP variables have a lower bound of 0. Lower case Latin letters as a subscript represent an index. Table 2 contains further miscellaneous symbols used.

Table 1 ILP decision variables.

Symbol	Description
a_i^+ (a_i^-)	Maximum (minimum) number of events contributed by basic block i on the sub-path
a_{Total}^+ (a_{Total}^-)	Maximum (minimum) number of events occurring along the sub-path
b_i	Reduction factor for basic block i if it is used as a starting and/or ending block
e_i	Basic block i is used as an end of the sub-path
f	Binary variable indicating if the chosen path covers a complete path through the program
g_ℓ	Number of flows at the loop entrance of loop ℓ
h_ℓ	Number of flows exiting the loop ℓ
n_ℓ^T	Maximum number of flows through the back edge of tail-controlled loop ℓ
n_ℓ^H	Maximum number of flows into the body of head-controlled loop ℓ
o_ℓ	Binary variable indicating the if the start of the sub-path was placed inside the loop ℓ
$p_{i,j}$	Total number of flows from basic block i to j
r_s (r_e)	Binary variable indicating if a timing reduction is applied at the starting (ending) block
$s_{i,j}$	Edge from basic block i to j is used as a starting edge
s_j	Any incoming edge at basic block j is used as starting edge
w_i^+ (w_i^-)	Total number of cycles contributed by basic block i on the sub-path when generating an upper (lower) arrival function

Table 2 Miscellaneous symbols.

Symbol	Description
A_i^+ (A_i^-)	Maximum (minimum) number of events of basic block i
$\mathcal{B}$	A set containing all basic blocks of the program
$\mathrm{B}_\ell^{\mathrm{Up}}$ $(\mathrm{B}_\ell^{\mathrm{Low}})$	The upper (lower) loop bound of loop ℓ
C_i^+ (C_i^-)	WCET (BCET) of basic block i
$\mathcal{C}_z$	A set containing all calling edges to the function z
$\mathcal{E}_\ell$	A set containing all entry basic blocks of loop ℓ
$\mathcal{E}_\ell^r$ $(\mathcal{E}_\ell^i)$	A set containing all regular (irregular) entry basic blocks of loop ℓ
$\mathcal{F}$	A set containing all functions of the program
$\mathcal{L}$	A set containing all loops of the program
$\mathcal{L}_T$ $(\mathcal{L}_H)$	A set containing all tail-controlled (head-controlled) loops
$\mathcal{M}_\ell$	A set containing all basic blocks belonging to loop ℓ
$\mathcal{N}_\ell$	A set containing all back edges of loop ℓ
$\mathcal{P}_i$	A set containing all direct predecessors of basic block i
$\mathcal{R}_z$	A set containing all possible return edges of the function z
$\mathcal{R}_{z,(i,j)}$	A set containing all possible return edges of the function z when called using edge (i,j)
$\mathcal{S}_i$	A set containing all direct successors of basic block i
T_ℓ	Equals 1 if loop ℓ is tail-controlled, otherwise 0
$\mathcal{X}_\ell$	A set containing all exit basic blocks of loop ℓ

As mentioned previously, a WCET (resp., BCET) analysis is first executed where all accesses to a shared memory are assumed with a minimal (resp., maximum) latency. Additionally, variable timings (which may be influenced by the event-type under focus or caches) have to be considered carefully, such that they do not thwart a safe WCET (resp., BCET) estimation. Note that, the system is evaluated in isolation, without considering interference from other cores. We consider the timing of a basic block in terms of cycles. Therefore integer variables are suitable to represent the timing of a basic block. Subsequently, the CFG is synthetically modified, such that every basic block has a successor and predecessor. These additional blocks are not inserted into the actual program code and are only present in our analysis. A virtual source $\ominus$ is created for the entrypoint and inserted as a predecessor to the first basic block. In the same fashion, virtual sinks $\perp$ are created for all possible exits and inserted as a successor to the last basic blocks. Therefore, for every basic block i in the CFG, a flow constraint is generated as follows,

$$\sum_{j\in\mathcal{P}_i} p_{j,i} - e_i = \sum_{k\in\mathcal{S}_i} (p_{i,k} - s_{i,k}) \tag{1}$$

The integer variable $p_{i,k}$ describes the number of times the control flow (subsequently simply called *flow*) enters basic block k from basic block i. Each input flow of a basic block represents one execution of the basic block. The variable e_i is bound to a binary value and is set to 1 when the basic block i is used as the last basic block in the chosen sub-path. It represents a "movable" sink. In a similar manner, the variable $s_{i,k}$ is bound to a binary value and is set to 1 when the basic block k is the first basic block in the chosen sub-path. In particular, the edge from basic block i to basic block k is used as the initial flow. The set $\mathcal{P}_i$ contains all directly preceding basic blocks of i. Similarly, the set $\mathcal{S}_i$ contains all directly succeeding basic blocks of i. This way, Equation (1) functions as a node law, assuring that the amount of flow into a node is equal to the amount of flow leaving it. Additionally, one initial flow can be inserted into a basic block without violating the constraint. In the same fashion, a path can end at a particular node if the corresponding e variable is set to 1.

The flows originating from (resp., directed to) the virtual sources (resp., sinks) are defined as follows,

$$p_{\ominus,i} = s_{\ominus,i} \tag{2}$$

$$p_{i,\perp} = 0 \tag{3}$$

Since only one consecutive path is allowed, the sum over all starting (ending) points is limited to be smaller or equal to one. Additionally, if a starting point is existing, there has to be an ending point as well:

$$\sum_{i\in\mathcal{B}} \sum_{j\in\mathcal{P}_i} s_{j,i} = \sum_{i\in\mathcal{B}} e_i \leq 1 \tag{4}$$

where, $\mathcal{B}$ is the set holding all basic blocks of the current task.

The ILP variable s_i is set to 1 if any of the ingoing edges of basic block i is used as an initial flow. It is defined as follows:

$$s_i = \bigvee_{j\in\mathcal{P}_i} s_{j,i} \tag{5}$$

Logical operators like $\vee$ or $\wedge$ can be easily described inside the ILP formulation as shown by, e.g., Johannes [12].

We assume that all instructions which may cause an event are known. It is possible to perform a value analysis for this purpose, although potential over- or underapproximations due to unknown values should be handled carefully. Architectures featuring out-of-order execution need a particularly careful micro-architectural analysis, as instruction order may change during the execution. We will not discuss these issues in detail since they exceed the scope of this paper. We define the maximum number of events per basic block i as A_i^+. This is used to calculate the amount of events happening on the chosen sub-path.

$$a_i^+ = A_i^+ \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \tag{6}$$

$$a_{\text{Total}}^+ = \sum_{i \in \mathcal{B}} a_i^+ \tag{7}$$

The ILP variable a_i^+ represents the maximum accumulated number of events of basic block i over all its executions which are part of the chosen sub-path. a_{Total}^+ defines the maximum number of events existing on the chosen sub-path.

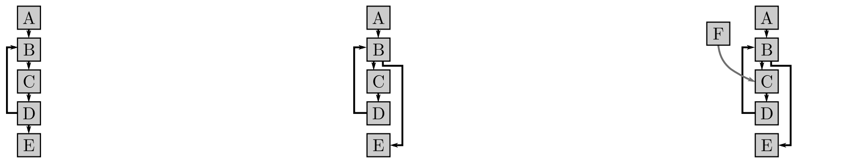
Besides the control flow and the events, also the timing has to be considered. We define w_i^+ as the number of cycles which basic block i contributes on the chosen sub-path.

$$w_i^+ = \left(C_i^- \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - (C_i^- - 1) \cdot b_i \tag{8}$$

C_i^- is the BCET of the basic block i. The BCET is chosen instead of WCET here, since we are interested in the maximum amount of events in a given time interval. Hence, using the WCET would be too optimistic, as the accumulated time over the sub-path may require less time. The ILP variable b_i is bound to an integer value between [0,2] and is defined as follows:

$$b_i = \begin{cases} 0 & \text{if } s_i = e_i = 0, \\ 2 & \text{if } s_i \wedge e_i \wedge \left(\sum_{j \in \mathcal{P}_i} p_{j,i} > 1 \right), \\ 1 & \text{else.} \end{cases} \tag{9}$$

The variable b_i functions as a reduction factor to the timing contribution of basic block i. As it is not considered at which particular location inside the basic block its events are triggered, the first and last basic block of the sub-path need to be handled with special care in order to be safe: Since a sub-path through the program can in fact start (or end) at a specific instruction inside the basic block, assuming its full BCET for this case would be too pessimistic. For this particular case we assume that all events at this bounding block happen at the very last (or first if the ending block) cycle of the basic block. In case the chosen sub-path does neither start nor end at basic block i, the accumulated timing w_i^+ is not reduced, as the reduction factor b_i is set to 0. If basic block i is chosen as the start and end of the sub-path (and it does not solely consist of the basic block i), the reduction factor b_i is set to 2. Thereby the timing contribution of basic block i is reduced by $2 \cdot (C_i^- - 1)$. Finally, if the basic block i is chosen as the start or end block (or the sub-path only consists of BB i), the reduction factor is set to 1 as a safe overapproximation. We show in Section 3 a simplistic approach to increase the granularity to a single-event level with a minor preparation of the control flow graph to reduce the introduced pessimism.



(a) Tail-controlled. **(b)** Head-controlled. **(c)** Head-controlled (irr.).

Figure 1 Sample loop structures.

Finally, the sum of all timing contributions is limited to be smaller or equal to the chosen time interval Δt, while maximizing the number of events.

$$\Delta t \geq \sum_{i \in \mathcal{B}} w_i^+ \tag{10}$$

$$\max : a_{\text{Total}}^+ \tag{11}$$

2.3 Handling Loops and Function Calls

So far, the model does not limit loop iterations. It is assumed that all loops are annotated with loop bounds. The deriving of loop bounds or control type (head- or tail-controlled) is beyond the scope of this paper and well researched [22, 25]. The previous work by Jacobs et al. [11] covers the handling of loops only very briefly. It is stated that the original IPET formulation has to be extended for the case that a path is starting inside a loop, where the loop's back edge may be taken an additional loop bound-times. Yet, no formal description is given. In the sequel, we introduce a tight and accurate description of handling loops. Besides, we introduce how function calls can be handled which the previous work [11] lacks of.

We differentiate between head- and tail-controlled loops. For tail-controlled loops we limit the number of back edges taken:

$$\forall \ell \in \mathcal{L}_T : \sum_{(i,j) \in \mathcal{N}_\ell} p_{i,j} \leq n_\ell^T \tag{12}$$

$\mathcal{L}_T$ defines the set of all tail-controlled loops. The set $\mathcal{N}_\ell$ contains all back edges of the loop ℓ. A back edge of a loop originates from the loop tail to its head. In the exemplary loop in Figure 1a this is the edge from basic block D to B. The ILP variable n_ℓ^T denotes the maximum flow through all back edges of the loop ℓ.

$$n_\ell^T = (\mathrm{B}_\ell^{\mathrm{Up}} - 1) \cdot \left(\sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} + o_\ell \right) \tag{13}$$

$\mathrm{B}_\ell^{\mathrm{Up}}$ is defined as the upper loop bound of loop ℓ. The upper loop bound of a loop defines the maximum number of loop body iterations. The set $\mathcal{E}_\ell$ contains all basic blocks, which are entrances of loop ℓ, while $\mathcal{M}_\ell$ contains all members of this loop (including nested loop members). We define an entrance block of a loop as a basic block which belongs to the loop and has a predecessor which is not part of the loop. In the exemplary loop in Figure 1a basic block B is the entrance block. This implies that $p_{j,i}$ in Equation (13) covers all edges which are entering the loop from outside, which would be the edge $p_{A,B}$ in the sample loop.

The binary ILP variable o_ℓ is forced to 1 in case any of the basic blocks inside the loop is chosen as a starting point and is defined as follows:

$$o_\ell = \sum_{i \in \mathcal{M}_\ell} \sum_{j \in (\mathcal{P}_i \cap \mathcal{M}_\ell)} s_{j,i} \tag{14}$$

Thereby, Equation (13) permits the loop body to be executed $\mathrm{B}_\ell^{\mathrm{Up}}$ times for every time the loop is entered. Furthermore, if the starting point is chosen inside the loop, the loop body can be executed $\mathrm{B}_\ell^{\mathrm{Up}}$ times additionally. This is required, as the starting block can also be chosen inside a loop.

The constraints handling head-controlled loops are very similar, yet with a few modifications in order to tighten the resulting number of events. In contrast to the tail-controlled loops, for head-controlled loops we limit the number of times the loop is actually entered. Otherwise, one additional loop execution more than feasible by the CFG would be permitted.

Example: Assume the head-controlled loop in Figure 1b has an upper loop bound of 1. If the starting point is chosen at, e.g., basic block B, the number of executed back edges would be restricted to 1, since there is no flow entering the loop. Yet, without violating the constraints, the loop body could be executed twice according to the model (sequence $\{B, C, D, B, C, D\}$), since the back edge is only executed once. Especially in case of nested loops, an overapproximation of a single loop iteration can lead to a significant overapproximation of total number of events. We therefore introduce the following equations, which limit the number of times a head-controlled loop is entered.

$$\forall \ell \in \mathcal{L}_H : \sum_{i \in \mathcal{E}_\ell^r} \sum_{j \in (\mathcal{S}_i \cap \mathcal{M}_\ell)} p_{i,j} \leq n_\ell^H \tag{15}$$

The set $\mathcal{L}_H$ contains all head-controlled loops, while the set $\mathcal{E}_\ell^r$ contains the *regular* entrance block of the loop ℓ ($|\mathcal{E}_\ell^r| = 1$). The exemplary loop in Figure 1b only has a regular entry, while the exemplary loop in Figure 1c has two entries: One regular entry (B), and one *irregular* entry (C) (irregular entries arise due to, e.g., goto-statements into loops at the source code level). Equation (15) restricts the number of times the loop body is entered via its regular entry to a maximum of n_ℓ^H. Regarding the exemplary loop in Figure 1c this represents the edge from B to C. This limit is defined as follows:

$$n_\ell^H = \mathrm{B}_\ell^{\mathrm{Up}} \cdot \left(\sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} + o_\ell \right) - o_\ell - \sum_{i \in \mathcal{E}_\ell^i} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} \tag{16}$$

Similar to tail-controlled loops, Equation (16) permits the loop body to be executed $\mathrm{B}_\ell^{\mathrm{Up}}$ times for every flow entering the loop. In case the starting point is chosen inside the loop (o_ℓ=1), the loop body can be entered an additional ($\mathrm{B}_\ell^{\mathrm{Up}}$-1) times. The deduction of 1 stems from the fact that if the starting point is chosen inside the loop, the loop is already entered once. The right-hand subtractive term is required for irregular loops. If the loop is entered via an irregular entry, the first loop iteration clearly does not include one entry from the regular entry into the loop body. As we limit the loop iterations via the number of times the loop body is entered via its regular entry, the upper limit n_ℓ^H has to be lowered for each time the loop is entered via an irregular entry.

In order to tighten the results, the ILP model also considers the minimum loop iterations.

$$\forall \ell \in \mathcal{L} : g_\ell = \sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} \tag{17}$$

$$h_\ell = \sum_{i \in \mathcal{X}_\ell} \sum_{j \in (\mathcal{S}_i \setminus \mathcal{M}_\ell)} p_{i,j} \tag{18}$$

$$\sum_{(i,j) \in \mathcal{N}_\ell} p_{i,j} \geq \min\left(g_\ell, h_\ell\right) \cdot \left(\mathrm{B}_\ell^{\mathrm{Low}} - T_\ell\right) \tag{19}$$

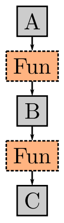


Figure 2 Exemplary CFG.

The set $\mathcal{L}$ contains all loops, whereas set $\mathcal{X}_\ell$ contains all exit blocks of the loop ℓ. Equations (17) and (18) are solely present for a better readability. Equation (17) defines the number of flows arriving at the loop head, while (18) defines the flows exiting the loop. Equation (19) sets a minimum number of loop iterations for each time the loop is entered and exited. $\mathrm{B}_\ell^{\mathrm{Low}}$ is the lower loop bound of loop ℓ, whereas T_ℓ equals 1 if ℓ is tail-controlled and 0 otherwise. The min()-Function in Equation (19) is described in the ILP as shown by Oehlert et al. [19].

Beside loops, our model is also capable of modeling function calls. It is sensitive to call edges and their corresponding return edges, i.e., for each calling edge, all valid return edges are evaluated. Calling contexts are currently not supported. To ensure tightness, we restrict the difference between ingoing and outgoing flows of functions. Since the start or end block may be chosen inside a called function, the in- and outgoing flows of a function may differ.

$$\forall \gamma \in \mathcal{F} : \forall (i,j) \in \mathcal{C}_\gamma : p_{i,j} \geq \min\left(\sum_{(m,n)\in\mathcal{R}_{\gamma,(i,j)}} p_{m,n}, \sum_{(x,y)\in\mathcal{C}_\gamma} p_{x,y} \right) - s_\gamma \tag{20}$$

$$\sum_{(m,n)\in\mathcal{R}_{\gamma,(i,j)}} p_{m,n} \geq \min\left(p_{i,j}, \sum_{(x,y)\in\mathcal{R}_\gamma} p_{x,y} \right) - e_\gamma \tag{21}$$

The set $\mathcal{F}$ consists of all functions inside the program, while $\mathcal{C}_\gamma$ contains all calling edges to the function γ. The set $\mathcal{R}_\gamma$ contains all possible return edges from the function γ. Furthermore, the set $\mathcal{R}_{\gamma,(i,j)}$ contains all possible return edges from the function γ when called via the edge (i,j) ($\mathcal{R}_{\gamma,(i,j)} \subseteq \mathcal{R}_\gamma$). s_γ is set to 1 if any basic block of function γ or a basic block contained by a function called by γ is used a starting block. e_γ is the corresponding counterpart for the ending points. Equation (20) sets up one constraint for every call inside the program. It sets a lower bound for the number of times the calling edge (i,j) is executed. Therefore, the minimum of flows entering the function γ and exiting via a return-edge belonging to the caller-edge (i,j) is determined. Equation (21) then sets a lower bound on the number of times a corresponding return-edge is executed. More generally speaking, the equations enforce that only call- and return-edges which belong together are allowed to be used. *Example:* Figure 2 depicts an exemplary CFG with 2 calls. The set of constraints modeling the function *Fun* contains two incoming edges, one from basic block A and one from B, as well as two corresponding exiting edges. Obviously, only the CFG-feasible paths $A \to \mathit{Fun} \to B$ and $B \to \mathit{Fun} \to C$ should be allowed, yet paths like $A \to \mathit{Fun} \to C$ not. The original IPET formulation can easily ensure this by forcing a calling edge's number of executions to be equal to the executions of its feasible return edges. As in our case sub-paths may also start (end) in a called function, the number of calls and returns may differ. Therefore the lower bound of a call-edge (return-edge) is decreased by one in case the starting (ending) point is chosen inside the called function. As recursive functions may be exited (called) multiple times without being called (exited), this difference can also be greater than 1 (e.g., the sub-path is chosen to start in the deepest recursion level). Therefore the min()-function is used, such that minimum of overall executed calling edges and dedicated returns is evaluated in Equation (20), whereas Equation (21) handles the return-edges likewise.

This differentiation is done on one hand to ensure tightness (dedicated caller-return pairs) and on the other to enable starting and ending points to be chosen inside called functions.

2.4 Lower Bound on the Event Arrival Function

The previous approach by Jacobs et al. only focused on the extraction of an upper event arrival curve. In this section we present how lower event arrival curves can be extracted.

A lower bound on the event arrival function $\eta_i^-(\Delta t)$ can be similarly derived using the introduced ILP model, yet with several modifications and additions. Since we want to determine the minimum amount of events in a given time window, we use the WCET of a basic block instead of the BCET used for the upper bound. Therefore Equation (8) is replaced with the following one:

$$w_i^- = \left(C_i^+ \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - b_i \cdot ((s_i \wedge r_s) \vee (e_i \wedge r_e)) \tag{22}$$

Instead of the BCET C_i^- of a basic block i, its WCET C_i^+ is used. In order to derive a safe lower event arrival curve, C_i^+ has to include all potential interferences, stalls or similar. If C_i^+ is depending on the event-type under focus, it is possible to derive an upper event arrival curve up-front and use system analysis tools [5, 23] to determine a safe WCET. In case the basic block i is used as a starting and/or ending block and the corresponding binary variable r is set to 1, the block's timing is reduced by b_i (c.f. Equation (9)). This again is done as a safe overapproximation, since we are not considering at which particular locations the event triggering instructions are located in a basic block. Although the multiplication term does not appear to be linear, it can be expressed using a simple case-structure since $((s_i \wedge r_s) \vee (e_i \wedge r_e))$ is restricted to Boolean values. In a similar manner Equation (6) is replaced:

$$a_i^- = \left(A_i^- \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - b_i \cdot ((s_i \wedge r_s) \vee (e_i \wedge r_e)) \cdot A_i^- \tag{23}$$

A_i^- represents the minimum number of events in basic block i. The first term remains the same while a second subtractive term is introduced. Similar to Equation (22), in case basic block i is the start and/or end block of the path, its number of events can be reduced by $b_i \cdot A_i^-$. The modifications of (23) is done since we do not account for the location of events inside the basic blocks, similarly as in Equation (22). By this overapproximation we assume, that all events happen at the very first cycle (very last cycle) of a starting (ending) node. Therefore, if r_s (or respectively r_e for an end block) is set to 1, a basic block's timing is reduced and $b_i \cdot A_i^-$ events are subtracted. The variables r_s and r_e are used in order to apply a safe overapproximation for the first and last basic block of a sub-path, yet still cover all occurring events when a full path through the program is found.

We insert additional constraints to detect the case that a complete path through the program (starting at the entrypoint and ending at a sink) is chosen.

$$s_\ominus = s_{\ominus,j} \tag{24}$$

$$e_\perp = \bigvee_{i \in \mathcal{T}} e_i \tag{25}$$

$$f = s_\ominus \wedge e_\perp \tag{26}$$

In Equation (24) the basic block j is the entry basic block of the program (c.f. block A in Figure 4). The set $\mathcal{T}$ contains all possible exiting basic blocks. Therefore f is set to 1 in case the chosen path starts at the entrypoint and ends at an exiting block, resulting in a complete path through the program.

Finally, Equations (10) and (11) are replaced by the following two:

$$\Delta t \leq \left(\sum_{i \in \mathcal{B}} w_i^- \right) + \left(f \wedge \left(\overline{r_s \vee r_e}\right)\right) \cdot M \tag{27}$$

$$\min : a_{\text{Total}}^- \tag{28}$$

Most notably the direction of the comparison operator in Equation (27) is flipped and the objective is changed to minimize. Again, Δt is given as a constant, representing the time interval for which the minimum number of events should be determined. Therefore the solver is forced to find a (sub-)path in the CFG which takes at least Δt cycles and the minimum amount of events. M is a sufficiently large constant. A trivial sufficient value is the WCET of the analyzed program.

In case a complete program path is covered and no reductions in terms of cycles and events are applied, Equation (27) is always satisfied. Therefore the arrival function converges at a complete path with the minimum number of total events.

3 Event Arrival Extraction Over All Existing Paths

In the following, we present how event arrival curves can be obtained with an adjustable level of precision while still resulting in a safe overapproximation. This subject is not part of the scope of the previous work [11].

3.1 Extraction Algorithms

As described previously, in order to derive an upper (resp., lower) bound on the event arrival curves by a given task, we need to explore different time intervals and extract for each duration the maximum (resp., minimum) number of events during this interval. For this, the IPET approach is customized to consider all sub-paths of duration Δt and maximing (resp., minimizing) the number of events. This procedure has to be repeated multiple times to cover all possible values of Δt. In the following we present two algorithms to explore the space of all possible values of time intervals.

Algorithm 1 is used to generate an arrival curve with an adjustable level of *time granularity* I . Here the value of Δt is bound to increasing values with a fixed increment I while solving the ILP for every value of Δt. The WCET of the program is used as an upper bound, since by definition no path can result in a higher timing than the WCEP. Note that, the smaller the value of I, the more fine-grained the generated arrival curve is. This comes with a linear increase in the number of ILP variants to be solved, one for every possible new value of Δt.

Algorithm 1 Fixed granularity extraction.

Input: I - Time granularity
Output: m - Map with the max. number of arrivals with Δt as a key

```
Map $m$
for ($\Delta t$=0; $\Delta t \leq$ WCET; $\Delta t$ += $I$) do
    $m$[ $\Delta t$ ] = solveILP($\Delta t$)
end for
```

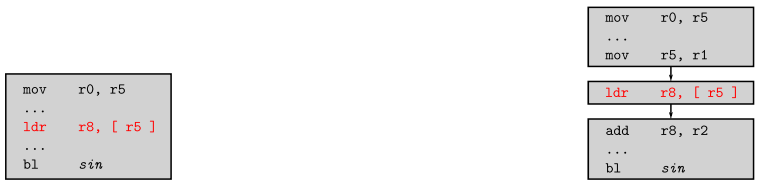
Algorithm 2 Binary search.

Input: -
Output: m - Map with the max. number of arrivals with Δt as a key

```
Map $m$, List $w$ // $w$ contains all windows to be analyzed
$w$.push({ 0, WCET } )
while !($w$.empty()) do
    Pair $curWindow = w$.pop()
    if !($m[curWindow$.lower] exists) then
        $\Delta t = curWindow$.lower
        $m[curWindow$.lower] = solveILP($\Delta t$)
    end if
    if !($m[curWindow$.upper] exists) then
        $\Delta t = curWindow$.upper
        $m[curWindow$.upper] = solveILP($\Delta t$)
    end if
    if $m[curWindow$.lower] $\neq$ $m[curWindow$.upper] and ($curWindow$.upper - $curWindow$.lower ) $> 1$ then
        x = $\lfloor (curWindow.\text{lower} + curWindow.\text{upper})/2 \rfloor$
        $w$.push($\{curWindow.\text{lower}, x\}$)
        $w$.push($\{x, curWindow.\text{upper}\}$)
    end if
end while
```

It is noteworthy that this approach still results in a safe (overapproximated) arrival curve where a coarse-grain arrival curve always dominates a fine-grain arrival curve. A further discussion will be presented in the evaluation Section 4.

While this approach is reasonable for a limited amount of sample points over the arrival curve, it is not applicable for generating an arrival curve covering all potential intervals (i.e $I = 1$). For this circumstance, we present in Algorithm 2 another procedure based on a binary search. We exploit two facts regarding the event arrival curves: i) they are monotonically increasing, ii) they are piecewise step functions (i.e., we will not necessarily have for instance a memory access at every cycle of execution). Therefore, for a given interval of Δt, we first examine the maximum number of events at the outer boundaries of the interval. If this number is equal at the boundaries, then no new event has occurred during this interval and thereby no further analysis is required inside the current time interval, since all intermediate values will result in the same maximum (resp., minimum) number of events at the interval boundaries. Otherwise, the interval (initially set to [0,WCET]) is split in half and the procedure is further repeated until all intervals in the curve are covered. Note that, both algorithms can be used to generate either an upper event arrival curve or a lower event arrival curve.



```
mov    r0, r5
...
ldr    r8, [ r5 ]
...
bl     sin
```

(a)

```
mov    r0, r5
...
mov    r5, r1
```

```
ldr    r8, [ r5 ]
```

```
add    r8, r2
...
bl     sin
```

(b)

Figure 3 A sample basic block before and after splitting.

3.2 Refining the Basic Block Granularity

The number of events is analyzed on a basic block level. In case a basic block i has A_i events, this number is accounted for the whole block, leading to another overapproximation since we do not consider where these events are located during the execution of the basic block. In order to refine the level of granularity we partially re-structure the basic blocks which contain potential event triggering instructions. These basic blocks are transformed into multiple "sub basic blocks" as shown in [19]. Therefore, all basic blocks containing instructions which potentially trigger an event are split up into so-called sub basic blocks to isolate the event. Such sub basic blocks solely consist of the event's single instruction.

Consider the example depicted in Figure 3. After refining the granularity the basic block is split up into 3 sub basic blocks, where the second sub basic block only consists of the potentially data accessing instruction. This is shown in Figure 3b. This technique can be applied prior to the ILP generation. Besides, the ILP model with the refined sub basic blocks can be set up using the same constraints as presented.

Therefore, combining this refining technique and the presented extraction algorithms, the granularity can be adjusted at two levels: 1) Calculating a fixed number of sample points versus a complete curve coverage. 2) Considering a clustered number of events per basic block versus isolating each event in a separate sub basic block.

4 Evaluation

All experiments are performed on an Intel Xeon Server (20 cores at 2.3 GHz, 94 GB RAM) and the ILPs were solved using Gurobi 7.5.0. For evaluation purposes the MRTC benchmark suite [9] with annotated loop bounds from the TACLeBench project [7] are used. All benchmarks are compiled with the WCET-aware C compiler (WCC) [8] and the `-O2` flag activated which enables several ACET-oriented optimizations. As an exemplary evaluation platform the ARM7TDMI architecture (without caches) is chosen. Timing analyses are performed using methods described by Kelter [13]. The benchmark `duff` is excluded from the evaluation, as it is not supported by the currently used timing analysis tool.

For all our experiments, we focus on extracting event-arrival functions for *data accesses*. We therefore assume each access of a data object to generate an event.

4.1 An Illustrative Example

In the following, we illustrate the approach considering the control flow graph example depicted in Figure 4. We show how to derive the arrival curve $\eta^+(\Delta t)$ representing an upper bound on the number of data accesses. For this, each basic block i is annotated with its execution time C_i (for this particular example we assume that the BCET of a basic block is equal to its WCET) and its number of events A_i. Note that we do not consider any

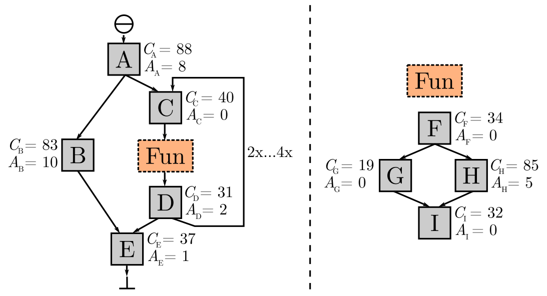


Figure 4 Sample control flow graph.

distribution of events inside a basic block. The considered example contains a tail-controlled loop with a minimum and maximum number of executed back edges of 2 and 4 and therefore loop bounds of [3,5].

We first derive the ILP model of the given CFG example. A virtual source is inserted as a predecessor of the task's entrypoint basic block A and a virtual sink as a successor to its exiting basic block E. We start by setting up the node equation for the basic block A (cf. Equations (1) - (3)).

$$s_{\ominus,A} - e_A = p_{A,B} - s_{A,B} + p_{A,C} - s_{A,C} \tag{29}$$

We continue with the rest of the basic blocks of our main function.

$$p_{A,B} - e_B = p_{B,E} - s_{B,E} \tag{30}$$
$$p_{A,C} + p_{D,C} - e_C = p_{C,F} - s_{C,F} \tag{31}$$
$$p_{I,D} - e_D = p_{D,E} - s_{D,E} + p_{D,C} - s_{D,C} \tag{32}$$
$$p_{B,E} + p_{D,E} - e_E = 0 \tag{33}$$

As all node constraints for the main functions are set up, additional node constraints for the function `fun` are inserted in the same fashion.

$$p_{C,F} - e_F = p_{F,G} - s_{F,G} + p_{F,H} - s_{F,H} \tag{34}$$
$$p_{F,G} - e_G = p_{G,I} - s_{G,I} \tag{35}$$
$$p_{F,H} - e_H = p_{H,I} - s_{H,I} \tag{36}$$
$$p_{G,I} + p_{H,I} - e_I = p_{I,D} - s_{I,D} \tag{37}$$

After all basic node constraints have been inserted, additional constraints concerning the loop are also inserted (cf. Equations (12)-(14)).

$$p_{D,C} \leq n_{L1}^T \tag{38}$$
$$n_{L1}^T = 4 \cdot (p_{A,C} + o_{L1}) \tag{39}$$
$$o_{L1} = s_{D,C} + s_{C,F} + s_{F,G} + s_{F,H} + s_{G,I} + s_{H,I} + s_{I,D} \tag{40}$$

As shown in Figure 4, the loop is tail-controlled. Therefore Equation (38) limits the number of back edges executed to a maximum of n_{L1}^T. In case a chosen path starts inside the loop body o_{L1} is set to 1. Since the loop is not nested, $p_{A,C}$ can be at most 1, which bounds the number of back edges executed to be at most 4 in any case. In case the loop would be nested, for each flow entering the loop an additional 4 flows through the back edge would be permitted.

In order to tighten the number of possible events, we also consider the minimum number of loop iterations (cf. Equations (17)-(19)).

$$p_{D,C} \geq \min(p_{A,C}, p_{D,E}) \cdot 2 \tag{41}$$

Equation (41) sets a minimum number of loop iterations in case the chosen path enters and exits the loop.

Furthermore, we restrict the number of in- and outgoing flows of the function `Fun` (cf. Equations (20),(21)):

$$p_{C,F} \geq \min\left(p_{I,D}, p_{C,F}\right) - s_{\texttt{Fun}} \tag{42}$$

$$p_{I,D} \geq \min\left(p_{C,F}, p_{I,D}\right) - e_{\texttt{Fun}} \tag{43}$$

Anyhow, since `Fun` is only called by one location and not recursive, Equations (42) and (43) can be omitted in this case.

Subsequently the constraints concerning the events are inserted (cf. Equations (6), (7)).

$$a_A^+ = 8 \cdot s_{\ominus,A} \tag{44}$$

$$a_B^+ = 10 \cdot p_{A,B} \tag{45}$$

$$\ldots$$

$$a_I^+ = 0 \tag{46}$$

$$a_{\text{Total}}^+ = a_A^+ + a_B^+ + \ldots + a_I^+ \tag{47}$$

As the timing contribution of a basic block is dependent on the subtracting factor b_i (cf. Equation (9)), the corresponding constraints are inserted:

$$b_A = \begin{cases} 0 & \text{if } s_A = e_A = 0, \\ 2 & \text{if } s_A \wedge e_A \wedge \left(p_{\ominus,A} > 1\right), \\ 1 & \text{else.} \end{cases} \tag{48}$$

$$\ldots$$

Finally, the timing constraints are added (cf. Equations (8)-(10)).

$$w_A^+ = 88 \cdot s_{\ominus,A} - 87 \cdot b_A \tag{49}$$

$$\ldots$$

$$w_I^+ = 32 \cdot (p_{G,I} + p_{H,I}) - 31 \cdot b_I \tag{50}$$

$$\Delta t \geq w_A^+ + w_B^+ + \ldots + w_I^+ \tag{51}$$

With Δt being a constant, representing the length of the current interval.

The resulting lower and upper arrival function are depicted in Figure 5 where the granularity of Δt for the algorithm of extraction was set to 1 cycle. In the following, we detail the results of the arrival curve $\eta^+(\Delta t)$. The very first step appears at Δt=1 to 10 events (basic block B). The subsequent second step to 18 events occurs at Δt=2, happening on the path from basic block A to B. At Δt=85 a step to 19 events occurs which happens on the path $\{A, B, E\}$. The next step to 20 events is at Δt=298. This is occurring on the path $\{A, C, F, H, I, D, C, F, H\}$. The next step up to 22 events happens at Δt=415, where the previous path is extended to include basic blocks I and D (forming two complete loop iterations). At Δt=520 the maximum number of events increases to 27, including additional executions of basic blocks C, F and H. Note that there is no intermediate step to 23 events via the loop exiting path $\{D, E\}$ due to the lower loop bound of 3. From this point on the arrival curve follows a repetitive pattern. The arrival curve converges at Δt=1112 with 44 events, which covers the whole right side of the CFG from Figure 4 with the maximum amount of loop iterations.

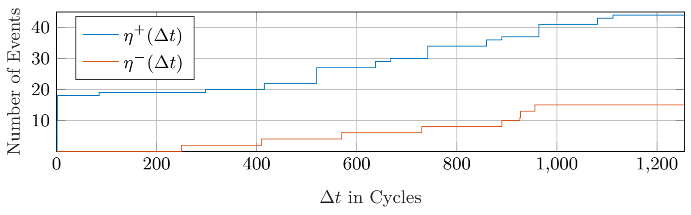


Figure 5 Extracted event arrival curves for CFG in Figure 4.

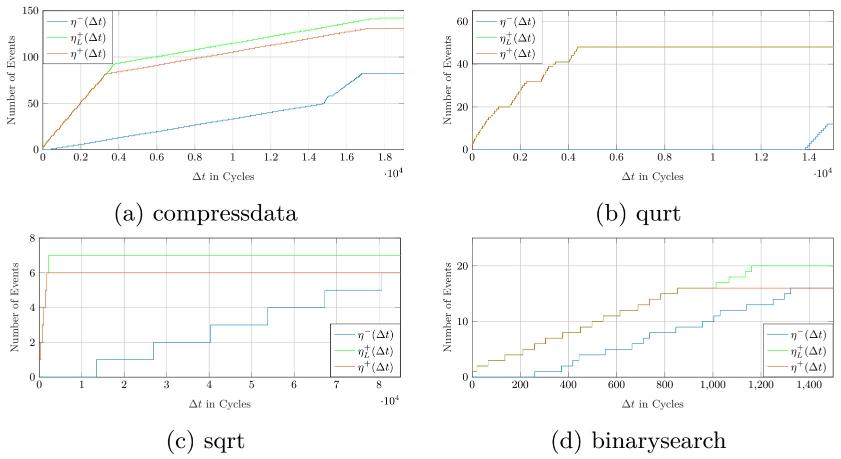


Figure 6 Extracted event arrival curves for benchmarks (refined BBs, binary search).

4.2 Benchmarks Evaluation

In the following, we present the event arrival curves of 4 selected benchmarks from the MRTC benchmark suite [9]. The benchmarks are chosen in order to investigate different program behaviors. Note that, an exhaustive evaluation of the benchmark suite follows in Section 4.3. All event arrival curves were extracted using Algorithm 2, while refining the event granularity to a single access. Additionally, an upper event arrival function $\eta_L^+(\Delta t)$ is generated using the same parameters, yet neglecting the loop differentiation and minimum iteration constraints introduced in Section 2.3. The sole purpose of this is to show the increased tightness due to these additional constraints in comparison to the previous work. In case of the benchmark `qurt`, $\eta_L^+(\Delta t)$ is identical to $\eta^+(\Delta t)$.

Figures 6a and 6b show $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ for the benchmarks `compressdata` and `qurt`. For both benchmarks the upper curve differs from the lower curve. This is caused by variable loop bounds, conditional statements and multiple program exits. E.g., the benchmark `qurt` can terminate with solely 12 data accesses in total or with up to 48.

Figures 6c and 6d depict the lower and upper arrival curve functions for the benchmarks `sqrt` and `binarysearch`. For both programs $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ converge to a common value. This results from the fact that each possible path through the program covers an identical total number of data accesses. However it is noteworthy that the minimum and maximum arrival of events per interval of time differs.

4.3 Granularity Evaluation

The execution time of the algorithm used for extracting the arrival curves depends on the granularity considered. Figure 7 depicts the upper event arrival functions for the benchmark `compressdata` considering different granularities. The finest possible granularity (i.e., $\Delta t = 1$

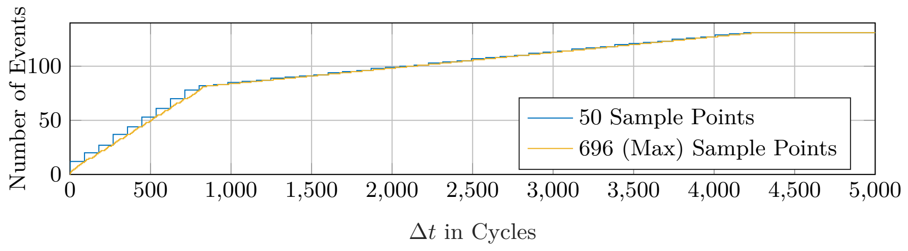


Figure 7 Event arrival curves with a different granularity and therefore number of samples.

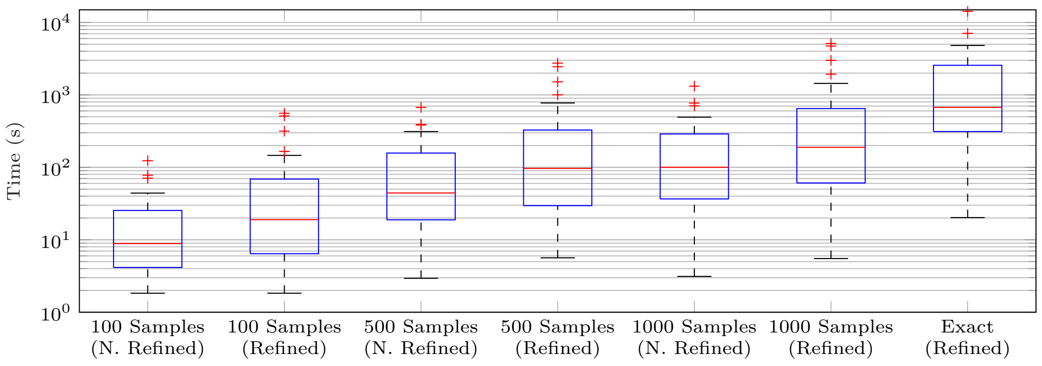


Figure 8 Overall runtimes of the extraction algorithms with different granularities.

cycle) is leading to a total number of 696 sample points (using the presented Algorithm 2). A more coarse granularity using only a total of 50 sample points is depicted as well. Note that the arrival curve with a coarse granularity always dominates the arrival curve with a finer granularity therefore leading to a *safe* approximation of the arrival curve. Even though we reduce the number of sample points, we still receive an arrival curve very close to the possible finest granularity but with the benefit of a smaller execution time. This obviously depends on the structure of the program under analysis.

Figure 8 depicts the overall execution times of the extraction, separated by the applied granularity. It is differentiated between the total number of sample points and considering the utilization of the proposed basic block refinement in Section 3.2. The right-hand side boxplot shows the execution time when using the binary search approach (Algorithm 2, 5h timeout). The central mark of each box denotes the median, while the edges depict the 25th and 75th percentiles. The maximum whisker length is defined as 1.5 times the difference between the 75th and 25th percentile. Note that a higher number of sample points leads to a finer granularity and therefore more precision of the results. The refined BB approach also leads to more precise results as it isolates the instructions potentially accessing data, compared to the non-refined BB approach where a basic block may contain multiple data accesses. However, the refinement leads to a more complex ILP model and therefore longer execution times.

Therefore, we can clearly see that the execution time increases as we increase the number of sample points. As expected, the execution time increases as well with the utilization of the BB refinement. While the median of the extraction runtime without a basic block refinement and just 100 samples is about 9 seconds, it increases to 189 seconds with 1000 samples and refinement applied. The median of the binary search approach runtime is 673 seconds. Out of the 34 benchmarks evaluated, 10 benchmarks were canceled due to the 5h timeout when

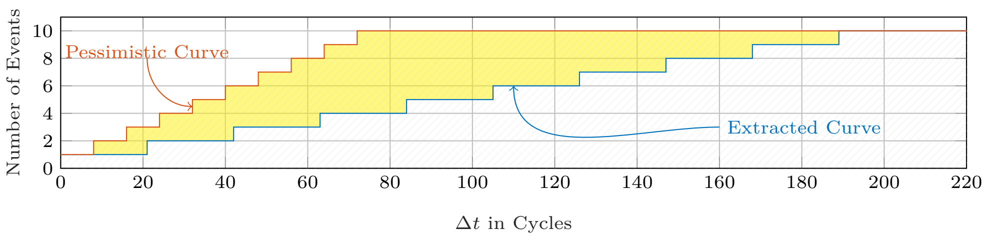


Figure 9 The metric used to evaluate the overapproximation is based on the area between the extracted curve and a corresponding pessimistic curve.

performing Algorithm 2. Therefore, there is clearly a trade-off to find between precision and execution time. In the following, we present a metric to measure the precision loss resulting from a coarser granularity approach.

4.4 Measuring the Overapproximation

In order to evaluate the level of overapproximation, we introduce the metric d_{appr}. The metric d_{appr} is defined as the area between the extracted curve and a corresponding simplistic pessimistic curve, normalized on the area below the pessimistic curve. The pessimistic curve only takes into account the maximum number of events over a complete program path and the minimum time between two events (given, e.g., by memory latencies). Therefore, d_{appr} is defined as follows:

$$d_{\text{appr}} = \frac{A_{\text{Pess}} - A_{\text{Extr}}}{A_{\text{Pess}}} \tag{52}$$

Whereas A_{Pess} is the area below the pessimistic curve and A_{Extr} is the area below the extracted curve. Figure 9 depicts the parameters used. The upper curve represents the pessimistic curve, solely generated using the maximum overall number of events and minimum time between events. The lower curve represents a curve extracted using the presented ILP model. The area difference (marked in yellow) is calculated and then normalized on the total area below the pessimistic curve. Thereby, d_{appr} reflects a magnitude to which extend the extracted curve is tighter in comparison to the pessimistic approach. When comparing the metric d_{appr} of curves extracted using different parameters of granularity (e.g., basic block refinement), the level of introduced overapproximation can be evaluated. A higher value of d_{appr} denotes a tighter curve, hence most likely leading to a tighter system-level analysis.

Figure 10 shows d_{appr} for the extracted upper arrival curves using 100, 500 and 1000 sample points. Accesses are considered separately considering a refined BB approach or bundled as initially structured by the program. It also depicts d_{appr} for upper arrival curves using the binary search approach (cf. Algorithm (2)) with basic block refinement applied. In cases the binary search algorithm was canceled due to the 5h timeout, the bar is not depicted in the diagram. The pessimistic reference curve for each benchmark was generated by using the maximum overall number of events and the minimum number of cycles between two events, given by the memory latencies. The benchmarks are listed on the x-axis. Benchmarks `janne_complex`, `expint`, `fac`, `fibcall`, `prime`, `recursion` and `cover` were evaluated but are not shown in the diagram, since no potential data accesses were detected (no data was allocated to the `.data` section).

As expected, d_{appr} is always greater or equal for a fixed number of samples when considering separated requests in comparison to bundled requests. The highest relative difference comparing separated and bundled accesses at a fixed number of sample points

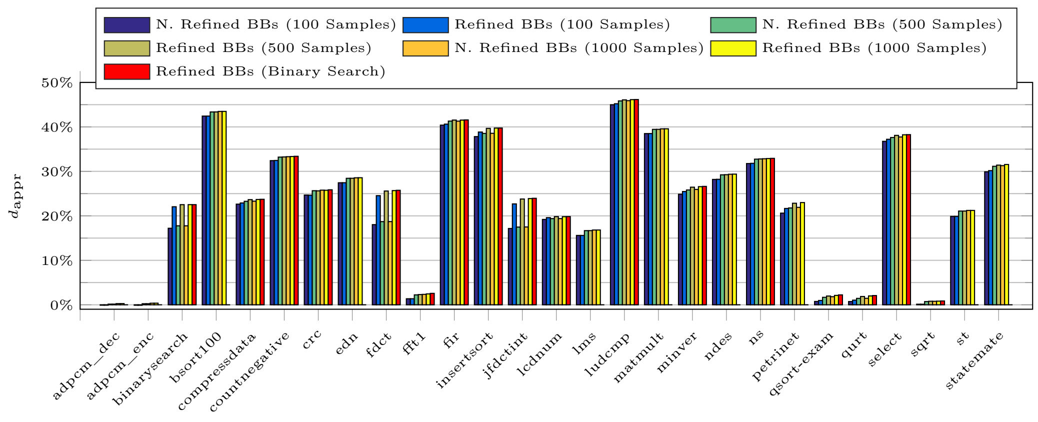


Figure 10 Metric d_{appr} for benchmarks of the MRTC benchmark suite [9].

occurs, amongst others, at the `fdct` benchmark. Using 1000 sample points and considering all requests separated, d_{appr} is at 25.7%, whereas considering accesses bundled per BB (same number of samples) results a d_{appr} value of only 18.7%. However, there are also several benchmarks for which the consideration of separated requests does not result in a lower value of d_{appr}. The benchmark `bsort100` represents such an example. Though d_{appr} increases with the number of samples, it is irrelevant whether requests are split into single blocks or not.

An exception can be seen for the benchmarks `adpcm_decoder` and `adpcm_encoder` when extracted with only 100 samples (BB refinement irrelevant), as they yield a value of d_{appr} of -1%. This is due to the low number of sample points in regard to the benchmarks' size and event arrival curves' steepness. Besides, for no other benchmark and granularity configuration a negative value of d_{appr} was observed. Overall it can be observed that d_{appr} is increasing with a higher number of samples as it is expected.

Bringing together the results regarding the required runtime from Figure 8 and the quality of the approximated curves, we can conclude that approximating the event arrival curve offers a good trade-off between extraction time and quality. If we take the benchmark `crc` as an example, the required extraction time for 500 samples and without basic block refinement drops by 98% in comparison to the extraction using the binary search algorithm in combination with refinement. Yet, d_{appr} only drops by 0.2%.

5 Conclusion and Future Work

In this paper we presented an approach to extract safe and tight event arrival functions from code-level analysis. The extracted event arrival functions can be generated with an adjustable level of granularity in order to reduce the execution time of the proposed extraction algorithm. Despite the induced overapproximation by the choice of the granularity, the presented approach results safe upper bounds of the actual event arrival curve. Furthermore, it has been shown that for some benchmarks a very good trade-off can be achieved in order to extract rapidly event arrival functions with a very good quality precision.

As a part of future work, we plan to integrate calling contexts into the model. This could further improve the tightness. Besides, we plan to exploit the detailed event arrival function knowledge for optimizations, hence improving a system's worst-case timing using the gained informations.

References

1 B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Proceedings of the 2011 Design, Automation & Test in Europe Conference & Exhibition*, 2011. `doi:10.1109/DATE.2011.5763145`.

2 S. Altmeyer, C. Burguière, and R. Wilhelm. Computing the Maximum Blocking Time for Scheduling with Deferred Preemption. In *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*, 2009. `doi:10.1109/STFSSD.2009.12`.

3 Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A Generic and Compositional Framework for Multicore Response Time Analysis. In *Proceedings of the 2015 International Conference on Real Time and Networks Systems*, 2015. `doi:10.1145/2834848.2834862`.

4 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5), 2012. `doi:10.1007/s11241-012-9152-2`.

5 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional Performance Analysis in Python with pyCPA. In *Proceedings of the 2012 International Workshop on Analysis Tools and Methodologies for Embedded and Real-time System*, 2012.

6 Leonardo Ecco, Selma Saidi, Adam Kostrzewa, and Rolf Ernst. Real-time DRAM throughput guarantees for latency sensitive mixed QoS MPSoCs. In *Proceedings of the 2015 IEEE International Symposium on Industrial Embedded Systems*, 2015. `doi:10.1109/SIES.2015.7185038`.

7 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proceedings of the 2016 International Workshop on Worst Case Execution Time Analysis*, 2016. `doi:10.4230/OASIcs.WCET.2016.2`.

8 Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2), 2010. `doi:10.1007/s11241-010-9101-x`.

9 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Proceedings of the 2010 International Workshop on Worst-Case Execution Time Analysis*, 2010. `doi:10.4230/OASIcs.WCET.2010.136`.

10 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2), 2005. `doi:10.1049/ip-cdt:20045088`.

11 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET Analysis for Multi-core Processors with Shared Buses and Event-driven Bus Arbitration. In *Proceedings of the 2015 International Conference on Real Time and Networks Systems*, 2015. `doi:10.1145/2834848.2834872`.

12 Bisschop Johannes. *AIMMS. Optimization Modeling.* Haarlem, The Netherlands, 2009.

13 Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems.* PhD thesis, TU Dortmund University, Dortmund / Germany, 2015.

14 J. C. Kleinsorge, H. Falk, and P. Marwedel. Simple analysis of partial worst-case execution paths on general control flow graphs. In *Proceedings of the 2013 International Conference on Embedded Software*, 2013. `doi:10.1109/EMSOFT.2013.6658594`.

15 Adam Kostrzewa, Selma Saidi, and Rolf Ernst. Dynamic Control for Mixed-Critical Networks-on-Chip. In *Proceeding of the 2015 IEEE Real-Time Systems Symposium*, 2015. `doi:10.1109/RTSS.2015.37`.

16 Y. Li, H. Salunkhe, J. Bastos, O. Moreira, B. Akesson, and K. Goossens. Mode-controlled data-flow modeling of real-time memory controllers. In *Proceedings of the 2015 IEEE Symposium on Embedded Systems For Real-time Multimedia*, 2015. doi:10.1109/ESTIMedia.2015.7351770.

17 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 1995 Annual ACM/IEEE Design Automation Conference*, 1995. doi:10.1145/217474.217570.

18 Matthieu Moy and Karine Altisen. Arrival Curves for Real-Time Calculus: The Causality Problem and Its Solutions. In *Proceedings of the 2010 International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2010. doi:10.1007/978-3-642-12002-2_31.

19 Dominic Oehlert, Arno Luppold, and Heiko Falk. Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems. In *Proceedings of the 2017 Euromicro Conference on Real-Time Systems*, June 2017.

20 K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002. doi:10.1109/DATE.2002.998348.

21 Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 2015 International Conference on Hardware/Software Codesign and System Synthesis*, 2015. doi:10.1109/CODESISSS.2015.7331385.

22 T. Sewell, F. Kam, and G. Heiser. Complete, High-Assurance Determination of Loop Bounds and Infeasible Paths for WCET Analysis. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016. doi:10.1109/RTAS.2016.7461326.

23 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century.*, 2000. doi:10.1109/ISCAS.2000.858698.

24 S. Wasly and R. Pellizzoni. A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems. In *Proceedings of the 2013 Euromicro Conference on Real-Time Systems*, 2013. doi:10.1109/ECRTS.2013.28.

25 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.

A Measurement-Based Model for Parallel Real-Time Tasks

Kunal Agrawal[1]
Washington University in St. Louis
St. Louis, MO, USA
kunal@wustl.edu
https://orcid.org/0000-0001-5882-6647

Sanjoy Baruah[2]
Washington University in St. Louis
St. Louis, MO, USA
baruah@wustl.edu
https://orcid.org/0000-0002-4541-3445

Abstract

Under the federated paradigm of multiprocessor scheduling, a set of processors is reserved for the exclusive use of each real-time task. If tasks are characterized very conservatively (as is typical in safety-critical systems), it is likely that most invocations of the task will have computational demand far below the worst-case characterization, and could have been scheduled correctly upon far fewer processors than were assigned to it assuming the worst-case characterization of its run-time behavior. Provided we could safely determine during run-time when all the processors are going to be needed, for the rest of the time the unneeded processors could be idled in low-energy "sleep" mode, or used for executing non-real time work in the background. In this paper we propose a model for representing parallelizable real-time tasks in a manner that permits us to do so. Our model does not require us to have fine-grained knowledge of the internal structure of the code represented by the task; rather, it characterizes each task by a few parameters that are obtained by repeatedly executing the code under different conditions and measuring the run-times.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Real-time schedulability, Theory of computation → Parallel computing models

Keywords and phrases multiprocessor federated scheduling, parallel tasks, work and span, mixed criticality

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.5

1 Introduction

Scheduling theory is concerned with the analysis of real-time systems. As multiprocessor and multicore implementations of real-time systems become prevalent, it is desirable that the models used in scheduling theory for representing real-time workloads be capable of exposing the parallelism that may exist within these workloads. This need has given rise to formal task models such as the *fork-join* model [1, 2], the *sporadic DAG tasks* model [3] (see [4, Chapter 21] for a text-book description), the *multi-DAG* model [5], the *conditional DAG*

[1] Supported in part by NSF grants CCF-1337218 and CCF-173387
[2] Supported in part by NSF Grants CNS 1409175, CPS 1446631, and CNS 1563845

tasks model [6, 7] etc. Each of these models represents the internal structure of the piece of code being modeled at a relatively fine level of granularity, with the parallelism in the code typically modeled as a directed acyclic graph (DAG). Each vertex in such a DAG represents a segment of sequential code, and edges represent precedence constraints between such code segments: the segment of sequential code represented by the vertex at the tail of an edge much complete execution before the segment of sequential code represented by the vertex at the head of the edge may begin to execute.

Such DAG-based models for representing parallel real-time code have proved popular in the real-time scheduling theory community, and much important and interesting research has been accomplished that is based upon representing systems using these models. This body of research has indeed provided us with a deeper insight into the issues that arise in exploiting parallelism in multiprocessor real-time systems; however due to a variety of reasons (some of which are enumerated and discussed in some detail in Section 2) there are some classes of real-time applications for which such DAG-based representations may not be appropriate for the purposes of schedulability analysis; alternative representations are needed. In this paper we propose one such possible alternative representation that may be suitable under certain circumstances. In this model we do not attempt to explicitly represent the internal parallel structure of the code. Instead, we seek to identify a few important parameters of parallalizable code that are most useful for scheduling algorithms that seek to schedule such code upon multiprocessor platforms, and propose that the code be looked upon as a "black box" that is characterized by just these parameters. Furthermore, we do not require that the internal structure of the code be examined in order to obtain these parameter values. Rather, we propose that values for these parameters be estimated via extensive simulation experiments: repeatedly executing the code in a controlled laboratory environment in order to be able to compute bounds on the parameter values. (Such an approach is inspired by the large and growing body of current research [8] on probabilistic worst-case execution time – pWCET – analysis.) Since measurement-based approaches are typically not able to provide parameter values that are guaranteed correct with absolute certainty, we incorporate, from the mixed-criticality scheduling literature [9], Vestal's idea [10] of characterizing a single task with two sets of parameters: one set very conservative and hence trusted to a very high level of assurance and the other, far less conservative but more representative of "typical" behavior.

Organization. The remainder of this paper is organized as follows. In Section 2 we motivate the new model by identifying relevant characteristics of parallelizable real-time code that current models are not well-suited to represent, and formally define the workload and system model that we are proposing. In Section 3 we briefly discuss some prior research that provides the foundations upon which our proposed model is built. In Section 4 we derive, and prove the correctness and other relevant properties of, an algorithm for scheduling systems represented using the proposed model. Our overall objective is to be able to obtain more resource-efficient implementations of systems, while ensuring correctness; in Section 5 we explore some possible means of further enhancing the efficiency of the algorithm presented in Section 4. We conclude in Section 6 with a discussion on the relevance, significance, and limitations of our proposed model, and an enumeration of possible directions for continued research.

2 System model: Motivation and Definition

In this section we flesh out the details of the model we are proposing for representing parallelizable real-time code that is not conveniently represented using previously-proposed DAG-based task models. We will first motivate the model informally, and seek to explain aspects of the model via illustrative examples. A formal definition of the model is then provided in Section 2.1; our proposed algorithm for scheduling tasks represented using this model is described in Section 2.2.

Why a new model? As stated in Section 1 above, several excellent DAG-based models for representing parallel real-time code have been developed in the real-time scheduling theory community; however there are some classes of real-time applications for which such models have proved unsuitable. This may be for one or more of the following reasons:

1. The internal structure of the parallel code may be very complex, with multiple conditional dependencies (as may be represented in e.g., the conditional DAG tasks model [6, 7]) and (bounded) loops. Explicit enumeration of all possible paths through such code in order to identify worst-case behavior may be computationally infeasible.[3]
2. If some parts of the code are procured from outside the application-developers' organization, the provider of this code may seek to protect their intellectual property (IP) by not revealing the internal structure of the code and instead only providing executables – this may be the case if, e.g., commercial vision algorithms are used in a real-time application. (Although reverse-engineering of the executable code in order to determine its internal structure may be possible in principle, such reverse engineering tends to be tedious and error-prone.)
3. Algorithms for the analysis of systems represented using DAG-based models tend to have run-time pseudo-polynomial or exponential in the size of the DAG. Such run-times have traditionally been considered acceptably small enough to allow the algorithms to be practical in practice; however, this state of affairs may not continue in the future. For many cyber-physical real-time systems, constraints such as deadlines are typically dictated by physical factors. As the processors upon which we implement such cyber-physical real-time systems become increasingly more powerful, it becomes possible to incorporate far more complex processing that would be represented as larger DAGs than was previously the case. As this trend towards more complex processing and the consequent larger DAGs continues, run-times pseudo-polynomial in the size of these larger DAGs may become too large be used in practice during system design and analysis.
4. Further exacerbating the situation, explicitly representing the internal structure of some pieces of parallel code in DAG form results in DAGs that may be of size exponential in the size of the code. Consider, for example, the following code snippet written in OpenMP (`http://www.openmp.org/`), an application programming interface (API) that supports multi-platform shared memory multiprocessing programming:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<10; i++) {
    //do_something
 }
```

[3] We point out that techniques for *approximating* the worst-case behavior of complex conditional parallelizable code have been proposed with regards to specific scheduling algorithms such as global fixed-priority [6], global EDF [7] or federated [11].

This code snippet would translate to a DAG with $(1 + 10 + 1 =)$ 12 nodes. If we were to replace the "10" in the upper bound of the `for` loop with a "100", however, the resulting DAG would have 102 nodes; replacing it with "1000" would yield 1002 nodes, etc. – increasing the size of the program by one ASCII character results in an almost ten-fold increase in the size of the DAG.

5. Particularly for conditional code, it may be the case that the true worst-case behavior of the code is very infrequently expressed during run-time.[4]. Traditional models based on conditional DAGs may not be suitable for representing such code (although mixed-criticality [10, 12, 13, 14] extensions of such conditional DAG models are a possibility – to our knowledge, such models have not yet been proposed, let alone studied).

For pieces of parallel real-time code possessing one or more of the characteristics discussed above, DAG-based representations may not be appropriate for the purposes of schedulability analysis; alternative representations are needed. Let us now discuss what such a representation should provide.

Identifying relevant characteristics of parallelizable real-time code. In modeling parallelizable real-time code that is to be executed upon a multiprocessor platform, a prime objective is to enable the exploitation of the parallelism that may be present in the code by scheduling algorithms, in order to enhance the likelihood that we will be able to meet timing constraints. We are interested here in developing *predictable* real-time systems – systems that can have their timing (and other) correctness verified prior to run-time. For the purposes of enabling a priori timing verification, decades of research in the parallel computing community suggests the following two timing parameters of a piece of parallelizable code are particularly significant:

1. The **work** parameter denotes the cumulative worst-case execution time of all the parallel branches that are executed across all processors. Note that for non-conditional parallelizable code this is equal to the worst case execution time of the code on a single processor (ignoring communication overhead from synchronizing processors).
2. The **span** parameter denotes the maximum cumulative worst-case execution time of any sequence of precedence-constrained pieces of code. It represents a lower bound on the duration of time the code would take to execute, regardless of the number of processors available.
   The span of a computation is also called the *critical path length* of the computation, and a sequence of precedence-constrained pieces of code with cumulative worst-case execution time equal to the span, a *critical path* through the computation.

The relevance of these two parameters arises from well-known results in scheduling theory concerning the multiprocessor scheduling of precedence-constrained jobs (i.e., DAGs) to minimize makespan – this is the widely-studied $\mathrm{P}|\ \mathrm{prec}|\ \mathcal{S}_{\max}$ problem in the classic 3-field $\alpha \mid \beta \mid \gamma$ notation that is commonly used in scheduling theory [15]. This problem has long been known to be NP-hard in the strong sense [16]; i.e., computationally highly intractable. However, Graham's *list scheduling* algorithm [17], which constructs a work-conserving schedule by executing at each instant in time an available job, if any are present, upon any available processor, performs fairly well in practice. It was shown [17] that list

[4] Consider, for example, a real-time application that periodically monitors a sensor for anomalous input. Most of the time the sought-for anomolous input is not detected, and not much computation needs to be performed. But on the rare occasions when anomalous input is detected, considerable additional processing of such input is necessary.

scheduling makes the following guarantee: if $\mathcal{S}_{\max}$ denotes the minimum makespan with which a particular DAG can be scheduled upon m processors, then the schedule generated by list scheduling this DAG upon m processors will have a makespan no greater than $(2 - \frac{1}{m}) \times \mathcal{S}_{\max}$. This result, in conjunction with a hardness result in [18] showing that determining a schedule for this DAG of makespan $\leq \frac{4}{3}\mathcal{S}_{\max}$ remains NP-hard in the strong sense[5], suggests that list scheduling is a reasonable algorithm to use in practice, and in fact most run-time scheduling algorithms that are used for scheduling DAGs upon multiprocessors use some variant or the other of list scheduling. We will do so in this paper as well.

An upper bound on the makespan of a schedule generated by list scheduling is easily stated. Letting *work* and *span* denote the work and span parameters of the DAG being scheduled, it has been proved in [17] that the makespan of the schedule for a given DAG is guaranteed to be no larger than

$$\frac{\mathit{work} - \mathit{span}}{m} + \mathit{span} \tag{1}$$

Thus a good upper bound on the makespan of the list-scheduling generated schedule for a DAG may be stated in terms of only its work and span parameters. Equivalently if the DAG represents a real-time piece of code characterized by a relative deadline parameter D, $(\frac{\mathit{work} - \mathit{span}}{m} + \mathit{span}) \leq D$ is a sufficient test for determining whether the code will complete by its deadline upon an m-processor platform. *We therefore identify the work and span parameters of a piece of parallel real-time code as being particularly relevant from the perspective of schedulability analysis.*

A measurement-based approach to parameter estimation. The work and span parameters of tasks that are represented using DAG-based models are quite straightforward to compute in time linear in the representation of the DAGs (algorithms for doing so are described in [3, 7]). As we have discussed above, however, our interest is in characterizing parallel code that is typically not conveniently represented using DAG-based models. We propose that for the purposes of representing pieces of such code for schedulability analysis, we *ignore* their internal structure and instead seek to characterize them solely via their work and span parameters. And since we cannot in general determine the precise values of the work and span parameters of a piece of code without knowing its internal structure, we advocate here that *measurement-based* approaches be used to *estimate* these parameters. Measurement-based approaches have been developed for estimating probabilistic worst-case execution time (pWCET) [20, 21, 22] distributions of individual pieces of code, and implemented in pWCET tools such as RapiTime (`https://www.rapitasystems.com/products/rapitime`) from Rapita Systems. We now briefly discuss how measurement-based approaches may be adapted to estimate the work and span parameters of parallel code. Ignoring overhead associated with implementing global scheduling, observe that the work parameter of a piece of code is equal to the time needed to complete its execution upon a single processor, while its span parameter is equal to the time needed to complete its execution upon an unbounded number of processors. Hence one can estimate the probability distribution of the work parameter of a piece of code by using pWCET techniques to estimate its WCET distribution upon a single processor. One can similarly estimate the probability distribution of the span parameter by

[5] In fact, assuming a reasonable complexity-theoretic conjecture that is somewhat stronger than P $\neq$ NP, a result of Svensson [19] implies that a polynomial-time algorithm for determining a schedule of makespan $\leq 2\mathcal{S}_{\max}$ for all m is ruled out.

1. first adapting the measurement-based techniques underpinning pWCET-estimation to determine the makespan probability distribution upon a given number of processors (rather than the completion-time upon a single processor); and then
2. estimating the makespan distribution of the parallel code upon platforms in which the number of processors is repeatedly increased, until further increases do not result in significant changes to the estimated distribution.

The proposed model: multiple work and span estimates. As discussed above, it is possible, by suitable adaptation and application of pWCET techniques, to estimate probability distributions for the work and span parameters of pieces of parallel real-time code. It is understood in the pWCET community that pWCET-based techniques cannot in general determine bounds that are guaranteed to be correct with absolute certainty; rather, they provide bounds that are guaranteed correct to specified probabilistic degrees of confidence/ levels of assurance (Davis et al. [8] provide a thoughtful and considered discussion as to how the concept of probabilities should be interpreted when used in such a manner). In the model we propose for representing parallel real-time code, we suggest that each such piece of code be characterized by *two* pairs of (*work*, *span*) parameter values, each pair corresponding to a different probability threshold in the work and span distributions and therefore valid at different levels of assurance. Specifically, one pair of values should be *very* conservative and therefore trusted to a very high level of assurance and the other, while still relatively safe, should be more representative of "typical" behavior, not attempting to cover scenarios that are highly unlikely to occur. We illustrate via an example.

▶ **Example 1.** Suppose that we were able to determine for a piece of code that

- Its work parameter is > 120 with some small probability p, but > 900 with a far smaller probability $p' \ll p$.
- Its span parameter is > 40 with probability p; however the probability that it is > 600 is p'.

We could characterize this piece of code with two ordered pairs of (work,span) values – a $(1 - p)$ probability of being $\leq (120, 40)$, and a far greater $(1 - p')$ probability of being $\leq (900, 600)$.

We require that *correctness criteria hold under the more conservative estimate.* Suppose for instance that it were specified that this code should execute within a relative deadline of D; we require that the makespan by $\leq D$ provided $work \leq 900$ and $span \leq 600$. ◀

The proposed run-time scheduling approach. Since correctness is defined with respect to the more conservative estimates for work and span, in order to satisfy correctness requirements we must provision computing resources to a task assuming these more conservative estimates. However, it is our expectation that the task's run-time behavior is very likely to be bounded by the less conservative parameter estimates, and hence statically provisioning adequate resources for it under the more conservative assumptions is likely to result in significant wastage of computing resources during run-time. One manner of ameliorating such wastage is by keeping some of the provisioned resource in "reserve", perhaps by placing some processors in sleep mode or having them execute background (non real-time) work, with the option of switching them to work upon executing the task if we determine, during run-time, that the task's run-time behavior is in fact not likely to be bounded by the less conservative parameter estimates. The following example illustrates.

▶ **Example 2.** Suppose the code in Example 1 to be scheduled with a relative deadline equal to 690, upon a 10-processor platform. According to Expression 1, the makespan of the schedule upon 10 processors is no more than

$$\frac{900-600}{10} + 600 = (30 + 600) = \mathbf{630}$$

assuming the more conservative work and span estimates hold. Hence, correctness is guaranteed.

In fact, ten processors are not necessary for correctness: if we were executing this piece of code upon just four processors, the corresponding makespan bound according to Expression 1 would be $(\frac{900-600}{4}) + 600 = \mathbf{675}$. Our run-time algorithm could therefore realize some energy savings by simply switching off six of the ten provided processors, and executing the task on the remaining four.

Could we switch off seven processors? The reader may verify that with three processors Expression 1would yield a makespan bound of $(\frac{900-600}{3}) + 600 = \mathbf{700}$. Since 700 exceeds the specified relative deadline of 690, we conclude that we may miss the deadline if we were to switch off seven processors and the system behaved worse than anticipated by its less conservative parameters (although not its more conservative parameters). We therefore conclude that we need at least **4** processors to not be in sleep mode, in order to ensure correctness.

The run-time algorithm we will derive in this paper is designed for systems in which the less conservative parameters are very likely to hold "most of the time"; i.e., the value of p is itself very small. Provided such is the case for this example

- our run-time scheduling algorithm starts out scheduling the system on just three processors, leaving the remaining seven processors in sleep mode.
- If execution has not completed by some time-instant (whose value is precomputed), it wakes up the sleeping processors and makes all ten processors available for this task to execute upon.

We will prove later that with this algorithm, the task completes by the specified deadline of 690 provided the more conservative task parameters hold; **correctness** is thus established. Additionally, there is a $\leq p$ probability that it will not complete by the pre-computed time-instant and hence need to awaken the remaining seven processors.

To evaluate the **efficiency**, suppose, for this example, that $p = 0.05$, indicating that the less conservative parameters hold with 95% probability. There is therefore a $\leq 5\%$ probability that the task will not complete by the specified time-instant,[6] and the expected number of processors that would be needed is no larger than

$$\big(0.95 \times 3 + 0.05 \times 10\big) = \big(2.85 + 0.5\big) = \mathbf{3.35}$$

in contrast to the four that would be needed if our run-time scheduling algorithm were to not be used. ◀

Examples 1 and 2 above have illustrated the task model, and the associated run-time strategy for scheduling tasks that are so modeled, that we are proposing in this paper. We now formally define the task model in Section 2.1, and the run-time scheduler in Section 2.2, below.

[6] In fact, since the work and span parameters will not in general be perfectly correlated, the expected probability of this happening is likely to be far less than 5%. (This issue is revisited in Section 5.)

2.1 System Model

We now provide a formal definition of our model, by describing in detail the workload model we assume. The workload we seek to model comprises a single piece of parallelizable real-time code that is characterized by the following list of parameters:

$$\Big\langle \mathit{work}_O, \mathit{span}_O, \mathit{work}_N, \mathit{span}_N, D \Big\rangle,$$

with the following interpretation:

1. $(\mathit{work}_O, \mathit{span}_O)$. These represent very conservative estimates of the true "worst-case" values of the work and span parameters; as discussed above, we expect that these estimates will be obtained using the kinds of measurement-based techniques that have been developed for estimating probabilistic worst-case execution time (pWCET) distributions.
2. D denotes the *relative deadline* parameter: for correct execution it is required that the job be scheduled with makespan no greater than D.

   We highlight here that timing correctness is specified assuming that the $(\mathit{work}_O, \mathit{span}_O)$ parameter estimates are correct: the code is required to complete execution within the specified relative deadline D provided its work and span parameters are no larger than work_O and span_O respectively.
3. $(\mathit{work}_N, \mathit{span}_N)$. These are less conservative estimates on the values of the work and span parameters: it is expected that the actual values of the work and span parameters are *very* likely to be no larger than work_N and span_N respectively. (The subscript "N" in work_N and span_N stand for "nominal" [23].)

   These parameter estimates play no role in defining correctness; rather (as we have seen in Examples 1 and 2), their values may be used for the purposes of devising more resource-efficient scheduling strategies. It is hence not as critical that their values be assigned correctly as it is for the work_O and span_O parameters: while incorrectly estimated values for work_O and span_O may compromise timing correctness in the sense that we may end up missing deadlines, incorrectly estimated values for work_N and span_N simply result in less efficient implementations.

   We assume that $\mathit{work}_N \le \mathit{work}_O$ and $\mathit{span}_N \le \mathit{span}_O$. (Although our results are readily extended to situations where these assumptions do not hold, we do not see a rationale for relaxing these assumptions, since by very definition the $(\mathit{work}_N, \mathit{span}_N)$ parameters represent less conservative estimates than the $(\mathit{work}_O, \mathit{span}_O)$ parameters.)

In this paper, we consider the scheduling of a single such task upon a dedicated bank of identical processors. We point out that our results are directly applicable to the scheduling of recurrent – *periodic* or *sporadic* – real-time DAGs under the federated paradigm [24] of multiprocessor scheduling, provided each periodic/ sporadic task satisfies the additional constraint that its relative deadline parameter is no larger than its period parameter (i.e., they are *constrained-deadline* tasks). We believe our approach is particularly appropriate for scheduling systems of recurrent tasks: for such tasks, we anticipate that the $(\mathit{work}_N, \mathit{span}_N)$ parameters will bound the behavior of most invocations ("dag-jobs" [3]) of the task, with an occasional rare dag-job exceeding these bounds. Hence many of the allocated processors will remain in sleep mode most of the time, with the occasional dag-job requiring that the sleeping processors be awakened until that dag-job completes execution, after which they can be returned to sleep mode.

2.2 The Scheduling Algorithm

Given a task as specified above:

$$\Big\langle work_O, span_O, work_N, span_N, D \Big\rangle,$$

that is to be executed upon a platform comprising m identical processors, we first perform some **pre-run-time** schedulability analysis that determines whether we are able to schedule the task upon the m processors in a manner that ensures correctness. Recall that *correctness* is specified as requiring that the task meet its deadline provided its work parameter is $\leq work_O$ and its span parameter is $\leq span_O$: by Expression 1, this is guaranteed provided

$$\left(\frac{work_O - span_O}{m} + span_O\right) \leq D; \tag{2}$$

If Condition 2 does not hold, our scheduling algorithm declares failure: it is unable to schedule this instance in a manner that guarantees timing correctness. Otherwise, it computes a pair of values m_N and $\mathcal{S}_N$ – the manner in which these values are computed will be derived in Section 4.2. These computed parameters have the following intended interpretation: provided the less conservative work and span parameter estimates are correct (i.e., work is $\leq work_N$ and span is $\leq span_N$ for the task), list scheduling can schedule the task upon m_N processors to have a makespan no greater then $\mathcal{S}_N$.

Run-time scheduling. Suppose that the piece of parallelizable real-time code represented by this task is activated at some time-instant t_o during run-time.

1. The scheduler sets a timer to go off at time-instant $(t_o + \mathcal{S}_N)$, and begins executing the task upon m_N processors using the list-scheduling algorithm [17]. The remaining $(m - m_N)$ processors assigned to this task are placed/ remain in sleep mode.
2. If the task has not completed execution by time-instant $(t_o + \mathcal{S}_N)$, then the scheduler awakens the $(m - m_N)$ sleeping processors, and uses list-scheduling to execute the remainder of the task upon the entire bank of m processors.
3. As mentioned in Section 2.1 above, in the case of recurrent tasks these awakened processors are returned to sleep mode upon completion of execution of the current dag-job of the task.

In Section 4 we will derive the manner in which the values of m_N and $\mathcal{S}_N$ are to be computed in order to guarantee correctness: the algorithm completes execution of the task within D time units of its arrival, provided its work parameter is $\leq work_O$ and its span parameter is $\leq span_O$.

We close this section with an example illustrating the operation of our run-time scheduler.

► **Example 3.** Consider once again the instance discussed in Examples 1 and 2. In the notation of Section 2, this task is represented by the following parameters:

$$\Big\langle work_O, span_O, work_N, span_N, D \Big\rangle \quad = \quad \Big\langle 900, 600, 120, 40, 690 \Big\rangle\ .$$

It is to be scheduled upon $m = 10$ processors. In Example 4 we will show that the algorithm of Section 4 assigns the parameter m_N the value 3, and the parameter $\mathcal{S}_N$, a value $66\frac{2}{3}$. Hence our run-time scheduler starts out scheduling this task on 3 processors. If the task behaves as specified by its $work_N, span_N$ parameters, then by Expression 1 the makespan is no more than

$$\frac{120-40}{3}+40 \quad = \quad 26\frac{2}{3}+40 \quad = \quad \mathbf{66\frac{2}{3}}$$

and hence the additional seven processors are not needed. If it does not complete by time-instant $66\frac{2}{3}$, all ten processors become available for this task to execute upon, and results in Section 4 allow us to conclude that the task does execute correctly, completing by the specified deadline at time-instant 690. ◀

3 Related Work

The approach to the modeling and run-time scheduling of parallelizable tasks that we are proposing here draws inspiration from research in the areas of *parallel computing*, *mixed-criticality scheduling*, and *probabilistic WCET*. As stated in Section 2 above, the problem of scheduling DAGs to minimize makespan (the $\text{P}|\text{ prec}|\ \mathcal{S}_{\max}$ problem in 3-field notation [15]) has been very widely studied in "traditional" scheduling theory. Given the inherent intractability of this problem [16] and the existence of a good approximation (as represented by List Scheduling [17] with its associated makespan bound – Inequality 1), the parallel computing community soon began to focus upon the work and span parameters as reasonable proxies for parallelizable computational workloads; this is one of the fundamental ideas that underpins our proposed approach.

The concept of specifying multiple values, which are considered trustworthy to different levels of assurance, to a task's parameters was proposed by Vestal [10] and forms the basis of mixed-criticality scheduling theory. There is a large body of research exploring the Vestal model – see [14] for a survey. In studying the scheduling of mixed-criticality parallel tasks, Li et al. [23] first proposed a model in which each task is characterized by different work and span parameters at low and high criticality levels – it is this model that we are studying in depth here. (The overall context of the research in [23] is quite different from ours: while we are, in the terminology of [23], considering the scheduling of a single parallelizable real-time task with the objective of minimizing the number of processors used in the nominal case while concurrently guaranteeing to meet deadlines in the overloaded case, [23] was concerned with devising mixed-criticality scheduling algorithms with good *capacity augmentation bounds*.)

Our approach also draws upon ideas from the considerable body of prior research (e.g., [20, 21, 22]) on measurement-based techniques for estimating probabilistic worst-case execution time distributions (pWCET). The correctness of our scheduling framework very strongly depends upon the validity and accuracy of pWCET-estimation techniques, since we are in effect guaranteeing correct timing behavior (meeting deadlines) under the assumption that the more conservative estimations – $work_O$ and $span_O$ – are correct upper bounds. In contrast, incorrect estimations of $work_N$ and $span_N$ do not compromise correctness, although they could have an adverse impact on efficiency. Rather than being considered as estimations of worst-case parameter values, these parameters are perhaps closer in spirit to what Chisholm et al [25] have called *provisioned* parameter values and Li et al. [23], *nominal* parameter values – values that represent typical or common-case behavior and may be obtained by, e.g., somewhat inflating average-case parameter values.

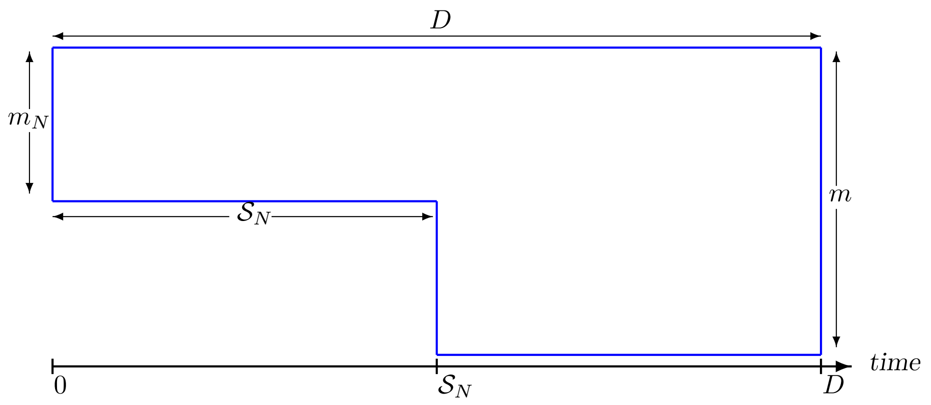


Figure 1 The parallel task begins execution at time-instant 0 with a deadline at time-instant D. It executes upon m_N processors over the interval $[0, \mathcal{S}_N)$, and upon m processors over the interval $[\mathcal{S}_N, D)$. (The x-axis thus denotes time, and the y-axis, the processors.)

4 Scheduling Algorithm Derivation and Analysis

Given a task characterized, as described in Section 2.1, by the parameters

$$\left\langle \text{work}_O, \text{span}_O, \text{work}_N, \text{span}_N, D \right\rangle$$

and m processors upon which to execute it, we discuss in this section how we should compute values of m_N and $\mathcal{S}_N$ in order to ensure that the run-time scheduling algorithm described in Section 2.2 above completes execution of the task within D time units of its arrival. We will start out in Section 4.1 assuming that values for m_N and $\mathcal{S}_N$ are already known, and derive sufficient conditions for ensuring timing correctness given these values of m_N and $\mathcal{S}_N$. We will then describe, in Sections 4.2, how values may be assigned to m_N and $\mathcal{S}_N$ in a manner that ensures that these sufficient conditions are satisfied.

4.1 Sufficient Schedulability Conditions

Suppose that we are given values of m_N and $\mathcal{S}_N$ (with $0 < m_N \le m$ and $0 \le \mathcal{S}_N \le D$), and the run-time algorithm schedules the task on m_N processors using list scheduling. If the task completes execution within $\mathcal{S}_N$ time units, correctness is preserved since $\mathcal{S}_N \le D$. It remains to determine sufficient conditions for correctness when the task does not complete by time-instant $\mathcal{S}_N$; this we do in the remainder of this section.

Figure 1 depicts the processors that are available for this task if it does *not* complete execution within $\mathcal{S}_N$ time units, thereby resulting in the run-time scheduler awakening the $(m - m_N)$ processors that had been in sleep mode over $[0, \mathcal{S}_N)$. We will now derive conditions for ensuring that the task completes execution by its deadline at time-instant D when executing upon these available processors, given that its work parameter may be as large as $\textit{work}_O$ and its span parameter, $\textit{span}_O$.

Let $\textit{work}'$ and $\textit{span}'$ denote the work and span parameters of the amount of computation of the parallel task that remains at time-instant $\mathcal{S}_N$ (these are > 0, since the task is assumed to not have completed execution by time-instant $\mathcal{S}_N$). This remaining computation executes upon m processors; By Expression 1 the overall makespan is therefore bounded from above by

$$\mathcal{S}_N + \left(\frac{\textit{work}' - \textit{span}'}{m} + \textit{span}' \right) \tag{3}$$

Since the remaining span at time-instant $\mathcal{S}_N$ is *span*$'$, an amount $(\mathit{span}_O - \mathit{span}')$ of the critical path of the task has executed during $[0, \mathcal{S}_N]$. At each instant when the critical path is not executing, it must be the case that all m_N processors are busy executing tasks not on the critical path. Hence the total amount of execution occurring over $[0, \mathcal{S}_N)$ is at least

$$\Big(\mathcal{S}_N - (\mathit{span}_O - \mathit{span}')\Big) \times m_N + (\mathit{span}_O - \mathit{span}'),$$

from which it follows that

$$\begin{aligned}
\mathit{work}' &\leq \mathit{work}_O - \mathcal{S}_N \times m_N + (\mathit{span}_O - \mathit{span}') \times m_N - (\mathit{span}_O - \mathit{span}') \\
&= \mathit{work}_O - \mathcal{S}_N \times m_N + (\mathit{span}_O - \mathit{span}') \times (m_N - 1) \\
&= \mathit{work}_O - \mathcal{S}_N \times m_N + \mathit{span}_O \times (m_N - 1) - \mathit{span}' \times (m_N - 1) \qquad (4)
\end{aligned}$$

Substituting Inequality 4 into the Expression 3, we obtain the following upper bound on the overall makespan:

$$\begin{aligned}
&\mathcal{S}_N + \Big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N + \mathit{span}_O \times (m_N - 1) - \mathit{span}' \times (m_N - 1) - \mathit{span}'}{m} + \mathit{span}'\Big) \\
&= \mathcal{S}_N + \Big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N + \mathit{span}_O \times (m_N - 1) - \mathit{span}' \times m_N}{m} + \mathit{span}'\Big) \\
&= \mathcal{S}_N + \Big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N + \mathit{span}_O \times (m_N - 1)}{m} - \mathit{span}' \times \frac{m_N}{m} + \mathit{span}'\Big) \\
&= \mathcal{S}_N + \Big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N + \mathit{span}_O \times (m_N - 1)}{m} + \mathit{span}' \times \big(1 - \frac{m_N}{m}\big)\Big) \qquad (5)
\end{aligned}$$

Since $m_N \leq m$, Expression 5 is maximized when *span*$'$ is large as possible; i.e., $\mathit{span}' = \mathit{span}_O$ (the physical interpretation is that the worst case occurs when no job on the critical path is executed prior to time-instant $\mathcal{S}_N$: instead the entire critical path executes after $\mathcal{S}_N$). Substituting $\mathit{span}' \leftarrow \mathit{span}_O$ into Expression 5, we get the following upper bound on the overall makespan:

$$\begin{aligned}
&\mathcal{S}_N + \Big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N + \mathit{span}_O \times (m_N - 1)}{m} + \mathit{span}_O \times \big(1 - \frac{m_N}{m}\big)\Big) \\
&= \mathcal{S}_N + \big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N - \mathit{span}_O}{m} + \mathit{span}_O\big)
\end{aligned}$$

Correctness is guaranteed by having this upper bound on the makespan be $\leq D$:

$$\begin{aligned}
&\Big(\mathcal{S}_N + \big(\frac{\mathit{work}_O - \mathcal{S}_N \times m_N - \mathit{span}_O}{m} + \mathit{span}_O\big)\Big) \leq D \\
\Leftrightarrow\ &\Big(\mathcal{S}_N - \frac{\mathcal{S}_N \times m_N}{m}\Big) \leq \Big(D - \frac{\mathit{work}_O - \mathit{span}_O}{m} - \mathit{span}_O\Big) \\
\Leftrightarrow\ &\mathcal{S}_N\Big(1 - \frac{m_N}{m}\Big) \leq \Big(D - \frac{\mathit{work}_O - \mathit{span}_O}{m} - \mathit{span}_O\Big) \qquad (6)
\end{aligned}$$

Expression 6 above is thus the sufficient schedulability condition we seek: values of m_N and $\mathcal{S}_N$ satisfying Expression 6 guarantee timing correctness.

4.2 Computing m_N and $\mathcal{S}_N$

We saw in Section 4.1 above that in order to ensure correctness, our scheduling algorithm should choose the parameters m_N and $\mathcal{S}_N$ such that Condition 6 above is satisfied. Recall that an additional goal is *efficiency*: the smaller the value of m_N, the better, since the remaining $(m - m_N)$ processors can be placed in sleep mode. In this section we describe how our algorithm computes such a value.

One reasonable approach for assigning a value to the m_N parameter is by using the task's *nominal* work and span parameters work_N and span_N. Assuming that these parameters bound the work and span values of a "typical" invocation of the task, it is guaranteed by Inequality 1 that upon m_N processors a typical invocation will have a makespan no greater than $\big((\mathit{work}_N - \mathit{span}_N)/m_N + \mathit{span}_N\big)$. We may hence assign $\mathcal{S}_N$ a value as follows:

$$\mathcal{S}_N \leftarrow \Big(\frac{\mathit{work}_N - \mathit{span}_N}{m_N} + \mathit{span}_N\Big) \tag{7}$$

Substituting this value for $\mathcal{S}_N$ into Expression 6, we get

$$\Big(\frac{\mathit{work}_N - \mathit{span}_N}{m_N} + \mathit{span}_N\Big) \times \Big(1 - \frac{m_N}{m}\Big) \le \Big(D - \frac{\mathit{work}_O - \mathit{span}_O}{m} - \mathit{span}_O\Big) \tag{8}$$

as a sufficient schedulability condition. Since every term other than m_N is a constant in this expression, the expression can be algebraically simplified to a form that is a quadratic expression in m_N; solving this quadratic expression, and taking the ceiling (since the number of processors m_N must be integral) yields the desired value. Once m_N is so computed, the value of $\mathcal{S}_N$ may be obtained from Expression 7. We illustrate via an example; the algorithm for computing m_N and $\mathcal{S}_N$ is provided in pseudo-code form after the example.

▶ **Example 4.** Consider once again the instance discussed in Examples 1 and 2:

$$\Big\langle \mathit{work}_O, \mathit{span}_O, \mathit{work}_N, \mathit{span}_N, D\Big\rangle \quad = \quad \Big\langle 900, 600, 120, 40, 690\Big\rangle$$

to be scheduled upon $m = 10$ processors.

Substituting these values into Expression 8, we get

$$\begin{aligned}
&\Big(\frac{\mathit{work}_N - \mathit{span}_N}{m_N} + \mathit{span}_N\Big) \times \Big(1 - \frac{m_N}{m}\Big) \le \Big(D - \frac{\mathit{work}_O - \mathit{span}_O}{m} - \mathit{span}_O\Big)\\
\equiv\quad & \Big(\frac{120-40}{m_N} + 40\Big) \times \Big(1 - \frac{m_N}{10}\Big) \le \Big(690 - \frac{900-600}{10} - 600\Big)\\
\equiv\quad & \Big(\frac{80}{m_N} + 40\Big) \times \Big(1 - \frac{m_N}{10}\Big) \le 60\\
\equiv\quad & 40 \cdot \Big(\frac{2}{m_N} + 1\Big) \times \Big(1 - \frac{m_N}{10}\Big) \le 60\\
\equiv\quad & 2 \cdot \Big(\frac{2+m_N}{m_N}\Big) \times \Big(\frac{10-m_N}{10}\Big) \le 3\\
\equiv\quad & (2+m_N) \times (10-m_N) \le 15m_N\\
\equiv\quad & 20 + 8m_N - m_N^2 \le 15m_N\\
\equiv\quad & m_N^2 + 7m_N - 20 \ge 0
\end{aligned}$$

from which we obtain

$$m_N \ge \frac{-7+\sqrt{129}}{2} \;\approx\; 2.18$$

Since the number of processors must be integral, we conclude that $m_N \leftarrow 3$. The corresponding value for $\mathcal{S}_N$ is equal to

$$\Big(\frac{\mathit{work}_N - \mathit{span}_N}{m_N} + \mathit{span}_N\Big) = \Big(\frac{120-40}{3} + 40\Big) = 26\frac{2}{3} + 40 = 66\frac{2}{3}$$

Algorithm 1: Computing values for $m_N, \mathcal{S}_N$.

```
Input: $\big(\langle \text{work}_O, \text{span}_O, \text{work}_N, \text{span}_N, D\rangle, m\big)$
Output: $\textit{failure}$, or values for $m_N$, $\mathcal{S}_N$
begin
    if $\big(m < \lceil(\text{work}_O - \text{span}_O)/(D - \text{span}_O)\rceil\big)$ then
        return $(\textit{failure})$   /* The test of Inequality 1 cannot guarantee that
          the deadline will be met on $m$ processors */
    end
    $A \leftarrow \text{span}_N$
    $B \leftarrow m \times (D - (\text{span}_O + \text{span}_N)) - (\text{work}_O - \text{span}_O) + (\text{work}_N - \text{span}_N)$
    $C \leftarrow (-1) \times m \times (\text{work}_N - \text{span}_N)$
    $m_N \leftarrow \big\lceil(-1 \times B + \big(\sqrt{B^2 - 4 \times A \times C}\big)/(2 \times A))\big\rceil$
    $\mathcal{S}_N \leftarrow \text{span}_N + (\text{work}_N - \text{span}_N)/m_N$
    return$(m_N, \mathcal{S}_N)$
end
```

Pseudo-code representation. It may be verified that Expression 8 can be rewritten to be of the form

$$A \times m_N^2 + B \times m_N + C \geq 0$$

with A, B, and C assigned the following values:

$$\begin{aligned} A &\leftarrow \text{span}_N \\ B &\leftarrow \Big(m\big(D - (\text{span}_O + \text{span}_N)\big) - (\text{work}_O - \text{span}_O) + (\text{work}_N - \text{span}_N)\Big) \\ C &\leftarrow -1 \times m \times (\text{work}_N - \text{span}_N) \end{aligned}$$

The pseudo-code in Algorithm listing 1 finds the positive root of this quadratic inequality; the ceiling of which denotes the number m_N of processors needed – this computation occurs in Line 8. In Line 9 the value computed for m_N is used to determine the value to be assigned to $\mathcal{S}_N$.

Run-time complexity. Algorithm 1 comprises straight-line code with no loops or recursive calls. Hence given as input the parameters specifying a task, it is evident that Algorithm 1 has constant – $\Theta(1)$ – run-time.

5 Achieving Greater Efficiency: A More Aggressive Approach

In an attempt to achieve efficiency (reducing the number of processors used in the "common case") while maintaining correctness (guaranteeing to meet deadlines provided task behavior does not exceed the worst-case bounds of $\textit{work}_O$ and $\textit{span}_O$), the approach derived in Section 4.2 above uses the nominal parameter values $\textit{work}_N$ and $\textit{span}_N$ to assign values to m_N and $\mathcal{S}_N$. In this section, we propose a more aggressive approach to achieving perhaps greater efficiency without compromising correctness in any manner. This more aggressive approach is based upon exploiting insights regarding(i) the probabilistic characterization of the run-time behavior of the system; and (ii) the typical behavior of List Scheduling.

The probabilistic characterization of run-time behavior. As discussed in Section 2 (and illustrated in Examples 1 and 2), we may have a *probability* associated with the likelihood that the work_N and span_N parameter values are correct. We may, for example be able to assert that there is a ≤ 0.05 probability that the actual work will exceed work_N, and a ≤ 0.05 probability that the actual span will exceed span_N. Now this threshold probability of 0.05 may have been selected because we desire that the probability that the $(m - m_N)$ sleeping processors will need to be awakened be ≤ 0.05. If so, the method for computing $\mathcal{S}_N$ and m_N described in Section 4.2 above may be overly conservative since the work and span distributions may not be perfectly correlated – if they are not, the probability that both the work would exceed work_N **and** the span exceed span_N, during a particular execution of the task is smaller than 0.05. (In the extreme if the two distributions are more or less independent, the probability is closer to 0.05^2 which equals .0025, a value that is is far smaller than the sought-for threshold probability of 0.05.)

Some observations on List Scheduling. Assuming that the actual work and span parameters of the computation do not exceed work_N and span_N respectively, in Section 4.2 we used Expression 1 to assign values to m_N and $\mathcal{S}_N$ in a manner guaranteeing that the computation will complete execution within an interval of duration $\mathcal{S}_N$ upon m_N processors. Note that Expression 1 is an *upper bound* on the makespan of a List Scheduling generated schedule of a DAG; this upper bound is tight only for DAGs possessing a very specific structural form and/ or List Scheduling making a particular sequence of scheduling decisions (and then only if each node of the DAG executes for its entire WCET). Simulation experiments using randomly-generated graphs seem to indicate that these structures and scheduling decisions are relatively rare; for randomly-generated graphs, the makespans of actual list-scheduling generated schedules tend to cluster closer towards the lower end of the interval between the upper bound of Expression 1 and the obvious lower bound of

$$\max\left(\frac{\mathit{work}}{m}, \mathit{span}\right) , \tag{9}$$

even if each node of the DAG does actually execute for its entire WCET. To illustrate this, we randomly generated 1000-node DAGs with varying numbers of edges in the manner described in Section 5.1 below; for each, we computed the lower bound of Expression 9, the actual makespan using a list scheduling implementation, and the upper bound of Expression 1, for scheduling the DAG upon a 10-processor platform. The results are listed in Table 1. The right-most column – the one titled “Ratio” – denotes the fraction of the interval between the lower bound and the upper bound upon which the actual makespan encroaches.

More aggressive computation of m_N and $\mathcal{S}_N$. We highlight the fact that being too optimistic in assigning values to $\mathcal{S}_N$ and m_N does *not* compromise correctness: the sole effect is upon efficiency in terms of the number of processors we are able to maintain in sleep mode, and the likelihood that these processors will need to be switched on during some run of the system. Hence one possible –more aggressive– approach towards achieving greater resource efficiency during run-time would be to assign $\mathcal{S}_N$ a value between the lower and upper bounds of Expressions 9 and 1 as follows (rather than according to Expression 7):

$$\mathcal{S}_N \leftarrow \max\left(\frac{\mathit{work}_N}{m_N}, \mathit{span}_N\right) + \alpha\cdot\left[\left(\frac{\mathit{work}_N - \mathit{span}_N}{m} + \mathit{span}_N\right) - \max\left(\frac{\mathit{work}_N}{m_N}, \mathit{span}_N\right)\right] \tag{10}$$

with $\alpha, 0 \leq \alpha \leq 1$ a “tuning” parameter: the smaller the value of α, the more aggressive the choice of $\mathcal{S}_N$. (An intuitive interpretation of the tradeoff here is that the smaller the

Table 1 Actual makespan, and lower and upper bounds, of randomly-generated DAGs upon a 10-processor platform. Each graph has 1000 vertices; each row corresponds to 1000-vertex DAGs with the number of edges specified in the first column. The last column denotes the ratio (actual makespan - lower bound) ÷ (upper bound - lower bound) – small values denote that the actual makespan is close to the lower bound.
(The experiments used to generate this table are detailed in Section 5.1.)

	MAKESPAN			
# edges	**Lower (Exp. 9)**	**Actual**	**Upper (Exp. 1)**	**Ratio**
977	2627	2667	2818	0.208
2017	2539	2587	2889	0.137
4921	2567	2603	3222	0.055
9935	2554	2709	3725	0.132
20094	2599	2977	4774	0.174
39935	4056	4113	6154	0.027
50036	4454	4480	6491	0.013
60212	5674	5674	7658	0.000

value of α, the greater the number of processors we can switch off, but the greater the likelihood that they will need to be awakened during some execution of the task.) For the randomly-generated DAGs of Table 1, a value of $\alpha \geq 0.208$ would have been safe: during run-time the sleeping processors are not awakened as long as the task's run-time behavior does not violate its nominal parameters $work_N$ and $span_N$.

If $\mathcal{S}_N$ is assigned a value according to Expression 10 rather than Expression 7, it is no longer the case that solving Expression 8 yields the desired value of m_N. We have not attempted to derive a closed-form solution for m_N when Expression 10 is used in place of Expression 7; rather, we iterate through candidate values for m_N over the range $[1, m)$, stopping at the first such value for which this value for m_N, and the resulting value for $\mathcal{S}_N$ computed according to Expression 10, causes Condition 6 to evaluate to true. This more aggressive approach to computing m_N and $\mathcal{S}_N$ therefore has run-time complexity $\Theta(m)$ where m denotes the number of processors available; a straightforward application of the idea of binary search reduces this to $\Theta(\log m)$.

5.1 The Experiments Reported in Table 1

We now briefly describe the experimental procedure used to generate the data populating Table 1. Graphs were synthesized using a DAG-generating variant of the well-known Erdös-Rényi method [26] for generating random graphs. The Erdös-Rényi method, given parameters (n, p), yields a graph on n vertices in which each edge has an independent probability p of existence. We modified this method to generated *directed acyclic* graphs with a target number of edges. Specifically,

- The number of vertices in the DAG, n, the maximum WCET parameter for a vertex w, and the desired number of edges e, are specified. The number of processors m upon which the DAG is to be scheduled is also specified.
- Each vertex is assigned a WCET parameter that is a randomly and uniformly drawn integer over the range $(1, w)$.

- The parameter p, denoting the probability of existence of each edge, is computed as follows:

$$p \leftarrow \frac{2e}{n \times (n-1)} \ .$$

  The idea is that since a DAG with n vertices has a maximum of $n/ \times (n-1)/2$ edges, if we were to create each such edge independently with probability p, on average the desired number e of edges would be created.
- All edges are assumed to be directed from the lower-indexed vertex to the higher-indexed vertex (hence a topological sorting of the DAG could yield the vertices in order of increasing index). Each such edge is created with probability p, as follows:

```
for i := 1 to n
   for j := (i+1) to n
     create edge (i,j) with probability p
```

- The work and span parameter of the generated DAG are computed, assuming that each vertex executes for exactly its WCET parameter value (i.e., WCET parameters are taken to represent the actual execution duration, rather than an upper bound on the execution duration). Using these values, a lower bound on makespan as given by Expression 9, and an upper bound as given by Expression 1, for the DAG upon the specified number m of processors are computed.
- A schedule of the DAG upon m processors using a standard implementation of list scheduling is generated (once again assuming that each vertex executes for exactly its WCET parameter value). The makespan of the resulting schedule is recorded.
- Each data-point reported in Table 1 was obtained by generating one hundred such graphs, computing the reported parameters upon each, and taking their averages.

6 Summary and Conclusions

Although DAG-based models for representing parallelizable real-time code have proved very popular in the real-time scheduling theory community, they suffer from several shortcomings that restrict their usefulness in representing some kinds of real-time code. In this paper, we have explored an alternative model, one that is based upon characterizing a task by just two parameters – work and span – with two estimates on upper bounds on the value of each parameter – one that may be very large but is trust-worthy to a very high level of assurance, and a second that is smaller and is more reflective of typical or nominal behavior. We have developed an algorithm for scheduling tasks that are so modeled upon a dedicated cluster of processors in a manner guaranteeing *correctness* – deadlines are always met provided run-time behavior does not violate the high-assurance bounds – while striving for *efficiency* – many processors can remain in sleep mode much of the time, only being switched on in rare circumstances when run-time behavior exceeds the normal bounds.

The model for representing parallelizable code that is being proposed in this paper, and the associated run-time scheduling algorithm, is particularly suitable for a certain kind of real-time application: one that repeatedly (i.e., periodically or sporadically) monitors the external environment seeking to detect some particular kind of anomalous sensory input. Most of the time the sought-for input is not detected, and not much computation needs to be performed. But on the rare occasions when the anomalous input is detected, considerable additional processing of such input is necessary; furthermore, such processing is highly parallel in nature. Example applications of this kind include real-time intrusion detection, vision-based monitoring systems, etc. For such systems, we would expect that the nominal workload, as represented by the $work_N$ and $span_N$ parameters, is quite small and may not exhibit much parallelism; however the workload upon "overload" (i.e., when the monitored-for condition occurs) is quite intensive ($work_O$ is large) but exhibits considerable parallelism (i.e., $span_O$ is relatively small compared to $work_O$: equivalently, the ratio $work_O/span_O$ is large). If the system is hard-real-time, resource allocation for guaranteeing correctness must be made under worst-case assumptions – the $work_O$ and $span_O$ parameters. However, much of these allocated resources will remain unused much of the time during run-time; by being able to determine in a timely manner precisely when these unused resources will be needed during run-time. our approach allows us to place these resources in sleep mode until needed.

We believe our main contribution here is the model for parallel tasks – the run-time scheduler is presented as proof-of-concept evidence of the potential benefits, in terms of resource-efficiency, of adopting this model. As future work we plan to demonstrate the model's applicability in a wider range of settings: under different scheduling paradigms (such as global EDF and global Fixed-Priority). We are also working on further extending the task model if additional profiling data of the task's run-time behavior is available (and known to be reliable). For example, straight-forward generalizations allow us to specify multiple sets of parameter values at different probability thresholds (rather than just two sets of values) – are we able to develop scheduling strategies that can meaningfully exploit such additional information?

References

1 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS*, pages 259–268. IEEE Computer Society, 2010.

2 Bjorn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2012.

3 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leem Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS 2012, pages 63–72, San Juan, Puerto Rico, 2012.

4 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer Publishing Company, Incorporated, 2015.

5 Jose Fonseca, Vincent Nelis, Gurulingesh Raravi, and Luis Miguel Pinho. A Multi-DAG model for real-time parallel applications with conditional execution. In *Proceedings of the ACM/ SIGAPP Symposium on Applied Computing (SAC)*, Salamanca, Spain, April 2015. ACM Press.

6 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 222–231, Lund (Sweden), 2015. IEEE Computer Society Press.

7 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 222–231, Lund (Sweden), 2015. IEEE Computer Society Press.

8 Robert I. Davis, Alan Burns, and David Griffin. On the meaning of pWCET distributions and their use in schedulability analysis. In *Proceedings 2017 Real-Time Scheduling Open Problems Seminar (RTSOPS)*, 2017.

9 Alan Burns and Robert Davis. Mixed-criticality systems: A review (9th edition). `http://www-users.cs.york.ac.uk/~burns/review.pdf` (Accessed on Aug 29, 2017), 2017.

10 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

11 Sanjoy Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *Proceedings of the 15th International Conference on Embedded Software (EMSOFT)*, Amsterdam, the Netherlands, 2015.

12 James Anderson, Sanjoy Baruah, and Bjoern Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, San Francisco, CA, April 2009.

13 Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. In *Proceedings of the International Workshop on Mixed Criticality Systems (WMC)*, December 2014.

14 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017.

15 R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Mathematics*, 5:287–326, 1979.

16 J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

17 R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

18 J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

19 Ola Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 745–754, New York, NY, USA, 2010. ACM.

20 S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *2001 IEEE Real-Time Systems Symposium (RTSS)*, pages 215–224, Dec 2001.

21 G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *2002 IEEE Real-Time Systems Symposium (RTSS)*, pages 279–288, 2002.

22 G. Bernat, A. Colin, and S. Petters. pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, The University of York, England, 2003.

23 Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

24 Jing Li, Abusayeed Saifullah, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems*, ECRTS '14, Madrid (Spain), 2014. IEEE Computer Society Press.

25 M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2015 IEEE*, pages 305–316, Dec 2015.

26 P. Erdös and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

Efficiently Approximating the Probability of Deadline Misses in Real-Time Systems

Georg von der Brüggen
Department of Computer Science, TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de
https://orcid.org/0000-0002-8137-3612

Nico Piatkowski
Department of Computer Science, TU Dortmund University, Germany
nico.piatkowski@tu-dortmund.de
https://orcid.org/0000-0002-6334-8042

Kuan-Hsun Chen
Department of Computer Science, TU Dortmund University, Germany
kuan-hsun.chen@tu-dortmund.de
https://orcid.org/0000-0002-7110-921X

Jian-Jia Chen
Department of Computer Science, TU Dortmund University, Germany
jian-jia.chen@cs.uni-dortmund.de
https://orcid.org/0000-0001-8114-9760

Katharina Morik
Department of Computer Science, TU Dortmund University, Germany
katharina.morik@tu-dortmund.de
https://orcid.org/0000-0003-1153-5986

Abstract

This paper explores the probability of deadline misses for a set of constrained-deadline sporadic soft real-time tasks on uniprocessor platforms. We explore two directions to evaluate the probability whether a job of the task under analysis can finish its execution at (or before) a testing time point t. One approach is based on analytical upper bounds that can be efficiently computed in polynomial time at the price of precision loss for each testing point, derived from the well-known Hoeffding's inequality and the well-known Bernstein's inequality. Another approach convolutes the probability efficiently over multinomial distributions, exploiting a series of state space reduction techniques, i.e., pruning without any loss of precision, and approximations via unifying equivalent classes with a bounded loss of precision. We demonstrate the effectiveness of our approaches in a series of evaluations. Distinct from the convolution-based methods in the literature, which suffer from the high computation demand and are applicable only to task sets with a few tasks, our approaches can scale reasonably without losing much precision in terms of the derived probability of deadline misses.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases deadline miss probability, multinomial-based approach, analytical bound

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.6

Related Version A full version of this paper is available at [27],
`https://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2018-brueggen-ecrts-deadline-miss-probability.pdf`.

Funding This paper is supported by DFG, as part of the Collaborative Research Center SFB876 (`http://sfb876.tu-dortmund.de/`), project A1 and project B2.

1 Introduction

For many embedded systems, timeliness is an important feature, especially when such systems interact with physical environments. A stronger requirement of timeliness is to provide *hard* real-time guarantees, i.e., to ensure that the calculated results are not just functionally correct but also *always* delivered within given timing constraints. Such hard guarantees are necessary if any deadline miss can be catastrophic and should be avoided. By contrast, a weaker requirement of timeliness is to allow occasional deadline misses, called *soft* real-time systems. In this case the system can still function correctly as long as the deadline misses can be quantified and bounded. For example, the system may adopt fault tolerance techniques like checkpointing, redundant execution, etc. [13, 20, 23, 28, 19], to neglect transient faults resulting from electromagnetic interference and radiation [3]. Although the additional computation incurred by such methods may lead to deadline misses, the system may still provide timing guarantees even without any online adaption [26]. A second example are the safety standards in the industry that require low (or very low) probability of failure (e.g., due to deadline misses) such as IEC-61508 [1] and ISO-26262 [14].

Probability theory is a basic language to describe probabilistic phenomenons, e.g., occasional deadline misses. It is based on the idea that most natural phenomena are either too complex to construct deterministic models or simply not fully observable but can be described in a probabilistic way. For example, we can establish probabilistic bounds on the worst-case execution times (WCETs) to model the execution of a task depending on the occurrence of soft errors and the triggered error recovery routines. This allows the system designer to provide probabilistic arguments based on the occurrence of error recovery. Otherwise, only the WCET, assuming that the recovery always takes place, has to be considered in the response time analysis, which is very pessimistic and therefore leads to overestimating the necessary system resources.

Probability of Deadline Misses. A key procedure needed for such soft real-time systems is the analysis of the probability of deadline misses for a real-time task. Now, we take a closer look of the problem by using the following example: Suppose that we have two periodic tasks τ_1 and τ_2 that release task instances, called jobs, periodically, starting from time 0. Each task $\tau_i \in \{\tau_1, \tau_2\}$ has two versions of execution times $C_{i,1}$ and $C_{i,2}$ with probability $\mathbb{P}_i(1)$ and $\mathbb{P}_i(2)$, respectively. The period of task τ_1 is 1 and the period of task τ_2 is 100. We assume that task τ_1 always has a higher priority than task τ_2 and task τ_1 can always meet its deadline under a fixed-priority preemptive scheduling strategy in a uniprocessor system.

In this example, the system reboots if a job of task τ_2 is not finished before the next job of task τ_2 is released. Therefore, the probability of deadline misses corresponds to the probability of system rebooting. Essentially, we are interested to know whether a job of τ_2, arriving at time t_a, can finish its execution before $t_a + 100$. This can be achieved by the *convolution* of the probability density functions of the jobs' execution times. An intuitive procedure is to evaluate the probability of the accumulative execution time, denoted as *workload*, of the jobs released from time t_a to $t_a + \ell - 1$ (inclusive), starting from $\ell = 1, 2, 3, \ldots, 100$. When ℓ is 1, we have 2^2 combinations of the workload of the two jobs released at time t_a. When ℓ is 2, we can have up to $2^2 \times 2 = 2^3$ combinations of the workload. It is rather obvious that we can have up to 2^{101} combinations of the workload when ℓ is 100, which is *exponential* with respect to the number of jobs that may interfere with a job of task τ_2.

Since there are only two versions of task τ_1, there are in fact only $\ell + 1$ different workload combinations of the ℓ jobs released from time t_a to time $t_a + \ell - 1$. As a result, there are only $2(\ell + 1)$ different workload combinations of the jobs released from time t_a to $t_a + \ell - 1$. We can evaluate all of them from $\ell = 1, 2, \ldots, 100$. However, this remains inefficient as we are only interested in the probability of the deadline miss at time $t_a + 100$. For this example, we do not actually care about the individual execution versions of the 100 jobs of task τ_1 released from t_a to $t_a + 99$. Instead, we only care about their overall workload, which can be calculated by using a binomial distribution over 100 independent random variables with the same distribution. As a result, we only have to consider 101 different workload combinations for the jobs of τ_1. Together with the job of task τ_2, there are in fact only 2×101 different workload combinations.

These approaches are different realizations of the same concept to convolute the probability density functions of the jobs' execution times. However, depending on how the convolution is performed, the complexity can differ largely.

Related Work. As explained above for uniprocessor systems, it is necessary to safely derive (an upper bound on) the probability of a desired workload constraint to analyze the probability of deadline misses or the probabilistic response time. Towards this, for periodic real-time task systems, Diaz et al. [9] developed a framework for calculating the deadline miss probability based on convolution. Moreover, Tanasa et al. [24] used the Weierstrass Approximation to approximate any arbitrary execution time distributions and applied a customized decomposition procedure to search all the possible combinations, in which the decomposition results in a list with $O(4^{|J|})$ elements where $|J|$ is the number of jobs in the interval of interest. These two results have exponential-time complexity with respect to the number of jobs in the interval of interest. Therefore, both of them suffer from the scalability with respect to the number of jobs. In the experimental results in [9] and [24], they can derive the probability of deadline misses with 7 and 25 jobs in the hyper-period, respectively.

For sporadic real-time task systems, in which two consecutive jobs of a task do not have to be released periodically, Axer et al. [2] proposed to evaluate the response-time distribution and iterate over the activations of job releases for non-preemptive fixed-priority scheduling. Maxim et al. [17] provided a probabilistic response time analysis by assuming probabilistic minimum inter-arrival as well as probabilistic worst-case execution times for the fixed-priority scheduling policy. Ben-Amor et al. [4] extended the probabilistic response time analysis in [17], considering precedence constrained tasks. All these approaches convolute the probability whenever a new job arrives in the interval of interest. Therefore, the convolution procedure is also heavily dependent on the number of jobs in the interval of interest.

Due to the high complexity, these convolution-based approaches are not scalable with respect to the number of jobs in the interval of interest and, thus, infeasible. Approximation techniques can be used to provide an upper bound on the probability. For example, re-sampling [17] and dynamic-programming based on user-defined granularity can be applied to reduce the time complexity. Moreover, Chen and Chen [8] provided a scalable approximation based on the Chernoff bounds. The evaluation results in [8] confirm the applicability and the scalability of such approximations, even when considering 20 tasks and more than thousand jobs in the hyper-period.

Our Contributions. We consider the problem of determining the deadline miss probability of a task under uniprocessor fixed-priority preemptive scheduling when each task has distinct execution modes that are executed with a known probability distribution. Our main contributions are:

- We provide a novel approach based on the multinomial distribution that, compared to the traditional convolution-based approach, allows to calculate the deadline miss probability with better analysis runtime and without any precision loss.
- The analysis is enhanced by a state pruning technique that significantly improves the runtime as well as the scalability without any loss of precision.
- We further improve our approach by merging equivalence classes, thus further reducing the runtime of our analysis while the introduced precision loss can be bounded in advance.
- In the evaluation, we show that our approach is applicable for significantly larger task sets than the previously known convolution-based approaches by testing it for task sets of up to 100 tasks.
- Furthermore, we provide additional analytical bounds based on the Hoeffding's [12] and Bernstein's [11] inequalities. Our evaluations show that these inequalities lead to fast results and can be used if the over-approximation is acceptable.

2 Task Model, System Model, and Notation

We consider a given set of n independent periodic (or sporadic) tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ in a uniprocessor system. Each task τ_i releases an infinite number of task instances, called jobs, and is defined by a tuple $((C_{i,1}, ..., C_{i,h}), D_i, T_i)$, where D_i is the relative deadline of τ_i and T_i is its minimum interarrival time. In addition, each task has a set of h distinct execution modes $\mathcal{M}$ and each mode j with $j \in \{1, ..., h\}$ is associated with a different worst-case execution time (WCET) $C_{i,j}$. We assume those execution modes to be ordered increasingly according to their WCETs, i.e., $C_{i,m} \leq C_{i,m+1}$ $\forall m \in \{1, ..., h-1\}$. Furthermore, we assume that each job of τ_i is executed in one of those distinct execution modes. To fulfill its timing requirements in the j^{th} execution mode, a job of τ_i that is released at time t_a must be able to execute $C_{i,j}$ units of time before $t_a + D_i$. The next job of τ_i must be released at $t_a + T_i$ for a periodic task and for a sporadic task the next job is released at or after $t_a + T_i$. In this work, we focus on *implicit-deadline* task sets, i.e., $D_i = T_i$ for all tasks, and *constrained-deadline* task sets, i.e., $D_i \leq T_i$ for all tasks. The task set is assumed to be scheduled according to a preemptive fixed-priority scheduling policy, i.e., each task has a unique fixed priority, the priority cannot be changed during runtime, and the priority of each task instance is identical to the priority of the related task. At each point in time, the scheduler ensures that the job with the highest priority, among the jobs currently ready in the system, is executed. We assume that the tasks are indexed according to their priority, i.e., τ_1 has the highest and τ_n has the lowest priority. In addition, $hp(\tau_k)$ denotes the set of tasks with higher priority than τ_k and $hep(\tau_k)$ is $hp(\tau_k) \cup \{\tau_k\}$. For a task τ_i in $hp(\tau_k)$, $\rho_{i,t}$ is the maximum number of jobs that are released in an interval $[0, t)$, also called the interval of interest, and therefore interfere with task τ_k, i.e., the number of jobs released in the interval $[0, t)$ under the critical instance of τ_k. Furthermore, $\rho_{k,t}$ is the number of jobs of task τ_k in the analysis window. This notation implicitly assumes that the time window analyzed for τ_k starts at 0 for notational brevity. $\mathbb{P}_i(j)$ denotes the probability that a job of task τ_i is executed in mode j with related WCET $C_{i,j}$ and we assume that each job is executed in exactly one of these distinct execution modes, i.e., $\sum_{j=1}^{h} \mathbb{P}_i(j) = 1$. In addition, we assume that these probabilities are independent from each other according to the following definition:

▶ **Definition 1** (Independent Random Variables). *Two random variables are (probabilistically) independent if the realization of one does not have any impact on the probability of the other.*

Especially, for a newly arriving job the probability of the execution modes is independent from the execution mode of the jobs currently in the system or of previous jobs. We aim

Table 1 Important notation used in this work. Please note that not all explanations in this table are precise. The precise notations can be found in the Section indicated in the table.

	Task-related Quantities	
$\tau_i = ((C_{i,1},...,C_{i,h}), D_i, T_i)$	Task τ_i and related WCETs $(C_{i,1},...,C_{i,h})$, deadline D_i, and period T_i	Sec. 2
$(C_{i,1},...,C_{i,h})$	WCET of the h different execution modes of τ_i	Sec. 2
$\mathbb{P}_i(j)$	Probability that a job of $\tau_i s$ is executed in mode j with related WCET $C_{i,j}$	Sec. 2
$\mathcal{M}$	Set of the possible execution modes (assumed identical for all tasks). $\lvert\mathcal{M}\rvert = h$	Sec. 2
$hp(\tau_k)$ and $hep(\tau_k)$	Tasks with higher priority than τ_k (higher and equal priority, respectively)	Sec. 2
$\rho_{i,t} = \lceil t/T_i \rceil$	Maximum number of jobs of τ_i released in an interval $[0,t)$ under the critical instant	Sec. 2
$J(t) = \sum_{\tau_i \in hep(\tau_k)} \lceil t/T_i \rceil$	Total number of jobs released in the interval $[0,t)$	Sec. 5.1
S_t	Maximum accumulated workload over an interval of length t	Sec. 3.1
	Probabilistic Quantities	
Φ_k	Probability of deadline miss for task τ_k	Sec. 3.1
$\mathbb{P}(S_t > t)$	Probability of overload for an interval of length t	Sec. 3.1
$\bar{X}$	Arithmetic mean of a random variable X	Sec. 4
$\mathbb{E}[X]$	Expected value of a random variable X	Sec. 4
$\mathbb{V}[X]$	Variance of a random variable X	Sec. 4
$\boldsymbol{X}(t)$	Random variable representing the possible execution modes of all jobs in $[0,t)$	Sec. 5.1
$\mathcal{X}(t)$	The state space of $\boldsymbol{X}(t)$ with $\mathcal{X}(t) = \mathcal{M}^{J(t)}$ since all jobs are considered	Sec. 5.1
$\boldsymbol{x} \in \mathcal{X}(t)$	One concrete variable assignment for $\boldsymbol{X}(t)$ over $[0,t)$	Sec. 5.1
$\mathbb{P}(\boldsymbol{X}(t) = \boldsymbol{x})$	Probability that the state space $\boldsymbol{X}(t)$ has the concrete variable assignment $\boldsymbol{x}$	Sec. 5.1
$\boldsymbol{X}_i(t)$	Subset of random variables in $\boldsymbol{X}(t)$) that relate to τ_i	Sec. 5.2
$C_i(\boldsymbol{X}_{i,j}(t))$	WCET for the j^{th} job of τ_i based on its random execution mode $\boldsymbol{X}_{i,j}(t)$	Sec. 5.1
	Combinatorial Quantities	
$\mathbb{1}_{\{\text{expression}\}}$	Indicator function, i.e., evaluates to 1 iff the expression is true, and 0 otherwise	Sec. 5.1
$\sigma(\boldsymbol{x})$	A permutation of $\boldsymbol{x}$	Sec. 5.1
$\mathbb{S}_n$	Set of all permutations of length n	Sec. 5.1
$[\![\boldsymbol{x}]\!]$	Equivalence class of $\boldsymbol{x}$, i.e., all $\boldsymbol{x}' \in \mathcal{X}(t)$ that can be permuted into $\boldsymbol{x}$	Sec. 5.1

at relaxing the independence assumptions on tasks and jobs in future work by employing techniques from the field of probabilistic graphical models [21, 22].

A list of our notation together with a brief explanation can be found in Table 1.

3 Motivation, Problem Definition, and State-of-the-Art

In this section, we will motivate the importance of the considered problem, i.e., the calculation of the probability of deadline misses, and formally define it. Afterwards, the state-of-the-art techniques are introduced, namely the traditional convolution-based approach by Maxim and Cucu-Grosjean [17] as well as the approach by Chen and Chen [8] that uses Chernoff bounds and the moment-generating function. We use the term *traditional convolution-based approach* when referring to the approach by Maxim and Cucu-Grosjean to avoid confusion, since our novel approach based on multinomial distributions also uses convolution.

3.1 Motivation and Problem Definition

One main assumption when considering real-time systems is that a deadline miss, i.e., a job that does not finish its execution before its deadline, will be disastrous and thus the WCET of each task is always considered during the analysis. Nevertheless, if a job has multiple distinct execution schemes, the WCETs of those schemes may differ largely. One example are software-based fault-recovery techniques as they rely on (at least partially) re-executing the faulty task instance. However, when such techniques are applied, the probability that a fault occurs and thus has to be corrected is very low; otherwise hardware-based faulty-recovery techniques would be applied. If such re-execution may happen multiple times, the resulting execution schemes have an increased related WCET while the probability decreases drastically. Therefore, considering solely the execution scheme with the largest WCET at design time would lead to largely over-designing the system resources. Furthermore, many real-time systems can tolerate a small number of deadline misses at runtime as long as these deadline misses do not happen too frequently. Hence, being able to predict the probability of a deadline miss is an important property when designing real-time systems. We will consider the probability of deadline misses for a single task here which is defined as follows:

▶ **Definition 2** (Probability of Deadline Misses). *Let $R_{k,j}$ be the response time of the j^{th} job of τ_k. The probability of deadline misses (DMP) of task τ_k, denoted by Φ_k, is an upper bound on the probability that a job of τ_k is not finished before its (relative) deadline D_k, i.e.,*

$$\Phi_k = \max_j \{\mathbb{P}(R_{k,j} > D_k)\}, \qquad j = 1, 2, 3, ... \tag{1}$$

It was shown in [17] that the DMP of a job is maximized when τ_k is released at its critical instant, i.e., together with a job of all higher priority tasks and all consecutive jobs of those higher priority tasks are released as early as possible. This implicitly assumes that no previous job has an overrun that interferes with the analyzed job. Hence, *time-demand analysis* (TDA) [16] can be applied to determine the worst-case response time of a task when the execution time of each job is known. TDA is an exact schedulability test for constrained and implicit deadline task sets with pseudo-polynomial runtime that, under the assumption that the schedulability of all higher priority tasks is already ensured, determines the schedulability of task τ_k by finding a point in time t where the total workload generated by tasks in $hep(\tau_k)$ is smaller than t. To be more precise: τ_k is schedulable if and only if

$$\exists\, t \text{ with } 0 < t \leq D_k \qquad \text{such that} \qquad S_t = C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq\, t \tag{2}$$

Thus, if $D_k \leq T_k$, task τ_k is schedulable if the statement $S_t \leq t$ is true. When probabilistic WCETs are considered, the WCET will obtain a value in $(C_{i,1}, ..., C_{i,h})$ with a certain probability $\mathbb{P}_i(j)$ for each job of each task τ_i. Therefore, for a given t we are not looking for a binary decision anymore. Instead, we are interested in the probability that the accumulated workload S_t over an interval of length t is at most t. The probability that τ_k cannot finish in this interval is denoted accordingly with $\mathbb{P}(S_t > t)$. We call the situation where S_t is larger than t an *overload* for an interval of length t and hence $\mathbb{P}(S_t > t)$ is the overload probability at time t. According to the previously introduced notation, $\rho_{i,t} = \lceil t/T_i \rceil$ for each task τ_i in $hp(\tau_k)$ and $\rho_{k,t} = 1$, i.e., only the first job of τ_k is considered here. Since TDA only needs to hold for one t with $0 < t \leq D_k$ to ensure that τ_k is schedulable, the probability that the test fails is upper bounded by the minimum probability among all time points at which the test could fail. Therefore, the probability of a deadline miss Φ_k can be upper bounded by

$$\Phi_k = \min_{0<t\leq D_k} \mathbb{P}(S_t > t) \tag{3}$$

The number of points considered in Eq. (2) and therefore in Eq. (3) can be reduced by only considering the *points of interest*, i.e., D_k and the releases of higher priority tasks. Nevertheless, in the worst case this still leads to a pseudo-polynomial number of points. Since the minimum value among all these points is taken, an upper bound will still be obtained when only a subset of those points is considered. Two approaches to calculate Φ_k are known from the literature and are summarized in the following subsections.

In some cases it is easier to determine $\mathbb{P}(S_t \geq t)$ instead of $\mathbb{P}(S_t > t)$, especially when analytical bounds are used (see Sec. 3.3 and Sec. 4). Since $\mathbb{P}(S_t \geq t) \geq \mathbb{P}(S_t > t)$ by definition, these values can be used directly when looking for an upper bound of $\mathbb{P}(S_t > t)$.

3.2 Traditional Convolution-Based Approaches

Each task is defined by a vector of the possible WCETs and the related probabilities, e.g., $\begin{pmatrix} 3 & 5 \\ 0.9 & 0.1 \end{pmatrix}$ where 3 and 5 are the WCETs and 0.9 and 0.1 are the related probabilities. The notation we use is similar to the one used by Maxim and Cucu-Grosjean in [17]. The convolution of two such vectors is denoted by $\otimes$ and results in a new vector. To determine this new vector, each element of the first vector is combined with each element of the second vector by 1) multiplying the related probabilities, and 2) summing up the related WCETs.

▶ **Example 3** (Convolution). $\begin{pmatrix} 3 & 5 \\ 0.9 & 0.1 \end{pmatrix} \otimes \begin{pmatrix} 5 & 6 \\ 0.8 & 0.2 \end{pmatrix} = \begin{pmatrix} 8 & 9 & 10 & 11 \\ 0.72 & 0.18 & 0.09 & 0.01 \end{pmatrix}$

Note that the summation of the probabilities is 1 for each of these vectors. The general idea of the traditional convolution-based approach [17] is the direct enumeration of the WCET state space[1] and the related probabilities. To this end, it considers the jobs in non-decreasing order of their arrival times. For each arriving job, the current system state, represented by a vector of possible states, i.e., possible total WCETs and related probability, is convoluted with the arriving job. This results in a new vector of possible states, representing the state space after the arrival of the job. After all jobs released before a certain time point are convoluted, the probability that the workload is smaller than the next arrival time of a job is calculated. Afterwards, the jobs arriving at that time are convoluted with the current states, and the probability for the next arrival time is checked etc. This process is repeated until $t = D_k$ is reached. A small example explaining the approach considering two tasks can be found in Figure 1. The first jobs of τ_1 and τ_2 are both convoluted with the initial state and the four resulting states are each convoluted with the second release of τ_1 at $t = 8$. Obviously, when all jobs that are released up to any point in time are convoluted, states that result in the same execution time can be combined by adding up the related probability, e.g., the states with WCET 13 and 14, respectively, in Figure 1.

On one hand, applying the traditional convolution-based approach can easily lead to a state explosion where the number of states is exponential in the number of jobs. On the other hand, it calculates the exact probabilities for each t in the interval of interest in one iteration. To tackle the problem of state explosion, Maxim and Cucu-Grosjean introduced a re-sampling approach to reduce the number of states to a given threshold and thus to reduce the runtime while only slightly decreasing the precision as shown in [17].

3.3 Chernoff-Bound-Based Approaches

Chen and Chen [8] use the *moment generating function* (*mgf*) in combination with the *Chernoff bound* to over-estimate the deadline miss probability. We only briefly introduce the techniques here, i.e., describe how they can be used in our setting. Details can be found in, e.g., [18]. The *mgf* of a random variable is an alternative way to specify its probability distribution. For the specific case of the WCET distribution of a task τ_i the *mgf* is $\text{mgf}_i(s) = \sum_{j=1}^{h} \exp(C_{i,j} \cdot s) \cdot \mathbb{P}_i(j)$ where *exp* is the exponential function, i.e., $\exp(x) = e^x$, and $s > 0$ is a given real number.

[1] Please note that the approach in [17] does not only consider probabilistic WCETs but also probabilistic periods. Since we only consider probabilistic WCETs here, the approach is summarized accordingly.

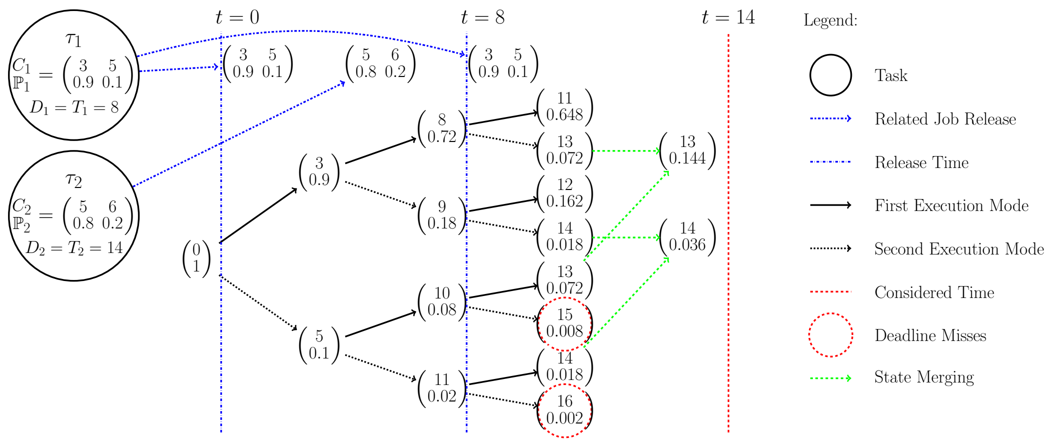


Figure 1 An example for the traditional convolution-based approach. Assume that $\mathbb{P}(S_{14} > 14)$ should be determined for two tasks τ_1 and τ_2. The initial state is convoluted with the two jobs released at $t = 0$ and the second job of τ_1 released at $t = 8$. Then, $\mathbb{P}(S_{14})$ is determined by summing up the probabilities of the states related to a workload larger than 14 (red dotted circle), leading to $\mathbb{P}(S_{14} > 14) = 0.01$. Note that states with the same execution time can be merged (dashed green arrows). This usually happens when the related paths are permutations of each other, e.g., both paths to 13 have one execution of $C_{1,1}$ and one of $C_{1,2}$.

The Chernoff bound can be exploited to over-approximate the probability that a random variable exceeds a given value. This statement is summarized in the following lemma:

▶ **Lemma 4** (Lemma 1 from Chen and Chen [8]). *Suppose that S_t is the sum of the execution times of the $\rho_{k,t} + \sum_{\tau_i \in hep(\tau_k)} \rho_{i,t}$ jobs in $hep(\tau_k)$ at time t. In this case*

$$\mathbb{P}(S_t \geq t) \leq min_{s>0} \left(\frac{\prod_{\tau_i \in hep(\tau_k)} (mgf_i(s))^{\rho_{i,t}}}{\exp(s \cdot t)} \right) \tag{4}$$

The *Chernoff bound* is in general pessimistic and there is no guarantee for the quality of the approximation, even if the optimal value for s is known, i.e., the value that minimizes the right-hand side in Eq. (4). However, as the condition always holds, an upper bound can be obtained by taking the minimum over any number of s values. In contrast to the convolution-based approach, the evaluation of the right hand side of Eq. (4) is linear to the number of jobs in the interval of interest.

4 Analytical Upper Bounds

Concentration inequalities have various applications in machine-learning, statistics, and discrete-mathematics. Here, we show how some of them can be used to derive analytical bounds on $\mathbb{P}(S_t \geq t)$ which are easier to compute than the Chernoff bounds. Specifically, we will apply the Hoeffding's inequality [12] and Bernstein's inequality [11].

The *Hoeffding's inequality* derives the targeted probability that the sum of independent random variables exceeds a given value. For completeness, we present the original theorem here:

▶ **Theorem 5** (Theorem 2 from [12]). *Suppose that we are given M independent random variables, i.e., $X_1, X_2, \ldots, X_M$. Let $S = \sum_{i=1}^{M} X_i$, $\bar{X} = S/M$ and $\mu = \mathbb{E}[\bar{X}] = \mathbb{E}[S/M]$. If*

$a_i \leq X_i \leq b_i$, $i = 1, 2, \ldots, M$, then for $s > 0$,

$$\mathbb{P}(\bar{X} - \mu \geq s) \leq \exp\left(-\frac{2M^2 s^2}{\sum_{i=1}^{M} (b_i - a_i)^2}\right) \tag{5}$$

Let $s' = sM$, i.e, $s = s'/M$. Hoeffding's inequality can also be stated with respect to S:

$$\mathbb{P}(S - \mathbb{E}[S] \geq s') \leq \exp\left(-\frac{2{s'}^2}{\sum_{i=1}^{M} (b_i - a_i)^2}\right) \tag{6}$$

By adopting Theorem 5, we can derive the probability that the sum of the execution times of the jobs in $hep(\tau_k)$ from time 0 to time t is no less than t:

▶ **Theorem 6.** *Let a_i be $C_{i,1}$ and b_i be $C_{i,h}$. Suppose that S_t is the sum of the execution times of the $\rho_{k,t} + \sum_{\tau_i \in hp(\tau_k)} \rho_{i,t}$ jobs in $hep(\tau_k)$ released from time 0 to time t. Then,*

$$\mathbb{P}(S_t \geq t) \leq \begin{cases} \exp\left(-\frac{2(t - \mathbb{E}[S_t])^2}{\sum_{\tau_i \in hep(\tau_k)} (b_i - a_i)^2 \rho_{i,t}}\right) & \text{if } t - \mathbb{E}[S_t] > 0 \\ 1 & \text{otherwise} \end{cases} \tag{7}$$

where $\rho_{i,t} = \left\lceil \frac{t}{T_i} \right\rceil$ and $\mathbb{E}[S_t] = \sum_{\tau_i \in hep(\tau_k)} (\sum_{j=1}^{h} C_{i,j} \mathbb{P}_i(j)) \cdot \rho_{i,t}$.

Proof. Since the execution time of a job of task τ_i is an independent random variable, there are in total $\rho_{i,t}$ independent random variables with the same distribution function upper bounded by $C_{i,h}$ and lower bounded by $C_{i,1}$ for each $\tau_i \in hep(\tau_k)$. With Eq. (6) and $s' = t - \mathbb{E}[S_t]$, we directly get:

$$\mathbb{P}(S_t \geq t) = \mathbb{P}(S_t - \mathbb{E}[S_t] \geq t - \mathbb{E}[S_t]) \leq \exp\left(-\frac{2(t - \mathbb{E}[S_t])^2}{\sum_{\tau_i \in hep(\tau_k)} (b_i - a_i)^2 \rho_{i,t}}\right) \tag{8}$$

when $s' > 0$. Otherwise, i.e., when $s' \leq 0$, we use the safe bound $\mathbb{P}(S_t \geq t) \leq 1$. ◀

The Chernoff bound and the related inequality by Hoeffding and Azuma can be generalized by the *Bernstein's inequality*. The original corollary is also stated here:

▶ **Theorem 7** (Corollary 7.31 from [11]). *Suppose that we are given L independent random variables, i.e., $X_1, X_2, \ldots, X_L$, each with zero mean, such that $|X_i| \leq K$ almost surely for $i = 1, 2, \ldots, L$ and some constant $K > 0$. Let $S = \sum_{i=1}^{L} X_i$. Furthermore, assume that $\mathbb{E}[X_i^2] \leq \theta_i^2$ for a constant $\theta_i > 0$. Then for $s > 0$,*

$$\mathbb{P}(S \geq s) \leq \exp\left(-\frac{s^2/2}{\sum_{i=1}^{L} \theta_i^2 + Ks/3}\right) \tag{9}$$

The proof can be found in [11]. Note, however, that the result in [11] is stated for the two-sided inequality, i.e., as upper bound on $\mathbb{P}(|S| \geq s)$. Here, the one-sided result, which is a direct consequence of the proof in [11] (page 198), is tighter.

Hence, we can derive the following upper bound:

▶ **Theorem 8.** *Suppose that the sum of the execution times of all $L = \rho_{k,t} + \sum_{\tau_i \in hp(\tau_k)} \rho_{i,t}$ jobs is S_t. Let $K = \max_{\tau_i \in hep(\tau_k)} C_{i,h} - \mathbb{E}[C_i]$ be the centralized WCET of any job, where $\mathbb{E}[C_i] = \sum_{j=1}^{h} \mathbb{P}_i(j) C_{i,j}$ is the expected execution time of a job of task τ_i. Then,*

$$\mathbb{P}(S_t \geq t) \leq \begin{cases} \exp\left(-\frac{(t - \mathbb{E}[S_t])^2/2}{\sum_{\tau_i \in hep(\tau_k)} \mathbb{V}[C_i]\rho_{i,t} + K(t - \mathbb{E}[S_t])/3}\right) & \text{if } t - \mathbb{E}[S_t] > 0 \\ 1 & \text{otherwise} \end{cases} \tag{10}$$

for any $t > 0$, where $\rho_{i,t} = \left\lceil \frac{t}{T_i} \right\rceil$ and $\mathbb{E}[S_t] = \sum_{\tau_i \in hep(\tau_k)} (\sum_{j=1}^{h} C_{i,j} \mathbb{P}_i(j)) \rho_{i,t}$.

Proof. Since for each task $\tau_i \in hep(\tau_k)$ the execution time of a job of task τ_i is an independent random variable, there are in total $\rho_{i,t}$ independent random variables with the same distribution function. Suppose that C_l is a random variable representing the execution time of a job of task τ_i and let $Y_l = C_l - \mathbb{E}[C_i] = C_l - \sum_{j=1}^{h} C_{i,j}\mathbb{P}_i(j)$ denote its centralized execution time. Since the expected execution time of a job is fully determined by its corresponding task, we have $\mathbb{E}[C_l] = \mathbb{E}[C_i]$.

Hereinafter, we explain why we adopt $\mathbb{V}[C_i]$ instead of θ_i^2 as known from Theorem 7. Consider Eq. (9) with $S = \sum_{l=1}^{M} Y_l$. The exact variance $\mathbb{V}[Y_l] = \mathbb{E}[Y_l^2] - \mathbb{E}[Y_l]^2 = \mathbb{E}[Y_l^2]$ is unknown and hence some loose upper bound θ^2 must be considered in most applications of Bernstein's inequality, like stated in Theorem 7. Here, the probabilities of the different execution modes are given numerically, i.e., $\mathbb{P}_i(j)$ for $C_{i,j}$. Hence, for an arbitrary but fixed task τ_i with h different execution modes, this results in

$$\mathbb{V}[Y_l] = \sum_{j=1}^{h} \mathbb{P}_i(j)\left(C_{i,j} - \mathbb{E}[C_i]\right)^2 = \sum_{j=1}^{h} \mathbb{P}_i(j)\left(C_{i,j}^2 - 2C_{i,j}\mathbb{E}[C_i] + \mathbb{E}[C_i]^2\right)$$

$$= \sum_{j=1}^{h} \mathbb{P}_i(j)C_{i,j}^2 - \sum_{j=1}^{h} \mathbb{P}_i(j)2C_{i,j}\mathbb{E}[C_i] + \sum_{j=1}^{h} \mathbb{P}_i(j)\mathbb{E}[C_i]^2 = \mathbb{E}[C_i^2] - \mathbb{E}[C_i]^2 = \mathbb{V}[C_i] \tag{11}$$

i.e., $\mathbb{V}[Y_l] = \mathbb{V}[C_i]$, which can be computed exactly in time $\mathcal{O}(h)$. Instead of imposing an upper bound θ^2, we can invoke the tightest version of Theorem 7 by using the exact variance.

Since $\mathbb{E}[Y_l] = 0$ and $\forall 1 \leq l \leq M : Y_l \leq K$, we can invoke Theorem 7 with $s = t - \mathbb{E}[S_t]$. When $s \leq 0$, we use a safe bound $\mathbb{P}(S_t \geq t) \leq 1$. When $s > 0$, Eq. (9) can be rewritten as

$$\mathbb{P}\left(\sum_{l=1}^{M} Y_l \geq t - \mathbb{E}[S_t]\right) \leq \exp\left(-\frac{(t - \mathbb{E}[S_t])^2/2}{\sum_{l=1}^{M} \mathbb{V}[Y_l] + K(t - \mathbb{E}[S_t])/3}\right) \tag{12}$$

Finally, observing that $\sum_{l=1}^{M} Y_l = S_t - \mathbb{E}[S_t]$ and $\sum_{l=1}^{M} \mathbb{V}[Y_l] = \sum_{\tau_i \in hep(\tau_k)} \mathbb{V}[C_i]\rho_{i,t}$ (from Eq. (11)) completes the proof. ◀

5 The Multinomial-Based Approach

In the traditional convolution-based approach [17], the underlying random variable represents the execution mode of each single job. First, we take a closer look on the related state space and show that the complexity of this approach depends on the specific definition of these random variables. Afterwards, we explain how this state space can be transformed into an equivalent space that describes the states on a task-based level by proving the invariance when considering equivalence classes for each task. As a result, we introduce our novel approach that is based on the multinomial distribution. The section is concluded with a short discussion regarding the complexity of our approach compared to the traditional convolution-based approach presented in Section 3.2.

5.1 The State Space of the Traditional Convolution-Based Approach

In this approach [17], $\boldsymbol{X}(t)$ is the set of the random variables representing the individual jobs released in the interval $[0, t)$ in the order of their arrival times. Note that the notion of $\boldsymbol{X}(t)$ instead of $\boldsymbol{X}$ is necessary, since the underlying state space and thus the underlying set of random variables are dependent on the considered t. Let $J(t)$ be the number of jobs released in $[0, t)$ under the critical instance of τ_k. Hence, $\boldsymbol{X}(t)$ represents a set of $J(t)$ independent random variables representing the execution modes of the individual tasks, i.e., $\boldsymbol{X}(t)$ is the Cartesian product over those $J(t)$ variables. To understand how the computation can be simplified, it is necessary to explicitly consider the random variables $\boldsymbol{X}(t)$ as well as the dependence between $\boldsymbol{X}(t)$ and the quantities S_t and C_i. To simplify notation, let us assume that all jobs have a common set of h execution modes $\mathcal{M}$, i.e., $|\mathcal{M}| = h$.[2] Thus, the state space of the random variable $\boldsymbol{X}(t)$ is $\mathcal{X}(t) = \mathcal{M}^{J(t)}$. A concrete assignment of these variables is denoted $\boldsymbol{x} \in \mathcal{X}(t)$, and the portion of $\boldsymbol{x}$ that corresponds to the jobs of task τ_i is denoted $\boldsymbol{x}_i$. Each task τ_i releases $\rho_{i,t} = \lceil t/T_i \rceil$ jobs, and thus $J(t) = \sum_{\tau_i \in hep(\tau_k)} \lceil t/T_i \rceil$. Hence, $\lceil t/T_i \rceil$ of the $J(t)$ random variables in $\boldsymbol{X}(t)$ are related to the task τ_i. Since the execution time of the j^{th} job of task τ_i depends on the related random variable $\boldsymbol{X}_{i,j}(t)$ we denote it $C_i(\boldsymbol{X}_{i,j}(t))$. Linking the total workload S_t to the random variables, from Eq. (2) we get:

$$S_t = S_t(\boldsymbol{X}(t)) = C_k(\boldsymbol{X}_{k,1}(t)) + \sum_{\tau_i \in hp(\tau_k)} \sum_{j=1}^{\rho_{i,t}} C_i(\boldsymbol{X}_{i,j}(t)) \tag{13}$$

Based on this, we denote the exact expression for the probability of a overload at time t as

$$\mathbb{P}(S_t(\boldsymbol{X}(t)) > t) = \sum_{\boldsymbol{x} \in \mathcal{X}(t)} \mathbb{P}(\boldsymbol{X}(t) = \boldsymbol{x}) \mathbb{1}_{\{S_t(\boldsymbol{x}) > t\}} \tag{14}$$

Here, $\mathbb{1}_{\{\text{expression}\}}$ is the *indicator function* which evaluates to 1 if and only if the expression is true, and to 0 otherwise. Since the execution modes of the jobs are assumed to be independent, the joint probability mass $\mathbb{P}(\boldsymbol{X}(t))$ factorizes over the jobs. The probability of each execution mode per job is fully determined by its corresponding task, and hence

$$\mathbb{P}(\boldsymbol{X}(t) = \boldsymbol{x}) = \prod_{\tau_i \in hp(\tau_k)} \prod_{j=1}^{\rho_{i,t}} \mathbb{P}_i(\boldsymbol{x}_{i,j}(t)) \tag{15}$$

Each factor $\mathbb{P}_i(x)$ is the probability mass of any job of task τ_i, being in some state $x \in \mathcal{M}$. Note that Eq. (14) is exactly the quantity computed by the traditional convolution-based approach [17]. Hence, its stems from the state space $\mathcal{X}(t) = \mathcal{M}^{J(t)}$ that is exponential in the total number of jobs. Nevertheless, we leverage the independence of job modes to compute $\mathbb{P}(S_t(\boldsymbol{X}(t))) \geq t)$ over a different state space, which is the key insight of our method.

5.2 Invariance and Equivalence Classes

In Eq. (15), for any fixed task τ_i, the expression $\prod_{j=1}^{\rho_{i,t}} \mathbb{P}_i(\boldsymbol{x}_{i,j})$ is determined by the number of jobs for each state in $\mathcal{M}$. As an example, consider an arbitrary task τ_i with two distinct execution states, i.e., $\mathcal{M} = \{C_{i,1}, C_{i,2}\}$, and suppose that $\boldsymbol{x}_i = (C_{i,1}, C_{i,2}, C_{i,1}, C_{i,2})$, $\boldsymbol{x}'_i = (C_{i,1}, C_{i,1}, C_{i,2}, C_{i,2})$, and $\boldsymbol{x}''_i = (C_{i,2}, C_{i,1}, C_{i,1}, C_{i,2})$. The resulting probability is identical in all three cases, i.e., $\mathbb{P}_i(\boldsymbol{x}_i) = \mathbb{P}_i(\boldsymbol{x}'_i) = \mathbb{P}_i(\boldsymbol{x}''_i)$. We formalize this property subsequently.

▶ **Lemma 9** (Probability Permutation Invariance). *Let τ_i be a task with a set of distinct execution modes $\mathcal{M}$, let $\rho_{i,t}$ be the number of jobs of τ_i released up to time t, and let $\boldsymbol{x}_i \in \mathcal{M}^{\rho_{i,t}}$ be the random vector that represents the execution mode of all jobs which belong to task τ_i. The probability mass $\mathbb{P}_i$ is permutation invariant with respect to $\boldsymbol{x}_i$, i.e.,*

$$\forall\ \boldsymbol{x}_i \in \mathcal{M}^{\rho_{i,t}} : \forall \sigma \in \mathbb{S}_{\rho_{i,t}} : \mathbb{P}_i(\boldsymbol{x}_i) = \mathbb{P}_i(\sigma(\boldsymbol{x}_i)) \tag{16}$$

where $\mathbb{S}_n$ contains all permutations of n objects.

[2] If a task has less than h (or even only one) execution modes, dummy modes with probability 0 can ensure this condition. Alternatively, $\mathcal{M}_i$ and h_i can be defined based on the execution modes of τ_i.

Proof. The lemma follows directly from the independence of job-wise execution modes, thus $\mathbb{P}_i(\boldsymbol{x}_i) = \prod_{j=1}^{\rho_{i,t}} \mathbb{P}_i(\boldsymbol{x}_{i,j})$, and from the commutativity of the multiplication. ◀

Up to now, we considered just a single task τ_i, but the lemma indeed holds *for all* tasks simultaneously. Recall that the random modes of all tasks are represented by $\boldsymbol{X}(t)$. Let $\boldsymbol{X}_i(t)$ represent the random modes of the jobs of task τ_i, i.e., $\boldsymbol{X}_i(t)$ is the subset of random variables in $\boldsymbol{X}(t)$ that relate to the random modes of τ_i. Applying the permutation invariance to each $\boldsymbol{X}_i(t)$, we derive a partition on $\mathcal{X}(t)$ into equivalence classes.

▶ **Definition 10** (Execution Mode Equivalence Classes). For any $\boldsymbol{x} \in \mathcal{X}(t)$, its equivalence class $[\![\boldsymbol{x}]\!]$ with respect to permutation invariance is given by

$$[\![\boldsymbol{x}]\!] = \{\boldsymbol{x}' \in \mathcal{X}(t) \mid \forall \tau_i \in hep(\tau_k) : \exists \sigma \in \mathbb{S}_{\rho_{i,t}} : \boldsymbol{x}_i = \sigma(\boldsymbol{x}'_i)\} \tag{17}$$

Based on this definition, the statement $\forall \boldsymbol{x}' \in [\![\boldsymbol{x}]\!] : \mathbb{P}(\boldsymbol{x}) = \mathbb{P}(\boldsymbol{x}')$ is a straightforward corollary of Lemma 9. The equivalence relation in Lemma 10 is established by an equivalent occurrence of execution modes for each task. Hence, each equivalence class has a canonical representative, given by a tuple $\boldsymbol{\ell} \in \otimes_{\tau_i \in hep(\tau_k)}\{1, 2, \ldots, \rho_{i,t}\}^{|\mathcal{M}|}$, which for each task contains the number of jobs for all execution modes. For convenience we use $[\![\boldsymbol{\ell}]\!]$ to address the set of all $\boldsymbol{x}$ in the same equivalence class and rephrase Eq. (14) accordingly.

▶ **Lemma 11** (Class-based Overload Probability). *For any set of execution modes $\mathcal{M}$, let $\mathcal{L}(t) = \otimes_{\tau_i \in hep(\tau_k)}\{0, 1, 2, \ldots, \rho_{i,t}\}^{|\mathcal{M}|}$. Then,*

$$\mathbb{P}(S_t(\boldsymbol{X}(t)) \geq t) = \sum_{\boldsymbol{\ell} \in \mathcal{L}(t)} \prod_{\tau_i \in hep(\tau_k)} \frac{\rho_{i,t}! \prod_{j=1}^{|\mathcal{M}|} \mathbb{P}_i(j)^{\boldsymbol{\ell}_{i,j}}}{\prod_{x \in \mathcal{M}} \boldsymbol{\ell}_{i,x}!} \mathbb{1}_{\{S_t([\![\boldsymbol{\ell}]\!]) \geq t\}} \tag{18}$$

where $\boldsymbol{\ell}_{i,j}$ denotes the number of jobs of task τ_i which are in the j-th execution mode, and $S_t([\![\boldsymbol{\ell}]\!])$ denotes the execution time for some arbitrary $\boldsymbol{x} \in [\![\boldsymbol{\ell}]\!]$.

Proof. For all members of the class $[\![\boldsymbol{x}]\!]$, each task has the same number of jobs which are in the same state. Iterating over the set $\mathcal{L}(t) = \bigotimes_{\tau_i \in hep(\tau_k)}\{0, 1, 2, \ldots, \rho_{i,t}\}^{|\mathcal{M}|}$ corresponds to iterating over all such count vectors, which is in turn the same as iterating over all equivalence classes $[\![\boldsymbol{x}]\!]$. Each class $[\![\boldsymbol{\ell}]\!]$ contains all state permutations for all jobs of each task. For each task τ_i, this is equivalent to the well-known combinatorial problem of counting the number of ways how $\rho_{i,t}$ objects can be placed into $|\mathcal{M}|$ bins, given by the corresponding multinomial coefficient. Combining those for all tasks, we get

$$|[\![\boldsymbol{\ell}]\!]| = \prod_{\tau_i \in hep(\tau_k)} \binom{\rho_{i,t}}{\boldsymbol{\ell}_{i,1}\ \boldsymbol{\ell}_{i,2}\ \ldots\ \boldsymbol{\ell}_{i,|\mathcal{M}|}} = \prod_{\tau_i \in hep(\tau_k)} \frac{\rho_{i,t}!}{\prod_{x \in \mathcal{M}} \boldsymbol{\ell}_{i,x}!} \tag{19}$$

Combining these facts, we get

$$\sum_{\boldsymbol{x} \in \mathcal{X}(t)} \mathbb{P}(\boldsymbol{X}(t) = \boldsymbol{x}) = \sum_{\boldsymbol{\ell} \in \mathcal{L}(t)} |[\![\boldsymbol{\ell}]\!]| \mathbb{P}(\boldsymbol{X}(t) = [\![\boldsymbol{\ell}]\!]) \tag{20}$$

Observing that $\mathbb{P}(\boldsymbol{X}(t) = [\![\boldsymbol{\ell}]\!]) = \prod_{j=1}^{|\mathcal{M}|} \mathbb{P}_i(j)^{\boldsymbol{\ell}_{i,j}}$ implies the lemma. ◀

5.3 Detailing the Multinomial Approach

Now, we can combine the findings of Section 5.1 and Section 5.2 into an algorithm for calculating $\mathbb{P}(S_t > t)$, i.e., the probability of an overload for a length t, more efficiently.

For simplicity of presentation, we will also refer to the overload probability *at time* t and the state space *at time* t, implicitly assuming that both the probability and the state space is calculated considering the interval $[0, t)$ with respect to the critical instant of τ_k. The traditional convolution-based approach determines this probability by successively calculating the probability for all other points of interest in the interval $[0, t)$. Nevertheless, the probability for t is evaluated based on the resulting states after all jobs in $[0, t)$ are convoluted. With respect to t, the intermediate states are not considered.

We utilize this insight to calculate the vector representing the possible states at time t more efficiently. Lemma 9 shows that the overload probability of a state for a concrete variable assignment $\boldsymbol{x} \in \mathcal{X}(t)$ is identical to the probability of all permutations of $\boldsymbol{x}$, i.e., the related equivalence class. This allows us to consider the jobs in $J(t)$ in any order. We further know from Lemma 11 that all assignments that are part of the same equivalence class result in the same value for S_t. Considering only one task τ_i, those assignments differ regarding the order in which the execution modes happen but not with respect to the total number of executions in a given mode. However, if the jobs are convoluted in the non-decreasing order of their arrival times, this leads to a large number of unnecessary states that will be merged in the end. For example, in Figure 1 the state space can be reduced if the second job of τ_1 would be convoluted before the job of τ_2 is convoluted, since the resulting merged state space after the convolution of the two jobs of τ_1 only has 3 states that represent the number of executions in each mode. Therefore, to reduce the state space as much as possible, we consider the jobs ordered according to the tasks they are related to, i.e., first all $\rho_{1,t}$ jobs of τ_1 are considered, then all $\rho_{2,t}$ jobs of τ_2, etc. However, if the jobs are just reordered and then convoluted, this still leads to a large number states that are merged later on.

Regardless, the number of states is already significantly lower than in the traditional convolution-based approach. Fortunately, if the number of jobs for a task is known, all possible combinations and the related probabilities can be calculated directly using the multinomial distribution. To be more precise, assume a given task τ_i as well as a given number of releases $\rho_{i,t}$ in an interval of length t and let $\boldsymbol{\ell}_{i,j}$ be the number executions in mode $j \in \{1, ..., h\}$. We know that $\boldsymbol{\ell}_{i,j} \in \{0, 1, ..., \rho_{i,t}\}$ and $\sum_{j=1}^{h} \boldsymbol{\ell}_{i,j} = \rho_{i,t}$, leading to $\binom{\rho_{i,t}+h-1}{h-1}$ possible combinations of $\boldsymbol{\ell}_{i,1}, ..., \boldsymbol{\ell}_{i,h}$ where $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ is the binomial coefficient. For each combination, we can calculate the related probability as

$$\frac{\rho_{i,t}!}{\boldsymbol{\ell}_{i,1}!\boldsymbol{\ell}_{i,2}!...\boldsymbol{\ell}_{i,h}!}\mathbb{P}_i(1)^{\boldsymbol{\ell}_{i,1}} \cdot \mathbb{P}_i(2)^{\boldsymbol{\ell}_{i,2}} \cdot ... \cdot \mathbb{P}_i(h)^{\boldsymbol{\ell}_{i,h}} \tag{21}$$

where $\frac{\rho_{i,t}!}{\boldsymbol{\ell}_{i,1}!\boldsymbol{\ell}_{i,2}!...\boldsymbol{\ell}_{i,h}!}$ determines the number of possible paths for the related equivalence classes and $\mathbb{P}_i(1)^{\boldsymbol{\ell}_{i,1}} \cdot \mathbb{P}_i(2)^{\boldsymbol{\ell}_{i,2}} \cdot ... \cdot \mathbb{P}_i(h)^{\boldsymbol{\ell}_{i,h}}$ is the probability of one of these paths. The total workload of the $\rho_{i,t}$ jobs of τ_i is calculated for each of these combinations based on the related values of $\boldsymbol{\ell}_{i,1}$ to $\boldsymbol{\ell}_{i,h}$. The $\binom{\rho_{i,t}+h-1}{h-1}$ states represent the equivalence classes of τ_i and the related probabilities. After calculating these representatives for each task, the overload probability can be calculated by convoluting them and adding up the overload probabilities of the resulting state space. A concrete example for our approach, assuming that each task has two possible execution modes, is given in Figure 2. Details on how some equations can be simplified in this case can be found in the related full version [27]. Note that based on Lemma 9 the states representing the tasks can be convoluted in any order.

In fact, considering t, the job-based state space of the traditional convolution-based approach has been transferred into a task-based space state with identical properties regarding the overload probability. To visualize the different approaches, the traditional convolution-based approach constructs a binary tree based on the jobs (see Figure 1) where each layer

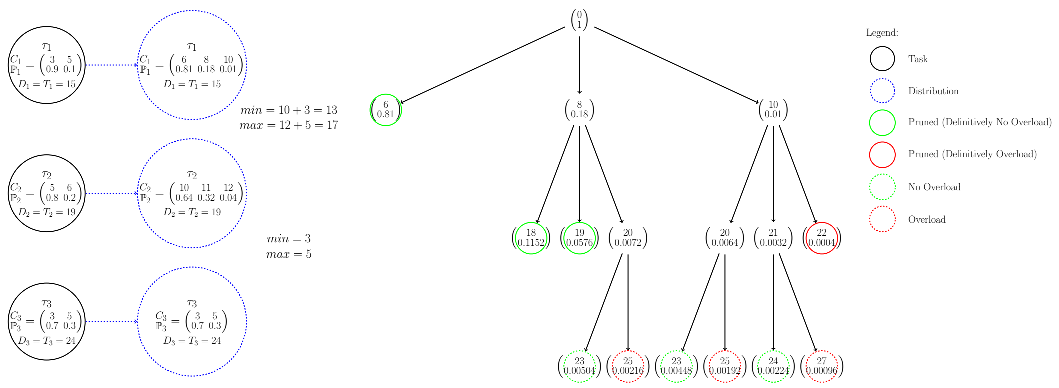


Figure 2 The multinomial approach convoluting 3 tasks with two modes. The number of children depends on the number of jobs of the related task. Note that nodes can be ignored in further steps if they never lead to an overload (green solid circles) or if they always lead to an overload (red solid circle). In the end, the overload probability at $t = 24$ is calculated by summing up the related probabilities (dashed and solid red) which leads to deadline miss probability of 0.00574.

represents the state of the system after the related job is convoluted. The multinomial-based approach on the other hand constructs a tree based on the tasks (see Figure 2) which means that the number of children on each level depends on the number of jobs the related task releases. If the nodes on the $J(t)^{th}$ level of the binary tree are merged as show in Figure 1, the number of states on that level is identical to the number of states on the k^{th} level of the tree resulting from our approach. While the state space of our reformulation is still large, it opens up opportunities for pruning strategies and other state reduction strategies which are not suitable for the traditional approach. These strategies will be explained in Section 6.

5.4 Complexity Discussion and Comparison

When considering the complexity of the multinomial-based approach for τ_k over an interval $[0, t)$ (an interval of length t that ends at time t for notational brevity) under the critical instance of τ_k, both the number of tasks that are contributing to the workload in the interval, i.e., $\rho_{i,t}$ for the higher priority tasks, and the total number of jobs in the interval $J(t)$ have to be considered. The number of multinomial coefficients depends on $\rho_{i,t}$ and the number of possible execution states h for each task and can be calculated as $\binom{\rho_{i,t}+h-1}{h-1}$. This is also called the h-simplex of the $\rho_{i,t}^{th}$ component. The convolution of these states over all tasks leads to a total number of states of $\prod_{i=1}^{k}\binom{\rho_{i,t}+h-1}{h-1}$.

The classical convolution-based approach considers each job individually with h possible outcomes and, therefore, leads to $h^{J(t)}$ states, i.e., it is exponential in the number of jobs. Hence, without state merging, it is not feasible for input sets with a sensible cardinality. However, the convolution-based approach in the process also calculates the deadline miss probability at all possible points of interest in the interval, i.e., at each point in time a job is released. Furthermore, states can be merged when they have the same related workload, e.g., states resulting from a permutation of the same number of abnormal executions of a given task. Lemma 9 directly implies that when convolution is used in combination with merging states, the final number of states for the convolution-based approach at time t is identical to the number of states created by the multinomial-distribution-based approach (assuming that all states created by our approach lead to pairwise different workloads). However, while our approach creates only necessary states, the traditional convolution-based approach not only creates unnecessary states but also requires additional overhead for state merging after each step. Therefore, when considering a single point in time our approach is significantly faster than the traditional convolution-based approach with task merging. On the other hand, since our approach needs to consider all points of interest individually, if the number of such points increases due to the number of tasks the traditional convolution-based approach should be favoured. However, we were not able to observe this behaviour in our evaluation since both our multinomial-based approach as well as the traditional convolution-based approach with state merging only rarely were able to provide results for task sets with a cardinality of 10. Hence, for our approach runtime optimizations are provided in the next section. Note that this differs depending on the actual setting and that the period range is the most important parameter since it relates to the number of jobs.

6 Runtime Improvement

Here we introduce two strategies to improve the runtime efficiency. The first one prunes the state space, i.e., discards states directly if the impact on the overload probability can be determined without considering the remaining tasks, detailed in Section 6.1. This reduces the runtime without sacrificing any precision. The second technique combines execution mode equivalence classes with very low probability when creating the task representations to reduce the size of the state space beforehand, explained in Section 6.2. While this leads to an increase of the resulting overload probabilities, this error can be bounded for each task under consideration and therefore also with respect to the total error of the derived overload probability. Note that both techniques can be combined, which is done in the evaluation.

6.1 Pruning the State Space

Our multinomial-based approach calculates the probabilities for each interval individually, a property we already used when we transferred the state space from a job-based to a task-based state space. For convenience, assume that in our multinomial-based approach the representatives of the tasks are convoluted according to the task index. Recall that the state space can be seen as a rooted tree where each node on the j^{th} row represents a possible state after the convolution of the first j tasks and that we are only interested in the nodes on the k^{th} (and last) layer, i.e., the states after all task representations are convoluted. Such a tree is displayed in the example in Figure 2. The general concept of pruning is to remove a state R if the resulting subtree, i.e., the subtree with root R, has no further impact on the evaluation on the k^{th} layer, i.e., either *all* states on the k^{th} layer in the subtree with root R evaluate to an overload or for *all* states on the k^{th} layer in the subtree with root R the resulting workload is less than the interval length. In the first case, the state is discarded and the related probability is added to the overload probability considering t. In the second case, the state is directly discarded. This is done by checking the boundary conditions. To this end, for each task we determine the minimum and maximum execution time it can contribute to the total workload up to time t, respectively, which can be easily done while calculating the vectors that represent the task. On the i^{th} layer, the minimum and maximum workload that can be contributed by the remaining tasks, denoted as $C_{\min_i}$ and $C_{\max_i}$, is the sum of the minimum and maximum values related to the remaining tasks. Let $\mathbb{P}(\text{discard})$ be a variable accounting for the overload probability of discarded states, initialized with 0. For each state Q created by the convolution of τ_i with the previous state space let $C(Q)$ be the related total workload. We check the two following conditions:

1. $C(Q) + C_{\max_i} \leq t$: In this case the subtree rooted at Q only leads to states that will not lead to an overload at time t, since the branch related to the maximum cumulative workload in this subtree does not lead to an overload. Therefore, Q can directly be discarded. In the example in Figure 2 those states are marked with a solid green circle.
2. $C(Q) + C_{\min_i} > t$: All paths in the subtree rooted at Q result in an overload at time t, since the branch related to the minimum cumulative workload in this subtree leads to an overload. Hence, Q can directly be discarded and $\mathbb{P}(\text{discard})$ is increased by the probability of Q. In Figure 2 those states are marked with a solid red circle.

Obviously all created states can only fulfill one of these two conditions but not both due to $C(Q) + C_{\min_i} \leq C(Q) + C_{\max_i}$. If Q fulfills none, the state is added to the representation of $\tau_1, ..., \tau_i$. The correctness of this pruning approach follows directly from the observations that the total probability of a subtree on each level is equal to the probability of the root and from the fact that the total workload of each branch is always smaller than the maximum workload (larger than the minimum workload, respectively). A proof is therefore omitted. Note that the order in which the tasks are considered has no impact on the applicability of the pruning technique.

When considering a similar technique for the traditional convolution-based approach, one major difference is that the overload probability of all values is calculated successively. To be more precise, it considers the critical instant of τ_k at time 0 and the deadline miss probability for all intervals $[0, t)$, where t is the release time of a higher priority task. The interval $[0, D_k)$ is calculated successively and the result at time t_b depends on the result at time t_a if $t_a < t_b$. We visualize this by a rooted directed binary tree where each layer represents an arriving job and the layers are created according to the jobs arrival time, i.e., the height of the tree depends on the number of considered jobs (see Figure 1). The nodes on each layer represent the state space after the convolution of the related job. One important property of this approach is that the probability of deadline miss is calculated on each layer. Hence, pruning a state, i.e., removing a state and the branches resulting from it, can only be done if those branches have no impact on the probability on *all* following layers, i.e, a state R at time t_a can only be pruned if all branches of the subtree with root R will for all $t_b \in (t_a, D_k]$ either lead to an overload at t_b or to no overload at t_b. This cannot be determined by evaluating the overload condition for any single time point $t_b \in (t_a, D_k]$. Assume, for instance, for a $t_b \in (t_a, D_k]$ that $C(Q) + C_{\min_{t_b}} > t_b$ where $C_{\min_{t_b}}$ is the minimum workload created by jobs released in the interval $[t_a, t_b)$. Let t_{b-1} and t_{b+1} be the previous and next considered points with respect to t_b in the convolution based approach. We observe that τ_k may have no overload at t_{b-1}, if the minimum workload of the job released at t_{b-1} is smaller than $t_b - t_{b-1}$. Similar arguments can be taken to create a case with no overload at t_{b+1} and for the cases where τ_k has no overload at t_b if $C_{\max_{t_b}}$ is considered.

6.2 Union of Execution Mode Equivalence Classes

The general concept of the presented runtime improvement technique is to reduce the state space by unifying equivalence classes with low probability when creating the representation for the individual tasks. In contrast to the pruning technique, this obviously results in a loss of precision when approximating the deadline miss probability for a given point in time. However, if done carefully, the precision loss can be upper bounded by a constant. We will introduce the concept based on the example in Table 2. Therein, we detail the release of 10 jobs in the interval of interest for a task τ_i with two execution modes that have a WCET of $C_{i,1} = 1$ and $C_{i,2} = 2$, with related probabilities $\mathbb{P}_i(1) = 0.975$ and $\mathbb{P}_i(2) = 0.025$. In the upper half, the original equivalence classes are displayed, i.e., one for each possible number

Table 2 Distribution for 10 releases of τ_i with $C_{i,1} = 1$, $C_{i,2} = 2$, $\mathbb{P}_i(1) = 0.975$, $\mathbb{P}_i(2) = 0.025$. The upper part details the distribution before and the lower part after merging equivalence classes.

# $C_{i,2}$ jobs	0	1	2	3	4	5	6	7	8	9	10
Total C_i	10	11	12	13	14	15	16	17	18	19	20
Probability	0.78	0.2	0.023	0.0016	$7.0 \cdot 10^{-05}$	$2.2 \cdot 10^{-06}$	$4.63 \cdot 10^{-08}$	$6.8 \cdot 10^{-10}$	$6.53 \cdot 10^{-12}$	$3.72 \cdot 10^{-14}$	$9.5 \cdot 10^{-17}$
# $C_{i,2}$ jobs	0	1	2	3	4	5	6 or 7		8, 9, or 10		
Total C_i	10	11	12	13	14	15	17		20		
Probability	0.78	0.2	0.023	0.0016	$7.0 \cdot 10^{-05}$	$2.2 \cdot 10^{-06}$	$4.701 \cdot 10^{-08}$		$6.564711 \cdot 10^{-12}$		

of jobs (0 to 10), together with their total WCET and their (rounded) related probability. We will explain afterwards how the approach can be generalized.

The probability decreases rapidly with respect to the number of executions in the mode related to $C_{i,2}$. Such distributions are common when considering probabilistic execution times for real-time systems. The reason is that if the execution mode with larger WCET has a comparatively high probability, classical non-probabilistic worst-case response time analysis considering the larger WCET should be used to ensure timeliness for relatively common cases. Since the probability of the equivalence classes decreases, the impact of those classes on the overload probability over the given interval decreases as well. Therefore, the number of states that are created in our approach, and thus the runtime, can be reduced by unifying some of these highly unlikely equivalence classes. To guarantee a safe approximation, i.e., the resulting overload probability is only increased, we define the merge of a set of equivalence class as follows:

▶ **Definition 12** (Union of Task Equivalence Classes). *Let $\mathcal{C} = \{[\![\boldsymbol{x}_i]\!], [\![\boldsymbol{x}'_i]\!], [\![\boldsymbol{x}''_i]\!], \ldots\}$ be a set of $|\mathcal{C}| = q$ equivalence classes of task τ_i in a given interval of interest $[0, t)$. For each class $[\![\boldsymbol{x}_i]\!] \in \mathcal{C}$, let $\mathbb{P}_i([\![\boldsymbol{x}_i]\!])$ and $C_i([\![\boldsymbol{x}_i]\!])$ denote its probability and the related total worst-case execution time, respectively. Furthermore, let $[\![\boldsymbol{x}_i^{\max}]\!] \in \mathcal{C}$ be the equivalence class with the highest total WCET, i.e., $[\![\boldsymbol{x}_i^{\max}]\!] = \arg\max_{[\![\boldsymbol{x}_i]\!] \in \mathcal{C}} C_i([\![\boldsymbol{x}_i]\!])$.*

When we union all classes in $\mathcal{C} = \{[\![\boldsymbol{x}_1]\!], \ldots, [\![\boldsymbol{x}_q]\!]\}$, the classes in $\mathcal{C}$ are replaced by a a new class $[\![\boldsymbol{x}_i^{\mathcal{C}}]\!] = \bigcup_{[\![\boldsymbol{x}_i]\!] \in \mathcal{C}} [\![\boldsymbol{x}_i]\!]$ that has the following characteristics:

1. $C_i([\![\boldsymbol{x}_i^{\mathcal{C}}]\!]) = C_i([\![\boldsymbol{x}_i^{\max}]\!])$
2. $\mathbb{P}_i([\![\boldsymbol{x}_i^{\mathcal{C}}]\!]) = \sum_{[\![\boldsymbol{x}_i]\!] \in \mathcal{C}} \mathbb{P}_i([\![\boldsymbol{x}_i]\!])$

As shown in Table 2, merging the equivalence classes for 6 and 7 executions of mode 2, the probability of the newly created class is the summation of their probabilities and the related WCET is the maximum among those two classes, i.e., the WCET of the class with 7 executions. We now show that merging a set of equivalence classes leads to a bounded error with respect to the overload probability.

▶ **Lemma 13** (Unifying Equivalence Classes Leads to a Bounded Maximum Error). *For task τ_i let $\mathcal{C} = \{[\![\boldsymbol{x}'_i]\!], [\![\boldsymbol{x}''_i]\!], \ldots\}$ be a set of $|\mathcal{C}| = q$ equivalence classes for the interval of interest $[0, t)$. If $\mathcal{C}$ is merged into $[\![\boldsymbol{x}_i^{\mathcal{C}}]\!]$ according to Definition 12, the probability of overload can only increase and the error is bounded by $(\sum_{[\![\boldsymbol{x}_i]\!] \in \mathcal{C}} |[\![\boldsymbol{x}_i]\!]| \mathbb{P}_i([\![\boldsymbol{x}_i]\!])) - |[\![\boldsymbol{x}_i^{\max}]\!]| \mathbb{P}_i([\![\boldsymbol{x}_i^{\max}]\!])$.*

This follows from Eq. (18), Eq. (20), and the fact that any $\mathcal{C}$ in which no class $[\![\boldsymbol{x}_i]\!]$ triggers the indicator function $\mathbb{1}_{\{S_t([\![\boldsymbol{x}]\!]) > t\}}$ does not introduce any error. Hence, if at least $[\![\boldsymbol{x}_i^{\max}]\!]$ triggers $\mathbb{1}_{\{S_t([\![\boldsymbol{x}]\!]) > t\}}$ the maximum probability increase happens if all other classes did not trigger $\mathbb{1}_{\{S_t([\![\boldsymbol{x}]\!]) > t\}}$ before the unification but do afterwards. Since the process can be repeated for all tasks this directly leads to:

▶ **Theorem 14** (Bounded For The Overall Increase On The Overload Probability). *If equivalence classes of tasks with respect to the interval* $[0, t)$ *are merged, the total increase of the overload probability for this interval is increased by the sum of the individual overload probability increase of the individually tasks.*

Now we can calculate the overloaded probability over $[0, t)$ with a bounded total error while reducing the states that have to be considered. Assume a value b for the allowed maximum error to be given and a set of n tasks. The maximum error is bounded by b if for each task the error is bounded by b/n. This can be achieved by ordering the related states in decreasing order of probability, traversing them in this order while summing up the probabilities of each state, and keeping all states until the summation is larger than $1 - b/n$. Afterwards the remaining states are unified into one.

So far we considered a setting similar to the one displayed in Table 2, i.e., the workload increases as the probability decreases. However, this is not necessarily the case, e.g., when a task has two execution modes with an equal probability or when a task has three execution modes and $C_{i,2}$ has the lowest probability. Nevertheless, in such cases the approach based on Theorem 14 can still directly be exploited since the union of equivalence classes is agnostic to the workloads and related probabilities as long as the total probability of the combined equivalence classes is less than b/n and thus the approach can directly be used. Hence, for a given task properties of the related distribution can be exploited in the process. For example, for two execution modes with identical probability the symmetry of the resulting distribution can be used if modes with a total probability of $b/2n$ at both ends are unified.

7 Evaluation

The main focus of our evaluation was to determine if our novel multinomial-based approach can provide good results in reasonable analysis runtime, especially considering the scalability with respect to the number of tasks for reasonable settings. To this end, for a given utilization U_{sum} and a number of tasks, we generated random implicit-deadline task sets with one execution mode according to the UUniFast method [6]. As suggested by Emberson et al. [10], the periods of those tasks were generated according to a log-uniform distribution with two orders of magnitude, i.e., $10ms - 1000ms$. We only considered tasks with two distinct execution modes in the evaluation, called normal and abnormal execution mode and hence $\mathcal{M} = \{N, A\}$. The normal execution mode is considered to have a (much) higher probability. The WCET in the normal mode was set according to the utilization, i.e., $C_{i,N} = U_i \cdot T_i$ and the WCET in abnormal mode was calculated as $C_{i,A} = f \cdot C_{i,N}$ for all tasks in the set.

We used a fixed setting, defined by U_{sum}, f, and $\mathbb{P}_i(A)$, tracking the resulting deadline miss probability and runtime related parameters. In each setting, the deadline miss probability for the lowest-priority task under the rate-monotonic scheduling approach was determined. In our evaluations, we considered the following approaches where the **bold** name indicates how the approach is referred to:

1. **Convolution:** The *traditional convolution-based approach* [17].
2. **Conv. Merge:** The *traditional convolution-based approach* [17] with state merging.
3. **Multinomial:** Our novel multinomial-based approach from Sec. 5.3.
4. **Pruning:** The approach in Sec. 5.3 combined with the pruning technique in Sec. 6.1.
5. **Unify:** The approach in Sec. 5.3 combined with the pruning technique in Sec. 6.1 and reducing the complexity with the union of equivalence classes presented in Sec. 6.2.
6. **Approx:** Approximation of **Pruning** by only considering the deadline of τ_k and the last releases of higher-priority tasks, inspired from the literature, e.g., [7, 5, 25, 8].

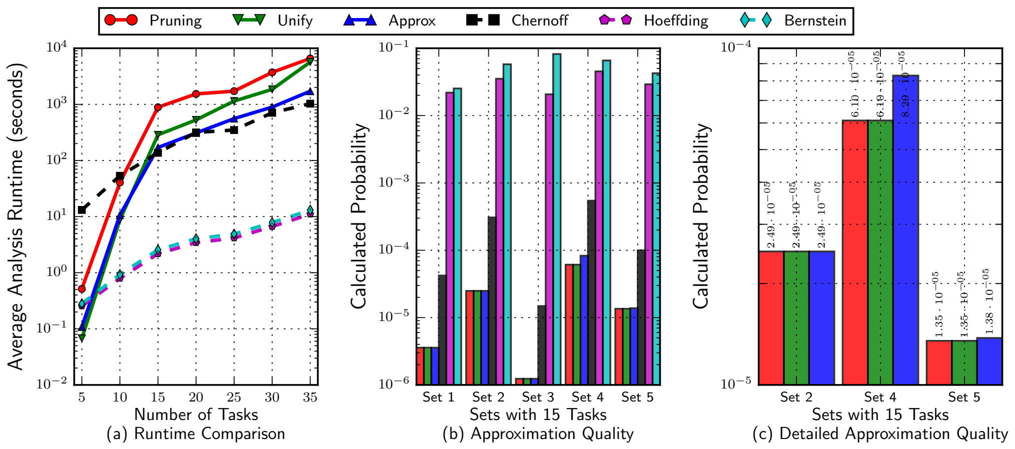


Figure 3 (a) Average runtime with respect to task set cardinality. (b) Approximation quality for 5 sets with 15 tasks. (c) Detailed approximation quality for the multinomial-based approaches.

7. **Chernoff:** The analytical approach using *Chernoff bounds* by Chen and Chen [8].
8. **Hoeffding:** The analytical approach using *Hoeffding's inequality* (Sec. 4).
9. **Bernstein:** The analytical approach using *Bernstein inequalities* (Sec. 4).

To allow runtime comparisons, all approaches were implemented in the same programming language, i.e., Python, and executed on the same machine, i.e., a 12 core Intel Xeon $X5650$ with 2.67 GHz and 20 GB RAM. For the analytical bounds, in contrast to the work by Chen and Chen [8], all releases of higher-priority tasks were considered since the bounds have a lower runtime than our novel approach.

Figure 3 shows the results for randomly generated tasks sets with a normal-mode utilization of $U_{sum} = 70$, and for all tasks $f = 2$ and $\mathbb{P}_i(A) = 0.025$ were assumed. Hence, $\mathbb{P}_i(N) = 0.975$. To analyze the scalability, the cardinality of the task sets ranged from 5 to 35 in steps of 5. In Figure 3(a) the average runtime of the analysis is displayed with respect to the cardinality. For a cardinality from 5 to 20 tasks, we evaluated 20 task sets while a cardinality from 25 to 35 tasks, due to the high runtime, 5 task sets were analyzed. For **Convolution** usually no result was delivered for a cardinality of 5, i.e., a crash due to an out of memory error occurred. Even for 3 tasks no result could be provided in some cases since, for instance, 38 jobs already leads to $2^{38} = 274877906944$ states for D_k in **Convolution**. For **Conv. Merge** and **Multinomial** a setting with 10 tasks often lead to no results. Hence, those three approaches are not displayed. However, the results for **Conv. Merge**, **Multinomial**, and **Pruning** were always identical (if **Conv Merg** and **Multinomial** derived results), showing that our pruning technique drastically decreases the runtime of the analysis and increases the scalability without any precision loss. We see that **Bernstein** and **Hoeffding** are orders of magnitude faster than the other approaches which are compatible with respect to the related runtime. The large runtime of **Chernoff** yields from finding a *good* s value in Eq. (4) which may differ for each point in time. The difference between **Approx** and **Pruning** stems from a different number of tested time points, i.e., for **Approx** this number depends on the number of tasks while for **Pruning** it is related to the number of jobs, while the calculation for one time point does not differ largely.

The statistical information of the derived deadline miss probabilities is unfortunately not meaningful. For example, for task sets with 15 tasks, the derived deadline miss probability in our evaluations under **Pruning** ranged from $3.0 \cdot 10^{-39}$ to $6.1 \cdot 10^{-5}$. Therefore, comparing the average values or other statistical means does not yield much information. In addition, comparing relative values is problematic if the probability gets low. Hence, we show a small sample of 5 task sets with roughly similar probabilities in Figure 3(b). These are the first 5 randomly generated task sets with deadline miss probability larger than 10^{-6}. This selection is only done to increase the readability of the figure. We observed in general similar relative behaviour among (nearly) all the evaluated task sets. We see that the error of **Bernstein** and **Hoeffding** is large compared to **Chernoff**, i.e., by several orders of magnitude, while the three approaches based on the multinomial distribution result in similar values, roughly one order of magnitude better than **Chernoff**. We also conducted experiments with different probabilistic distributions which in general lead to identical results.

In Figure 3(c), we compare the deadline miss probability of the three multinomial-distribution based approaches more closely. We can see that **Unify** performs very similar to **Pruning**, i.e., the error is in the magnitude of 10^{-9}. This is significantly smaller than the predefined *allowed error* of 10^{-6} for **Unify** in the experiments since: 1) execution mode equivalences classes are only merged for some of the tasks and the maximum error for each task may already be significantly smaller than 10^{-6}, and 2) the worst-case analysis in Sec. 6.2 is pessimistic. For **Approx** the error for Set 4 and Set 5 is in the magnitude of 10^{-5} and 10^{-7}, respectively, since only a subset of the points of interest is considered. In some rare cases even a larger relative difference could be observed.

Most importantly, all approaches we provide are able to deliver results even for large task sets, since the time needed to evaluate a single point in time remains still in the scale of minutes, i.e., in runs with 75 and 100 tasks one time point was evaluated on average in 621.6 and 791.1 seconds, respectively. Therefore, when a given task set needs to be analyzed, the approach can be used directly, especially since it is highly parallelizable due to the fact that different points in time can be analyzed completely individually. Hence, we suggest to first run *Hoeffding's* as well as *Bernstein's* bounds since they have a small runtime even for large task sets. If a sufficiently low deadline miss probability cannot be guaranteed from these bounds, we propose to run the multinomial-based approach with equivalence class union in parallel on multiple machines by partitioning the time points equally. We point out that it is especially helpful to use the union of equivalence classes if the periods of tasks differ largely, e.g., in automotive applications where periods often range from 1 to 1000 ms [15].

8 Conclusion

We provide a novel way to analyze the deadline miss probability of constrained-deadline sporadic soft real-time tasks on uniprocessor platforms where points in time are considered individually. Our main approach convolutes the equivalence classes of a task represented by the values of the multinomial distribution. The runtime of this approach can be improved by the detailed pruning technique without any precision loss. Furthermore, we present an approximation via unifying equivalent classes with a bounded loss of precision. In addition, we provide two analytical bounds based on the well-known Hoeffding's and Bernstein's inequalities which have polynomial runtime with respect to the number of considered time points. We demonstrate the effectiveness in the evaluations, specifically showing that our approaches scale reasonably even for large task sets.

References

1 IEC-61508 Edition 2.0. Functional safety of electrical / electronic / programmable electronic safety-related systems ed2.0. Technical report, International Electrotechnical Commission (IEC), 2010. URL: `http://www.iec.ch/functionalsafety/standards/page2.htm`.

2 Philip Axer and Rolf Ernst. Stochastic response-time guarantee for non-preemptive, fixed-priority scheduling under errors. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 172:1–172:7, 2013. `doi:10.1145/2463209.2488946`.

3 Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sept 2005. `doi:10.1109/TDMR.2005.853449`.

4 Slim Ben-Amor, Dorin Maxim, and Liliana Cucu-Grosjean. Schedulability analysis of dependent probabilistic real-time tasks. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 99–107, 2016. `doi:10.1145/2997465.2997499`.

5 Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Trans. Computers*, 53(11):1462–1473, 2004. `doi:10.1109/TC.2004.103`.

6 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. `doi:10.1007/s11241-005-0507-9`.

7 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2u: A general framework from k-point effective schedulability analysis to utilization-based tests. In *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, pages 107–118, 2015. `doi:10.1109/RTSS.2015.18`.

8 Kuan-Hsun Chen and Jian-Jia Chen. Probabilistic schedulability tests for uniprocessor fixed-priority scheduling under soft errors. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, France, June 14-16, 2017*, pages 1–8, 2017. `doi:10.1109/SIES.2017.7993392`.

9 José Luis Díaz, Daniel F. García, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*, pages 289–300, 2002. `doi:10.1109/REAL.2002.1181583`.

10 Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010. URL: `https://waters2016.inria.fr/files/2017/02/WATERS16-proceedings-final.pdf`.

11 Simon Foucart and Holger Rauhut. *A Mathematical Introduction to Compressive Sensing*. Springer New York, 2013. `doi:10.1007/978-0-8176-4948-7`.

12 Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. URL: `http://www.jstor.org/stable/2282952`.

13 Jie S. Hu, Feihui Li, Vijay Degalahal, Mahmut T. Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Compiler-directed instruction duplication for soft error detection. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 1056–1057, 2005. `doi:10.1109/DATE.2005.98`.

14 ISO-26262-1:2011. Iso/fdis26262: Road vehicles - functional safety. Technical report, International Organization for Standardization (ISO), 2000. URL: `https://www.iso.org/standard/43464.html`.

15 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

16 John P. Lehoczky, Lui Sha, and Yuqin Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, December 1989*, pages 166–171, 1989. doi:10.1109/REAL.1989.63567.

17 Dorin Maxim and Liliana Cucu-Grosjean. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 224–235, 2013. doi:10.1109/RTSS.2013.30.

18 Michael Mitzenmacher and Eli Upfal. *Probability and Computing - Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

19 Bogdan Nicolescu, Raoul Velazco, Matteo Sonza-Reorda, Maurizio Rebaudengo, and Massimo Violante. A software fault tolerance method for safety-critical systems: effectiveness and drawbacks. In *Integrated Circuits and Systems Design*, pages 101–106, 2002. doi:10.1109/SBCCI.2002.1137644.

20 Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliability*, 51(1):63–75, 2002. doi:10.1109/24.994913.

21 Nico Piatkowski, Sangkyun Lee, and Katharina Morik. Spatio-temporal random fields: compressible representation and distributed estimation. *Machine Learning*, 93(1):115–139, 2013. doi:10.1007/s10994-013-5399-7.

22 Nico Piatkowski and Katharina Morik. Stochastic discrete clenshaw-curtis quadrature. In *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York, USA, 19-24 June 2016*, JMLR: W&CP. JMLR.org, June 2016. URL: http://jmlr.org/proceedings/papers/v48/piatkowski16.html.

23 Semeen Rehman, Muhammad Shafique, Pau Vilimelis Aceituno, Florian Kriebel, Jian-Jia Chen, and Jörg Henkel. Leveraging variable function resilience for selective software reliability on unreliable hardware. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1759–1764, 2013. doi:10.7873/DATE.2013.354.

24 Bogdan Tanasa, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. Probabilistic response time and joint analysis of periodic tasks. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 235–246, 2015. doi:10.1109/ECRTS.2015.28.

25 Georg von der Brüggen, Jian-Jia Chen, and Wen-Hung Huang. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 90–101, 2015. doi:10.1109/ECRTS.2015.16.

26 Georg von der Brüggen, Kuan-Hsun Chen, Wen-Hung Huang, and Jian-Jia Chen. Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 303–314, 2016. doi:10.1109/RTSS.2016.037.

27 Georg von der Brüggen, Nico Piatkowski, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. Efficiently approximating the probability of deadline misses in real-time systems. Technical report, Department of Computer Science, TU Dortmund University, Germany, 2018. URL: https://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2018-brueggen-ecrts-deadline-miss-probability.pdf.

28 Dakai Zhu, Hakan Aydin, and Jian-Jia Chen. Optimistic reliability aware energy management for real-time tasks with probabilistic execution times. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 313–322, 2008. doi:10.1109/RTSS.2008.37.

Transferring Real-Time Systems Research into Industrial Practice: Four Impact Case Studies

Robert I. Davis
University of York, York, UK

Iain Bate
University of York, York, UK

Guillem Bernat
Rapita Systems Ltd., York, UK

Ian Broster
Rapita Systems Ltd., York, UK

Alan Burns
University of York, York, UK

Antoine Colin
Rapita Systems Ltd., York, UK

Stuart Hutchesson
Rolls-Royce PLC., Derby, UK

Nigel Tracey
ETAS Ltd. York, UK

Abstract

This paper describes four impact case studies where real-time systems research has been successfully transferred into industrial practice. In three cases, the technology created was translated into a viable commercial product via a start-up company. This technology transfer led to the creation and sustaining of a large number of high technology jobs over a 20 year period. The final case study involved the direct transfer of research results into an engineering company. Taken together, all four case studies have led to significant advances in automotive electronics and avionics, providing substantial returns on investment for the companies using the technology.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases real-time systems, industrial impact, automotive, avionics

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.7

Funding The work that went into writing this paper was funded, in part, by the ESPRC grant MCCps (EP/P003664/1). EPSRC Research Data Management: No new primary data was created during this study.

1 Introduction

This paper describes four impact case studies where real-time systems research has been successfully transferred into industrial practice. The four studies relate to:

1. Volcano: Guaranteeing the real-time performance of in-vehicle networks (Section 2).
2. RTA-OSEK and RTA-OS: The world's smallest commercial automotive real-time operating systems (Section 3).

3. RapiTime: A tool suite for analyzing the timing behaviour of real-time software (Section 4).
4. Visual FPS: The first CAA certified use of a fixed priority scheduler in an avionics system of the highest criticality (Section 5).

Preliminary versions of the first three impact case studies were informally published as white papers [30], [38], and [31]. Some of the content was also used in the University of York submissions to the Research Excellence Framework[1] (REF) assessment of UK Universities.

Each of the impact case studies is organised into the following subsections:

- *Impact Summary*: An executive summary of the impact achieved.
- *Background*: An overview of the industrial technology and practice prior to the research and development taking place.
- *Research*: An overview of the underpinning research, outlining the key insights and findings that contributed to the industrial impact.
- *Route to Impact*: The story of how the academic research was translated into viable commercial products.
- *Impact*: How the technology has been used, and by whom. (Note, due to non-disclosure agreements and commercial sensitivities, it is not always possible to give full details).
- *Beneficiaries*: Lists the beneficiaries and describes the benefits they have obtained by using the technology.
- *Future Challenges*: Sets out the main challenges in the specific technological area today.

The paper ends with a discussion of the key success factors and potential roadblocks. It is hoped that this information will be useful to others taking the exciting entrepreneurial step of trying to commercialise their research.

2 Volcano: Guaranteeing the Real-Time Performance of In-Vehicle Networks

2.1 Impact Summary

Controller Area Network (CAN) is a digital communications bus used by the automotive industry for in-vehicle networks. During 1994, research from the Real-Time Systems Research Group at the University of York introduced techniques that enable CAN to operate under high loads (approx. 80% utilisation) while ensuring that all messages meet their deadlines [65], [68], [67], [69]. This research led directly to the development of commercial products, now called Volcano Network Architect (VNA) and the Volcano Target Package (VTP). This Volcano technology (VNA and VTP) is now owned by Mentor Graphics. In recent years, VNA has been used to configure CAN communications for all Volvo production cars, with VTP used in the majority of Electronic Control Units (ECUs) in these vehicles, including the S40, S60, S80, V50, V70, XC60, XC70, XC90, C30, and C70; total production volume rising from 330,000 in 2008 to 530,000 vehicles per year in 2016. This Volcano technology is also used by Jaguar, LandRover, Aston Martin, Mazda, and the Chinese automotive company SAIC. It is used by the world's leading automotive suppliers, including Bosch and Visteon. It is also used by Airbus.

[1] `http://www.ref.ac.uk/2014/`

2.2 Background

Prior to the 1990s, cars used point-to-point wiring. This was expensive to manufacture, install and maintain. From 1991, the automotive industry began to use Controller Area Network (CAN) [21] to connect ECUs such as engine management and transmission control. Using this approach dramatically reduced the size, weight and complexity of the wiring harness, for example with CAN, a door system in a high-end car typically requires 4 wires, compared to 50+ with point-to-point wiring. The adoption of CAN led to significant cost savings and reliability improvements. It has supported a revolution in the complexity of automotive electronics, with the number of ECUs in a typical mainstream car increasing from 5-10 in the mid to late 1990's to 25-35 today.

CAN supports communications at typical bus speeds of 500Kbit/sec for powertrain applications and 125Kbits for body electronics. In a typical application, over 2000 individual signals (e.g. switch positions, wheel speeds, temperatures etc.) are sent in hundreds of CAN messages. There are deadlines on the maximum time that these messages can take to be transmitted on the bus. If a message fails to meet its deadline, then the reliability and functionality of the electronic systems can be compromised. This can lead to intermittent problems, and high warranty costs associated with 'no fault found' replacement of ECUs.

Messages queued by ECUs connected to a CAN bus compete to be sent on the bus according to their IDs, which represent their priority. Higher priority messages are sent in preference to those with lower priority. In the early 1990's, CAN messages were typically assigned IDs according to the data in the message, with a range of message IDs assigned to each supplier. Further, extensive testing was the only way of trying to verify that the messages would meet their deadlines. This was effective up to bus utilisations of about 30%; however, higher bus loads would result in deadline failures and intermittent problems.

2.3 Research

In 1994, three members of the Real-Time Systems Research Group at the University of York; Ken Tindell, Alan Burns, and Andy Wellings, introduced schedulability analysis of messages on CAN. This research [65], [68], [67], and [69] computed the longest time that each message could take from being queued by an ECU to being successfully transmitted on the bus and therefore received by other ECUs, referred to as the worst-case response time. This analysis enabled system designers to determine offline if all of the messages on a CAN bus could be guaranteed to always meet their deadlines during operation. This systematic approach was a significant improvement over the methods previously used in the automotive industry, which involved extensive testing, followed by hoping that the worst-case response time of every message had been seen.

This work also showed how to obtain optimal priority assignments for CAN messages. The research in [65] provided the fundamental analysis of message response times. This was extended in [68] to account for errors on the network, and integrated in [67] with information about the timing behaviour of the sending and receiving software. The analysis provided in [65], [68], [67], does not apply to all CAN hardware, some specific CAN Controller designs were shown in [69] to have relatively poor real-time performance, while others matched the requirements of the theory well. In 2007 research published by Davis et al. [33] corrected some flaws in the original analysis of CAN message response times, and was used by Mentor Graphics to check their Volcano Network Architect implementation. More recent research has addressed areas where the CAN controller hardware and the communications stack depart from the assumptions of the original research, such as non-abortable transmit buffers [54], and the use of FIFO [35], [36] and other work-conserving [37] queuing policies, as well as systems where peak network load is reduced using offset message release times [73]. Further research has studied robust priority assignment policies [32], including the case where some messages are constrained to have specific IDs [34].

2.4 Route to Impact

The initial research on schedulability analysis for CAN [65] was disseminated at the 1st International CAN Conference in 1994. As a direct result of this Ken Tindell was approached by Antal Rajnak, then working for Volvo Car Corporation. In April 1995, Ken Tindell and Robert Davis founded a start-up company called Northern Real-Time Technologies Ltd. (NRTT) to exploit the research in [65], [68], [67], and [69]. This company was contracted by Volvo Car Corporation to develop a CAN software device driver library and associated configuration tools [47], now referred to as the Volcano Target Package. Over the next two years, NRTT developed the Volcano Target Package through 4 major versions, and ported it to more than 10 different microprocessors used in the Volvo S80 and other automotive applications. At the same time, the message priority assignment policies and schedulability analysis techniques first introduced in [65], [68], [67], [69] were implemented in a CAN message configuration and analysis toolkit called Volcano Network Architect (VNA). The initial versions of VNA were developed by Kimble AB (a Swedish company founded by Antal Rajnak), working in conjunction with NRTT. Rights to the initial versions of the Volcano Target Package were transferred to Volcano Communications Technologies AB (a Swedish company founded by Antal Rajnak) which subsequently developed fully commercial versions of the Volcano technology (VNA and VTP), before being acquired by Mentor Graphics in 2005 [45]. From 1997 onwards the Volcano technology was used in the Volvo XC90, S80, S/V/XC70, S60, S40, and V50. When Volvo was bought by Ford in 1999, this technology was adopted by Ford Premier Automotive Group (Jaguar, Land Rover, and Aston Martin).

As part of its work on the Volcano technology, in 1995/6 NRTT consulted with Motorola, strongly influencing the hardware design used in the on-chip peripheral MSCAN controller [47], [43] (Section 4.2 of that document). This design used a 3 transmit buffer solution to ensure that the MSCAN controller can send out a stream of high priority CAN messages without releasing the bus – essential in achieving high bus utilisation without deadline failures. The 3 transmit buffer solution reduced the silicon area, and hence the unit cost of the hardware, compared to a 'full' CAN controller with 15 or 16 transmit buffers. This gave Motorola a competitive advantage, and reduced unit production costs for Volvo. Since 1997, microprocessors using MSCAN have been used in the door modules and other ECUs in a wide range of Volvo cars. In 2007, the analysis in [33] was used by Mentor Graphics to verify that the analysis provided by VNA [44] was correct. Further details of the Volcano Target Package and Volcano Network Architect can be found on Mentor Graphics' website [43], [44] with detailed descriptions given in [56].

2.5 Impact

The initial research [65], [68], [67], and [69] from 1994 was exploited in the design of CAN network layer software, called the Volcano Target Package (VTP), and network schedulability analysis tools, called Volcano Network Architect (VNA). The Volcano Target Package is deployed in ECUs, while Volcano Network Architect is used to configure networks and to ensure that the configurations obtained result in all messages meeting their time constraints. The research was initially exploited by a start-up company called Northern Real-Time

Technologies Ltd. (NRTT) that developed the first versions of the Volcano Target Package for Volvo Car Corporation (VCC) and worked in conjunction with Kimble AB to develop the first versions of Volcano Network Architect. Fully commercial versions of the Volcano technology (VNA and VTP) were later produced by Volcano Communications Technologies AB, which was sold to Mentor Graphics in 2005 [45].

In 2018, the Volcano Target Package is available for more than 30 different ECU micro-controllers [48], including: Fujitsu 16LX, FR Series; Hitachi H8S, SH7055, SH7058; Infineon C16x, TC179x, TC176x, XC800, XC2000; Renesas M16C, R32C/M32C; Freescale HC08, HC12, MC683xx, MPC5xx, MAC71xx; S12, S12X, MPC55xx, MPC 56xx; Mitsubishi M32R, MC32C; PowerPC; National CR16; NEC V85x, 78K0; ST Microelectronics ST9, ST10; Texas Instruments TMS470; Toshiba TMP92/TMP94.

Since the introduction of the Volvo S80 in 1998, Volcano Network Architect has been used to configure CAN communications in all new Volvo production cars, with the Volcano Target Package used in the majority of Electronic Control Units (ECUs) in these vehicles. During the period 2008 – 2016, this includes the S40, S60, S80, V50, V70, XC60, XC70, XC90, C30, and C70; total production volume 330,000 to 530,000 vehicles per year [70].

The Volcano technology (VNA and VTP) is also used by Jaguar, LandRover and Aston Martin. Since 2007, this technology has been used in its own branded vehicles by the Chinese automotive giant SAIC [46]. In 2012, Mazda announced that they would be using Volcano technology in order to make more efficient and reliable use of CAN in vehicles featuring their "Skyactiv Technology" [49]. The Volcano Target Package is also used by the world's leading automotive suppliers, including Bosch and Visteon.

2.6 Beneficiaries

Volcano Network Architect, and the Volcano Target Package software that conforms to its assumptions, enable system architects at automotive manufacturers to configure in-car networks using CAN such that all of the messages are guaranteed to meet their deadlines at bus loads (utilisations) of up to approx. 80%. This compares with a maximum of approx. 30% using the approach otherwise prevalent in industry, where message IDs (priorities) are assigned in groups according to ECU supplier, and extensive testing and a large engineering margin for error is used to gain some confidence that message deadlines will be met. Achieving higher bus utilisation enables far more functionality to be supported using the same bus speed and communications hardware, providing those automotive manufacturers that adopt this technology with a key competitive advantage. With higher bus utilisations, more ECUs can be connected to the same network, and the network can support a larger number of signals and messages. Wiring complexity can be reduced, with fewer connectors, increased reliability, and improved brand image. Further, there is enhanced support for the addition of lucrative 'software-only' options.

These benefits are summarised in the Volvo Technology Report [25]:

"The advantages to Volvo of the development and application of Volcano include: Production cost benefits due to high bus efficiency (four times as many signals can be transmitted at half the baud rate). Development cost benefits (in the form of a single, proven implementation which is much cheaper than multiple implementations by suppliers and conformance testing by Volvo). Improved network reliability, resulting in higher product quality. Reduction in Volvo´s test load. Reduction in supplier´s test load. High degree of flexibility (useful in many situations). Recognition of the real-time problem (Volvo developed solutions before the problem had been recognised generally)".

Although [25] was written in 1998, the benefits of using this technology remain the same today. They are highlighted in 2006 [46] in relation to the Chinese automotive giant SAIC:

"By using Volcano, network design is made easy and predictable, guaranteeing data communication, which reduces the verification effort to almost zero and eliminates warranty and change costs caused by networking issues."

Similarly, in 2012 [49]:

"Mazda's use of VNA has enabled significant improvements in network efficiency and reliability." . . . "This procedure increased the network utilization and significantly reduced the testing requirements and time".

The research on CAN also led directly to the design by Motorola (now Freescale) of a low-cost on-chip CAN peripheral MSCAN [25], [55] that requires less silicon area than a 'full' CAN controller, and so reduces unit costs in production.

In summary, car manufacturers and their sub-suppliers have benefited from the research in terms of reductions in development, production, and warranty costs. Development costs have been reduced via improvements in the time taken to verify network timing behaviour, reducing the cost of testing, and time-to-market. Production costs have been reduced via the ability to run in-vehicle networks at high loads while ensuring that all message deadlines are met. This has enabled increasing amounts of functionality to be accommodated using the same low cost CAN hardware. Improvements in network reliability, via off-line guarantees that messages will always meet their deadlines, have reduced warranty costs, in particular, costly 'no fault found' ECU replacement. In a competitive marketplace, benefits to the car manufacturers have been passed on to the consumer, in terms of vehicles that are less expensive, yet have more functionality, and better reliability.[2]

2.7 Future Challenges

The future challenges in this area originate from:

- The use of multiple networks, often of different types, with signals and messages transferred between them. Here, gateway policies, signal packing, and prioritization all influence end-to-end response times.
- The need to efficiently utilise network bandwidth. Message priority assignment, offset assignment and signal packing all influence the useful bandwidth that can be employed before messages begin to miss their deadlines. The recent CAN-FD protocol increases network speed during data transmission.
- The need to deal with legacy applications and ECUs. It is rare that any automotive system begins with a clean sheet design.
- Security issues. Connection of in-vehicle networks to the internet raises significant security concerns.

3 RTA-OSEK and RTA-OS: The World's Smallest Automotive Real-Time Operating Systems

3.1 Impact Summary

Research [2], [3], [24], [4], [66] published by the Real-Time Systems Research Group at the University of York from 1993 to 1995 was exploited in 1997 to design an exceptionally efficient Real-Time Operating System (RTOS), used in automotive Electronic Control Units (ECUs), and its associated schedulability analysis tools . By 2017, the RTOS had been deployed

[2] Accounting for inflation, the average car purchased in the USA in 2015 was less expensive than the average car purchased during 1990 ($25,300 versus $27,300).

in over 1.25 billion ECUs. It has been standardised upon by many of the world's leading automotive powertrain systems and chassis electronics suppliers, and is used in cars produced by nearly all of the world's major car manufacturers.

3.2 Background

In real-time embedded systems, such as the ECUs used in vehicles, system functionality is decomposed into multiple software tasks running on a microprocessor. The system requirements place time constraints on these tasks. Hence a task may be required to execute every 10 milliseconds, read and process data from sensors, and output its results within a specific time constraint or deadline. When there are multiple tasks with different periods and deadlines running on the same microprocessor, an RTOS is needed to schedule when each task should execute. It is essential that all of the tasks are guaranteed to meet their deadlines during operation; otherwise the system may suffer from intermittent timing faults that compromise its functionality and reliability.

Given the complex behaviour of these systems, it is impossible to obtain a 100% guarantee that tasks will always meet their deadlines via testing. Instead, a rigorous scientific and systematic solution to this problem is schedulability analysis; a set of techniques used to determine off-line if each task can be guaranteed to meet its deadline under a specific scheduling policy. Schedulability analysis is used to compute the worst-case response time, the longest time that can elapse from a task being released to it outputting its results and completing execution. If this is less than the deadline, then the task can be guaranteed to always meet its time constraints.

3.3 Research

In the early 1990's seminal research into schedulability analysis [2], [3], [24], [4], and [66] for fixed priority pre-emptive scheduling, originally called Deadline Monotonic Schedulability Analysis but now widely referred to as Response Time Analysis, was introduced by the Real-Time Systems Research Group at the University of York.

This analysis is applicable to fixed priority scheduling, and a task model that accurately accounts for the detailed timing behaviours of tasks in automotive systems. These timing behaviours include: tasks that are invoked sporadically (i.e. with minimum inter-arrival times, but not necessarily strictly periodically in time); tasks with deadlines that are less than their periods and prior to completion [2], [3] — accounting for tasks that need to make a response prior to their next invocation to avoid buffer overruns, and to carry out further computations after a response has been made, in preparation for the next cycle; tasks with offset release times [4] – used as a means of avoiding peak load in short time intervals; tasks with jittered released times [66] – that are triggered by the arrival of messages that can take a variable amount of time to be transmitted, and tasks that share resources [2], [3] – such as data structures and peripheral devices used for communication. The analysis also accounts for the overheads of a well-designed RTOS [24].

This research therefore introduced for the first time, schedulability analysis that could be applied in practice to commercial real-time systems, providing a rigorous approach to obtaining timing correctness. This was recognised in the EPSRC International Review of Computer Science undertaken in 2002: *"These researchers are credited with a significant body of research in static real-time scheduling theory. They have also demonstrated how to employ these theoretical results in practice, by accounting for networking and operating system overheads. This combination of theory and practice has resulted in important and practical applications of their work."*

The techniques developed also built upon other important research contributions such as the Stack Resource Policy [7] for resource locking.

3.4 Route to Impact

In 1997, Robert Davis and Ken Tindell co-founded a company called Northern Real-Time Applications (NRTA) Ltd., with the aim of developing an RTOS and schedulability analysis tools specifically tailored to automotive applications that use low cost microcontrollers.

There were two fundamental design goals:

1. The real-time behaviour of systems built using the RTOS must be fully analysable using schedulability analysis tools. In other words the behaviour of the RTOS must match the assumptions of the underpinning schedulability analysis techniques.
2. The memory and execution time overheads of the RTOS must be significantly less than those of any other RTOS available for use in automotive applications.

Robert Davis led the team that developed the SSX5 RTOS and associated schedulability analysis tools (originally called the "Time Compiler", later "Real-Time Architect (RTA)" and "RTA-OS Analysis Visualizer"). The schedulability analysis tools implemented Response Time Analysis as introduced in [2], [3], [24], [4], and [66]. The SSX5 RTOS was developed precisely to meet the assumptions of this analysis. The execution time overheads (of preemption, task termination, interrupt service routine entry and exit, and all system calls that can cause context switches) were minimised and made constant, independent of the number of tasks, allowing them to be accurately measured and integrated into the schedulability analysis implemented in Real-Time Architect.

The memory overheads of applications built on SSX5 were radically reduced by comparison with other automotive RTOSes. This was achieved via the use of single-stack execution and compile time, i.e. off-line, configuration of the RTOS data structures to minimise RAM usage. NRTA attracted significant venture capital funding in 1998 (£1 million from 3i) and again in 2000 (£9.2 million from 3i and TecCapital). In 2001, the company changed its name to LiveDevices Ltd.[3]

In March 2003 LiveDevices was sold to ETAS GmbH, a wholly owned subsidiary of Robert Bosch GmbH. The reason for the trade sale was that Robert Bosch had benchmarked RTA-OSEK and found it to be significantly more efficient than its subsidiary's Ercos RTOS. Rather than attempt to write a new OSEK RTOS from scratch and compete with LiveDevices, ETAS chose to buy the company, bringing the RTA-OSEK technology and the 20+ LiveDevices engineering team in-house.

3.5 Standards

During the development of the SSX5 RTOS, the automotive industry was working on standards via the OSEK organisation. As a Technical Committee Member of OSEK, NRTA influenced the OSEK OS standard [52] ensuring that the basic conformance classes (BCCx) could be achieved with a single-stack RTOS, leveraging the execution time and memory savings which that approach facilitates [29]. NRTA modified the SSX5 RTOS to comply with the OSEK standard, in the process renaming the product: RTA-OSEK.

[3] This name change was marketing led as the company was also developing Internet-of-Things technology, including a very small TCP-IP stack. This technology was not commercially successful; in hindsight it was around 10 years ahead of its time.

Subsequently, ETAS, as a premium partner of the AUTOSAR (AUTomotive Open System ARchitecture) partnership [6], have been heavily involved in specifying the AUTOSAR operating system standard [5], which extends the OSEK operating system standard. ETAS derived an AUTOSAR compliant RTOS called RTA-OS from RTA-OSEK [42].

3.6 Impact

As of 2018, ETAS sells two versions of the RTOS, RTA-OSEK and RTA-OS compliant with the OSEK and AUTOSAR operating system standards respectively.

The RTOS is currently available for more than 50 different ECU microcontrollers [42] including: Renesas: V850E, SH2, SH2A, H8S, H8SX, M16C; Xilinx Microblaze, PPC405 Core; Texas Instruments TMS470P, TMS570P; Infineon Tricore TC17x6, C166, XC2000; Freescale Star12, MPC555, MPC55xx, S12X, MPC56x, HC12X16, HC08, HCS12; Fujitsu 16LX; Analog Devices Blackfin, STMicroelectronics ST30, ST7, ST10.

RTA-OS is also available for the following multi-core processors: Infineon Aurix, Freescale MPC57xx, Renesas RH850, STMicroelectronics SPC57x, and the Xilinx Zynq-7000 family.

ETAS customers for the RTOS cover a wide range of application areas within Automotive Electronics. It has been standardised upon (used by default in all ECUs) by many of the world's leading automotive powertrain systems and chassis electronics suppliers, and is used in cars produced by nearly all of the world's major car manufacturers. By 2017, the RTOS had been deployed in over 1.25 billion ECUs. This number is increasing at a rate of between 1 and 2 million new ECUs per week.

3.7 Beneficiaries

Use of the RTOS and its associated schedulability analysis tools has benefitted automotive manufacturers and their Tier 1 suppliers in the following ways:

(i) A reduced memory footprint means that cheaper microcontroller variants with smaller on-chip RAM / Flash memory can be used. (The code size of RTA-OS is typically in the range 1 Kbytes to 1.5 Kbytes depending on the processor – making it, to the best of our knowledge, world's smallest commercial AUTOSAR OS.[4] This has reduced unit costs in production.

(ii) The very low execution time overheads[5] of the RTOS mean that more functionality can be included on a given low cost microprocessor reducing costs by avoiding the need for hardware upgrades to more capable but expensive devices.

(iii) A reduction in the time spent debugging intermittent timing issues. Schedulability analysis and appropriate use of proven real-time mechanisms have enabled off-line analysis of task response times, reducing system integration time and testing effort, and improving reliability.

For these reasons the world's major ECU suppliers and car manufacturers have adopted this technology. In a competitive market, some of these benefits will have been passed on to their customers in the form of cheaper, more reliable vehicles.

The Automotive Electronics market is both huge and highly competitive, with electronics now contributing 15-30% of overall vehicle production costs. For the reasons given above, the world's leading Automotive OEMs and Tier-1 suppliers have adopted the RTA-OSEK and RTA-OS operating systems. They have done so for the substantial benefits it brings to them and to their customers. The technology has led directly to the creation and sustaining, over a period of more than 15 years, of a large number of high technology jobs in York, UK.

[4] See section 8 of [41] for an example of the RTA-OS ROM and RAM usage.

[5] See section 8.5.1 of [41] for an example of the overheads in CPU cycles and nano-seconds for different types of context switches, along with diagrams explaining the precise measurements.

3.8 Future Challenges

Automotive systems are now moving towards implementations on multi-core hardware. This leads to the following challenges:

- The use of high performance multi-core hardware, with shared interconnects and other shared hardware resources makes it significantly more difficult to obtain an accurate understanding of the execution time behaviour of tasks, due to issues of cross-core interference. In some cases this interference can be so severe that guaranteed performance using multiple cores may be no better than could be obtained by utilising just one core.
- Synchronization via non-preemptive execution is no longer effective in a multi-core environment, creating significant difficulties in porting legacy applications. More complex and potentially substantially less efficient synchronization and locking mechanisms need to be employed, and large amounts of code potentially re-factored.
- High performance multi-core hardware means that it is cost effective to integrate different applications that would otherwise have run on independent ECUs onto the same hardware platform. These different applications have different criticality levels which leads to a host of interesting problems. Mixed criticality systems are currently a hot topic in real-time systems research [22].

4 RapiTime: A Tool Suite for Analyzing the Timing Behaviour of Real-Time Software

4.1 Impact Summary

Research [18], [16], [28], [27], [17] from the Real-Time Systems Research Group at the University of York published in 2002-2005 resulted in a measurement-based Worst-Case Execution time (WCET) analysis technology now called RapiTime, which was transferred to industry via a spin-out company, Rapita Systems Ltd, founded in 2004. The technology enables companies in the aerospace, space and automotive industries to reduce the time and cost required to obtain confidence in the timing correctness of the systems they develop. The RapiTime technology has global reach having been deployed on major aerospace and automotive projects in the UK, Europe, Brazil, India, China, and the USA. Key customers include leading aerospace companies as well as major automotive suppliers.

4.2 Background

Determining the longest time that software components can execute on a microprocessor, referred to as the Worst-Case Execution Time (WCET), is a key issue in the development of real-time embedded systems in the aerospace and automotive industries. Here, intermittent timing failures caused by software exceeding its budgeted execution time can lead to operational problems, reliability issues, and in some cases catastrophic consequences. In these applications the WCET of software components needs to be tightly bounded to avoid the need to over-provision hardware in terms of faster, but more costly processors.

Prior to this research, there were two main approaches to WCET estimation; end-to-end measurement and static analysis. End-to-end measurement techniques insert profiling code into the software. During testing this profiling code records the end-to-end execution time of each invocation of each software component. End-to-end measurement alone typically under-estimates the WCET, and provides little confidence that timing constraints will always be met during operation. Static analysis techniques analyse the software object code and compute the WCET using a model of the timing behaviour of the microprocessor. This is done without running the code. Using static analysis alone has the disadvantage that the computed WCETs depend on the model of the processor and its hardware acceleration features; as processor technology advances this becomes more complex, and expensive to determine, and in some cases may not be possible due to a lack of detailed information.

4.3 Research

During the NextTTA project (2002 to 2004), Guillem Bernat, Antoine Colin, Stefan Petters, and Alan Burns developed a set of hybrid and probabilistic techniques for WCET analysis [18], [16], [28], [27], and [17], now referred to as RapiTime. The RapiTime approach combines static analysis of the structure of the source code with timing measurements taken during testing, which record the execution time of short sub-paths through the code. RapiTime recognises that the best possible model of an advanced microprocessor is the microprocessor itself and therefore uses online testing to measure the execution time of short sub-paths in the code. By contrast, offline static analysis is the best way to determine the overall structure of the code and the paths through it. Therefore RapiTime uses path analysis techniques to build up a precise model of the overall code structure and determine which combinations of sub-paths form complete and feasible paths through the code. Finally the measurement and path analysis information are combined using mathematical techniques to compute WCETs in a way that accurately captures the execution time variation on individual paths due to hardware effects.

This novel approach combines the advantages of both measurement and static analysis techniques while avoiding the majority of their drawbacks. Unlike static analysis, it does not require the expensive and time consuming production of a precise timing model for each new microprocessor variant and its hardware acceleration features, and so is portable to a wide range of different microprocessors. RapiTime is also viable when the only accurate timing model that is available is the microprocessor itself. Further, RapiTime does not require the manual annotations that static analysis alone needs to establish essential information about control flow. This reduces the amount of engineering time required before meaningful results can be obtained, and removes a potential source of errors. Compared to measurement, RapiTime is able to identify the worst-case path and compute the overall WCET of software components from the WCETs of sub-paths when not all of the complete paths through the code have been executed. This significantly reduces the amount of testing required to verify timing correctness. For a detailed discussion of the advantages / disadvantages of static and measurement based approaches to timing analysis, the interested reader is referred to [71].

4.4 Route to Impact

During the EU FP5 NextTTA project, Guillem Bernat, Antoine Colin, Stefan Petters, and Alan Burns, introduced research on hybrid measurement-based WCET analysis. This approach combined both measurement and static analysis techniques to accurately estimate the WCET of complex software components running on advanced microprocessors. As part of the project, they also developed a prototype WCET analysis tool called pWCET [17]. This tool was evaluated on an Audi drive-by-wire system. Audi was an industrial partner in the NextTTA project. Audi's expression of interest in pWCET and its capabilities led directly to the formation of a spin-out company to transfer this technology into industry.

In 2004, Guillem Bernat, Ian Broster, Antoine Colin, and Robert Davis founded a spin-out company called Rapita Systems Ltd. (www.rapitasystems.com) to commercialise the technology and bring it to market. All rights to the technology and prototype tools were transferred to the company by the University of York in exchange for shares in the company.

In 2005, Rapita Systems received £200k of funding from Viking Investments Ltd. and an associated group of Business Angels [72]. Following the initial technology transfer, the pWCET prototype was re-implemented as a commercial quality tool and re-branded as RapiTime. RapiTime has since been extended to support analysis of systems written in C++ as well as the C, and Ada programming languages. RapiTime has been complemented by RapiCover, an on-target structural code coverage tool, and RapiTask, a tool which enables users to visualize high-level system scheduling, locate rare timing events such as race conditions, and verify actual timing behaviour. Both RapiCover and RapiTask use the underpinning RapiTime technology for code instrumentation and analysis. RapiTime, RapiCover, and RapiTask form part of the Rapita Verification Suite (RVS).

In 2006, BAE Systems used RapiTime on the Hawk Advanced Jet Trainer project [60]. Here, RapiTime was used to identify opportunities for WCET reduction, thus creating headroom for new functionality to be added to the system, while avoiding the need for a costly hardware upgrade. Using RapiTime, BAE identified that just 1% of hundreds of thousands of lines of code contributed 29% of the overall WCET. By focusing optimisation efforts on this 1% of the code, they were able to reduce the WCET by 23% [19]. Further, RapiTime was quantified as being able to identify timing problems with less than 10% of the effort of previous approaches, potentially saving months of work. As a result Rapita received a BAE chairman's award for Innovation in the category Transferring Best Practice.

In April 2016, Rapita Systems Ltd. was sold to Danlaw Inc. in a trade sale [64]. (Danlaw is a global connected vehicle, automotive electronics and embedded engineering enterprise with facilities in USA, Europe, India, and China).

4.5 Impact

As described in the previous section, research from the Real-Time Systems Research Group at the University of York was exploited in the development of an innovative Worst-Case Execution time (WCET) analysis technology called RapiTime. This technology was transferred to industry via the formation in 2004 of a spin-out company; Rapita Systems Ltd.

RapiTime has been deployed on, and is in continuous use on, a number of major long-term space, aerospace and automotive projects world-wide, examples include: Flight Control Computers [61] and FADECs (Full Authority Digital Engine Control); Alenia Aermacchi (Italy) Flight Control System for the M-346 military transonic trainer [62] (since 2010), and various projects for the European Space Agency (ESA) (since 2008). RapiCover has also been qualified for MC/DC coverage of the DO-178B DAL A Flight Control System of the M-346 [63]. Rapita has also won significant export orders to China via its distributor Cinawind.

4.6 Beneficiaries

RapiTime enables companies in the aerospace and automotive electronics industries to reduce the time and cost required to obtain confidence in the timing correctness of the systems they develop. It provides a cost-effective means of targeting software optimisation, such that new functionality can be added to existing systems without the need for expensive hardware upgrades. Further, RapiTime is portable across a wide range of different microprocessors, meaning that companies can use the same technology across multiple projects without the need for re-training or adoption of multiple solutions.

A major aerospace supplier described the benefits of using RapiTime to identify timing problems during continued development of a Flight Control System as follows: *"The biggest benefit that RapiTime brought to our development process was just how quickly we could get comprehensive timing measurements from our tests. Not only did we reduce our effort* *requirements for the testing, but we could use our results in ways that were infeasible before. It is now significantly faster for us to identify a timing issue, update the software to resolve the issue, test the updated software and verify that it's fixed"* - Wayne King, Engineering Fellow – 30th July 2009.

Without RapiTime, the timing measurement and analysis process needed to determine WCETs has to be done manually. This is a painstaking and error prone process that takes considerable time and effort. It also needs to be repeated when changes are made to the application software. Further, the manual process provides no information about the worst-case path, or the contribution of different sections of code to the WCET. This makes code optimisation an ad-hoc, ineffective and inefficient process, as optimising for the worst-case is very different from optimising for the average case.

Alenia Aermacchi engineers working on the M-346 Flight Control System in 2010 said, *"the main advantage [of using RapiTime] is the possibility to identify software bottlenecks that can be subject to optimisation. Without RapiTime the mandatory code optimisation would have been done without the knowledge of where to concentrate the efforts."* [62]. Overall, *"Using RVS, customers have cut the worst-case execution time of large scale, legacy applications by up to 50% with only a few days effort, and significantly reduced unnecessary testing and instrumentation overheads"* [61].

Embraer used Rapita's RVS tool suite to capture WCET and stack usage data for DO178B level A Flight Control Systems (FCS) [58]. Because it was not necessary to manually design a test case for the worst-case path, significant effort was avoided, saving time and money. *"We have successfully shown the viability of using RapiTime to measure WCET. It was able to support our hardware platform and once the system was set up, the analysis method could be repeated with relative ease. With the WCET results, time partitioning was easily configured in the platform for the FCS application. Processing resources could be optimized by tightening the time window, even leaving some room for future expansion"*, Felipe Kamei, Embraer [58].

Infineon asked Rapita Systems to use RapiTime to look for optimization opportunities to reduce the execution times of the SafeTCore drivers which form part of Infineon's PRO-SIL concept. (These drivers are functionally independent of micro-controller hardware and can run on all micro-controllers in Infineon's TriCore family). The timing analysis part of the case study focused on 5 Tricore functions, giving up to 43.9% reduction in the WCET [59].

4.7 Future Challenges

In the next 5 to 10 years, complex multi-core and many-core systems will present an extreme challenge in terms of the difficulty involved in obtaining tight worst-case execution time estimates.

- The use of high performance multi-core hardware, with shared interconnects and other shared hardware resources means that execution times can be heavily impacted by contention over shared resources by co-running tasks on other cores. WCETs obtained in isolation can thus be substantially optimistic, when compared to the values for an operational system that runs applications on multiple cores.
- Obtaining context independent WCETs presents a significant challenge, since it is not obvious what pattern of co-runner execution will produce the most interference on shared resources. Even if a fully context independent WCET can be found, then it may be substantially pessimistic, compared to the actual WCET in the context of the deployed system.

Hardware and software techniques which ensure isolation from the effects of co-runner contention may be effective here. Another promising approach, is to use measurement-based probabilistic timing analysis techniques.

5 Visual FPS: The First CAA Certified use of a Fixed Priority Scheduler in an Avionics System of the Highest Criticality

5.1 Impact Summary

Fixed Priority Scheduling (FPS) research from the Real-Time Systems Research Group at the University of York was exploited to make the design and maintenance of the software in Rolls-Royce's Full Authority Digital Engine Controllers (FADEC) more efficient in terms of resource usage and cost [51]. The most notable benefit to Rolls-Royce was that they did not need to procure new more powerful processing hardware for a project where the processor which they normally used had run out of capacity. Unlike conventional systems, a processor board for a safety-critical avionics system costs thousands of pounds and has lead times of between one and two years. Changing the hardware would have meant that the software team would not have had access to the actual target until very late in the development, and the project would most likely have been late incurring significant penalties. Overall, the risk of such a change was unacceptable.

5.2 Background

FADECs are responsible for the control and monitoring of aircraft engines. They play a vital role in not only the reduction of hazardous events related to the aircraft engine, but also the overall safety and certification of the aircraft. FADECs do much more than inject fuel and control the engine. They help keep both the aircraft's cabin and fuel at the right temperature, receive information and commands from the cockpit and send back information, they also log information about the engine for future maintenance, and play other vital roles such as helping the aircraft brake on landing via the use of thrust reversers. Over time, this has led to an increase in the amount of software in the system, most of which is hard real-time. The timing requirements that have to be guaranteed span not only deadlines, but also tight jitter requirements. These requirements have to be guaranteed for both independent tasks and precedence constrained tasks, referred to as transactions.

For many years, the avionics industry used static scheduling to try and meet the timing requirements. Despite the use of automated tools, e.g. search-based algorithms for choosing task attributes [23], a number of issues remained unresolved. Firstly, the static scheduler places restrictions on the timing requirements, for example minimising the number of periods used, and making them harmonics of a single period. Secondly, and more importantly, the schedules become hard to maintain as the number of tasks and their execution times change as the system's build progresses. Here, the software in the FADEC is slowly integrated through a number of carefully considered phases; however, most approaches to designing the schedule do not consider maintenance and ensuring the minimum change between synthesised schedules [40] nor the similarity of schedules between functional modes [39]. This is significant as changes to the order in which tasks execute changes both the timing and functional characteristics which makes regression testing a much larger and hence more costly activity. Finally, as with many systems the processor was almost fully utilised (approaching 100%), making meeting the timing requirements difficult. Therefore as part of a University Technology Centre based at the University of York, a significant body of research was initiated to consider how fixed priority scheduling could be migrated into the development of the FADEC software. This work needed to be performed in the context of DO-178B [57] (which was later replaced by DO-178C), and the software written in SPARK 95, which is a subset of Ada supported by verification tools [8].

5.3 Research

In the early 1990s there was significant work by a number of research groups on fixed priority preemptive scheduling, including by the Real-Time Systems Research Group at the University of York. This led to a number of approaches to both priority assignment and schedulability analysis [2] (see also the material referenced in Section 3.3). This analysis largely covered independent tasks and systems without overheads. Similar to the majority of works in this area, it did not consider how the scheduling policies used affect software development.

The first consideration, in 1995/6, was to establish a detailed understanding of why the software was currently developed the way it was and what the implications of any change would be [14]. This work was undertaken by Iain Bate, under the guidance of Alan Burns, as part of a long-term project funded by Rolls Royce., The most obvious and important conclusion of this work was that a non-preemptive approach should be used. There were three main reasons for this. Firstly, the existing software was written in a non-preemptive fashion, therefore the minimal change and the one that carried least risk was to stay with that approach. This also gave the easiest reversion path in case the Aviation Authorities, who regulate the certification of systems, rejected the final safety case. Secondly, fixed priority scheduling was already causing a significant increase in the number of paths through the software, since for the majority of tasks there was no longer a deterministic order of execution. As functional verification is much more costly than timing verification then managing its financial cost was deemed more important. Finally, the potential overheads and the complexity of the Real-Time Operating System (RTOS) was higher with a preemptive scheduler.

Given the decision to employ a non-preemptive scheduler, the lack of substantial work in the academic literature, and a need to keep the overheads as low as possible, the next piece of work looked at how the RTOS should be designed and analysed. This led to a detailed assessment of how the existing RTOS was designed, the minimal migration path possible while reusing the existing mechanisms for timing watchdogs, and how the timing overheads could be analysed. This assessment resulted in a new task release mechanism that ensured the overheads were O(1) [1]. The final technical challenge was how to take the complex timing requirements of the FADEC and map these onto a set of task attributes. The approach taken was based on the use of offsets to control the jitter within the system [11] and setting independent task deadlines such that the transactional (precedence) requirements were met [12]. The overall research and strategy [9], [13] were published in 1998.

5.4 Route to Impact

A key aspect of the work was engaging with Rolls-Royce's technical staff to understand how they develop systems and how the adoption of fixed priority scheduling would affect their work. This meant Iain Bate spending extensive periods of time within Rolls-Royce not only on fixed priority scheduling for FADECs but also gaining their trust by helping out with other immediate technical concerns [15]. As part of this strategy, a champion within the company was created who could not only guide the research but would help pull it into the organisation and then own it after the research was complete [51]. Four other important activities were undertaken. Firstly, FADECs have a need for regulatory approval and hence once some key decisions were taken a Preliminary Safety Case was established in 1997 which could then be discussed with both Rolls-Royce's engineers as well as representatives from the Civil Aviation Authority (CAA) [53]. The result of this step was a clear picture of the implications of the technology and company approval to continue the investigation. Secondly, a cost-benefit analysis was undertaken to not only understand the financial implications for the whole engine development but also the risks. Thirdly, a RTOS was written in SPARK and a qualified tool, *VisualFPS*, produced for task attribute assignment and schedulability analysis. Finally, Rolls-Royce placed patents [10] on the work in 1997 and the University of York's legal team addressed potential litigation issues that could emerge.

After the technology was "adopted" by the project team for its first project, the technology was abandoned as the project fell behind schedule and any unnecessary risk was cut. The links with Rolls-Royce then went quiet until the next project reached a point at which a hardware re-design would be necessary without VisualFPS. This led to its adoption in 2003 and its subsequent on-going use. Notably during this time there was little contact between the University of York and Rolls-Royce due there being no reason to change the adopted approach. This demonstrates that a robust future-proofed approach had been developed.

The FADEC software including the fixed priority non-preemptive scheduler were certified to DAL-A by the CAA in December 2002 for the Tay 611-8C engine. (Note, the Honeywell Digital Engine Operating System (DEOS), which uses fixed priority preemptive scheduling is noted as being *"contracted for use in 6 FAA certified jet products"* in 2001 [20]).

5.5 Standards

As previously stated, the FADEC software is produced in accordance with DO-178C/ED-12C. This provides a set of objectives for the software development and verification process, and requires evidence to be produced by the development organisation to demonstrate compliance as part of the engine/aircraft certification activity. However, DO-178C/ED-12C is a process-based guidance document, and the certification objectives focus on compliance to requirements and conformance to standards. There is little guidance on specific product performance aspects, for example. A common myth is that the standards and the regulatory authorities demand that timing requirements are always met and static analysis is mandatory [50]. Instead current best practice, arguments and evidence for acceptable safety, and graceful degradation when the inevitable failures occur is what matters. Predictability of system-level performance is the overriding principle. Safety experts including the regulatory authorities also provide some steer towards achieving these things in the presence of new technologies through position papers by the Certification Authorities Software Team. For example, CAST 20 [26] gives guidance to those considering using processors with caches. For the FADEC software, this meant supporting the relevant parts of the safety argument through a No Less Safe Than Before approach, i.e. that the new technology did not introduce new hazardous events or make existing ones more likely or more severe. It is worth noting that a significant influence in the regulatory authority's decision was that the scheduling approach came with mathematical analysis that had been peer reviewed in top international conferences by specialists in the field.

A further complication was that the certification regime differs between Europe and the US. The European Aviation Safety Agency (EASA) tend to regulate civil aviation certification centrally, using a team of experts employed by EASA. The Federal Aviation Agency (FAA) in the USA operate a "Designated Engineering Representative" (DER) scheme, where DERs are licensed by the FAA but employed by the applicant companies. This can lead to variation in how the certification rules and compliance evidence requirements are applied.

5.6 Impact

The impact of this work was easy to gauge as millions of pounds were saved by avoiding the need to change the hardware platform and hence the attendant risk of delivering the aircraft engine late. Since then, the technology has been used within Rolls-Royce without the need for any updates. The benefits of this are much harder to quantify. Outside the FADEC, fixed priority scheduling has now been widely adopted in avionics and other critical systems including automotive.

5.7 Beneficiaries

The beneficiaries of this successful technology transfer are many fold. External to Rolls-Royce, its questionable whether Rolls Royce adopting fixed priority scheduling made it easier for others, or whether the change of scheduling practice was inevitable, but it certainly didn't harm. Internal to Rolls-Royce, the benefits are clear. Financially it saved a significant amount of money some of which was easy to quantify, i.e. the immediate saving of not procuring a new processing platform. Further, there is the longer term benefit of an automated tool for synthesising the scheduler and producing analysis results. The change of scheduler has also given Rolls Royce's engineers the freedom to specify different timing requirements, which has allowed both better control of jitter and more flexible choices of task periods. Both of these have enabled improvements in engine performance and reductions in processor utilisation.

5.8 Future Challenges

The future challenges for Rolls Royce are significant.[6] As of 2018, they are currently undertaking their most ambitious engine re-design in more than 30 years. This re-design is targeted at dramatic improvements in engine efficiency, towards the industry wide Clean Skies initiative, while at the same time reducing costs. This has led to significant interest in mixed-criticality scheduling [22], cheaper to implement and maintain communications, and more advanced control and monitoring systems. In response, Rolls Royce has started a number of research projects including ones to derive a new scheduling and timing analysis strategy. There are a number of major challenges to tackle including:

1. Where do the values for the high-criticality WCET and low-criticality WCET estimates come from?
2. If low-criticality services can be dropped or degraded for a period of time, then how regularly and for how long?
3. How to generate the test vectors to support timing analysis?
4. How to create an equivalent No Less Safe Than Before argument?
5. How to move to preemptive, as apposed to non-preemptive, scheduling?
6. How to implement a predictable scheduler with minimal overheads complemented by appropriate timing analysis?
7. How to allocate tasks and assign task attributes so that the timing requirements are met?

6 Key Success Factors and Roadblocks

Below, we list some of the key success factors in transferring real-time systems research into industrial practice. First we consider the experience of developing the three start-up companies discussed above. With the benefit of hindsight, these were the main factors in ensuring that the companies succeeded, growing from less than 5 employees to more than 20, and culminating in successful trade sales.

[6] See the Keynote presentation at WMC (RTSS) in 2017 – `https://github.com/CPS-research-group/WMC2017/raw/master/keynote.pdf`

1. *Having an idea and then a product that made a step change for customers, providing a return on their investment.* Each commercial product provided this step change. Volcano increased network utilisation from 30% to 80% with improved reliability, and reduced development, production and warranty costs. The reduced memory footprint and overheads of the RTA-OSEK/RTA-OS operating systems resulted in lower production costs, while the use of proven real-time policies and mechanisms as well as schedulability analysis improved reliability resulting in lower warranty costs. Finally, RapiTime provided an efficient WCET analysis process, which was portable across different platforms, providing a significant reduction in testing and optimisation effort and costs.
2. *A core team of smart and hardworking people.* The founders of each company and the first few employees worked very hard (6 days per week 12+ hours per day) over many years to ensure that the company was a success.
3. *A product that was not easy to replicate: barrier to competition.* This was important in obtaining funding and getting a foothold in the market. It was particularly evident with the RTOS since the company was subsequently bought by one of its competitors.
4. *Extremely high product quality and outstanding customer support.* When a company is small and has only been around for a year or two it needs to build an excellent reputation. Quality is absolutely essential at this time, since it is make or break in terms of winning the trust of major companies who are considering adopting the technology.
5. *A balanced team of people.* On the technical side, it was not sufficient to just have technologists and software engineers who worked in the back office. Field application engineers and support staff who could do an exceptional job at customer sites / handling customer issues were also needed. Marketing and sales staff who actually understood the technology and could therefore talk effectively to both engineers and managers at customer sites were essential.
6. *Previous experience.* Having someone on board who has previous experience in a successful start-up company in the same field can be hugely advantageous, as they will understand what is needed to grow a company successfully and help avoid all manner of pitfalls.
7. *Attracting an acquisition.* An acquisition can lead to scaling up of the success of the technical transfer. In all cases the speed of adoption accelerated after acquisition. Therefore structuring the company not only for standalone success, but also acquisition was a common success factor.

There were also a number of major roadblocks and difficulties in turning promising research results into commercial reality.

1. *Funding the initial development from academic ideas and prototypes to saleable product.* A high quality industry ready product is very different from academic prototypes. It needs to be robust, with full error handling; easy to use, (since users will typically not be experts) and supported by full documentation including internal documents, e.g. requirement and test specifications, as well as external documentation such as user guides, tutorials, and marketing material. It also needs to be of extremely high quality; fully tested against its specification, and as far as possible the code needs to be bug free. The difficulty arises because considerable effort is needed in this area when the company first starts and has few sales. This effort has to be funded somehow. Self-funding by the founders can be effective if they can afford not to be paid for a while, or they can get one or two early contracts from a benevolent customer. Business angel or venture capital funding is also effective but comes at a cost of giving up some proportion of the equity (shares) in the company. Assistance from the host University or institution in terms of providing time to cover initial development efforts is also greatly beneficial at this stage.

2. *Adapting academic research to cater for industrial realities.* It is rarely if ever the case that academic theories and prototypes cover all of the details that need to be catered for in real industrial systems. There are inevitably different behaviours, aspects that are left out of models, and extra functionality that is required in commercial tools. The analysis of CAN used in the Volcano technology came close to a direct transfer. Even so, it required that the Volcano Target Package was carefully designed and developed to meet the assumptions of the theory, which was itself extended to account for specific implementation behaviours (e.g. polling input and output). Substantial engineering work was also needed to support key commercial requirements, such as the ability to re-configure signal packing and message IDs post-production. Each of the start-ups described in this paper undertook substantial engineering efforts as well as further adaptation and extension to academic results to produce commercially viable products. Again the difficulty arises because much of this effort is needed at a time when the company first starts and may have little funding and few staff.
3. *Finding the right sales staff.* In each of the three start-up companies discussed in the previous sections, it proved remarkably difficult to find people who were both good at sales and really understood the technology. In each company, sales were led by someone with a strong technical background who had the right personality and turned themselves into an excellent salesman via appropriate training. Bringing in "high flying" sales staff without a strong technology background was an expensive mistake. Beware that sales staff can be very good at selling themselves!
4. *Convincing major companies to adopt a new technology.* This is problematic due to the conservative approach often taken to purchasing from small companies. Major companies rightly have the following concerns: (i) Will the start-up be around in a year's time? (ii) Can it handle the volume of support that may be needed? (iii) Is the product really of a high enough quality to rely upon for future production? The main factors in addressing these questions were product and customer service quality, and simply time; it becomes easier to make larger sales once a company has been established for a few years.

For the final case study, where the technology transfer was into the company that directly benefited from its adoption, the existence of a long term link between the company and the research group was crucial. Simple things such as developing a common vocabulary, terms, and concepts take time. Also important are champions in both the industrial partner and the academic group. The simplistic notion that 'industry has the problem, and academia the solution' is far from true. Academics have a crucial role in understanding the problem, and experienced engineers are essential in shaping the solution. For example, moving to fixed priority scheduling meant that engineers now had greater flexibility in setting the timing requirements of the system, e.g. not just being restricted to a harmonic of the minor cycle rate. This raised the question of what the real timing requirements were. As part of the technology transfer the academics worked with engineers from Rolls-Royce across multiple disciplines, (e.g. software, hardware and control systems) to establish what the limits of the timing requirements were. This allowed significant extra benefits to be gained. Within the company there must be pull, and of course within academia, push. Research benefits from extensive use of abstraction to get to the core of the issues being addressed; however, to deploy this research the *devil is in the detail*, which takes both time and commitment.

Once a new technology, analysis method or design approach is adopted then it must be transferred completely. The academic cannot be part of the day-to-day application of the new ideas. A successful partnership has periods of deep interaction and periods of separation. New challenges may lead to a rekindled partnership.

7 Conclusions

In this paper, we described how real-time systems research has been successfully transferred into industrial practice via three start-up companies, and by direct application. Each of these impact case studies represents a success story for the real-time community. Each start-up company has developed commercially viable products, which are in use today by many of the world's leading automotive and aerospace companies and their tier 1 suppliers.

Volcano technology is used for in-vehicle communication between Electronic Control Units (ECUs) in millions of cars produced by Volvo since 1998, as well as in vehicles from a number of other major automotive manufacturers in Europe, America and Asia. The RTA-OS and RTA-OSEK real-time operating systems are used by the overwhelming majority of the world's car makers; with the number of deployed units exceeding 1.25 billion in 2017, and continuing to increase by 50-100 million per year (i.e. 1-2 million per week!). RapiTime is in use on a wide range of aerospace projects where customers need to understand the detailed execution time behaviour of their systems. The company, Rapita Systems, has recently undergone a successful trade sale (April 2016) and continues to employ a large number of graduate and post-graduate staff with expertise in real-time systems gained in the Real-Time System Research Group at the University of York. Rolls Royces' use of Visual FPS continues and has demonstrated that scheduling ideas from the research community can be exploited in the most safety-critical application domain.

Other real-time systems research groups have also succeeded in transferring their research into commercial products via start-up companies, examples include: Symptavision Gmbh (acquired by Luxoft in 2016) and Absint Gmbh, while others are just beginning.

It takes some excellent research and ideas, a willingness to take a risk and start a company or commit to a long term relationship with an industrial partner, a great deal of hard work and persistence, and perhaps an element of luck to succeed in transferring research into world-class commercial products and systems. We hope that these impact case studies will inspire others in the community to take this entrepreneurial step.

References

1 Neil C. Audsley, Iain J. Bate, and Alan Burns. Putting fixed priority scheduling into engineering practice for safety critical applications. In *Real-Time Technology and Applications Symposium (RTAS)*, pages 2–10, 1996.

2 Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

3 Neil C. Audsley, Alan Burns, and Andy J. Wellings. Deadline monotonic scheduling: Theory and application. *Control Engineering Practice*, 1(1):71–78, 1993.

4 Neil C. Audsley, Ken Tindell, and Alan Burns. The end of the line for static cyclic scheduling? In *5th Euromicro Workshop on Real-Time Systems*, pages 36–41, 1993.

5 AUTOSAR. Specification of operating system v3.1.1. Technical report, AUTOSAR, 2009. URL: `https://www.autosar.org/fileadmin/user_upload/standards/classic/3-0/AUTOSAR_SWS_OS.pdf`.

6 AUTOSAR. Premium partners. `https://www.autosar.org/about/current-partners/premium-partners/`, 2018. Accessed: 2018-02-27.

7 Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

8 John Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

9 Iain J. Bate. *Scheduling and Timing Analysis for Safety-Critical Systems*. PhD thesis, Department of Computer Science, University of York, 1998.

10 Iain J. Bate and Alan Burns. Flexible scheduling for engine controllers – uk patent application number 9710522.5 and us patent number 6,151,538, 1997.

11 Iain J. Bate and Alan Burns. Timing analysis of fixed priority real-time systems with offsets. In *9th Euromicro Workshop on Real-Time Systems*, pages 153–160, 1997.

12 Iain J. Bate and Alan Burns. An approach to task attribute assignment for uniprocessor systems. In *11th Euromicro Conference on Real-Time Systems*, pages 46–53, 1999.

13 Iain J. Bate and Alan Burns. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Systems Journal*, 25(1):5–37, 2003.

14 Iain J. Bate, Alan Burns, John A. McDermid, and Andrew J. Vickers. Towards a fixed priority scheduler for an aircraft application. In *8th Euromicro Workshop on Real-Time Systems*, pages 34–39, 1996.

15 I.J. Bate, A. Burns, T.O. Jackson, T.P. Kelly, W. Lam, P. Tongue, J.A. McDermid, A.L. Powell, J.E. Smith, A. J. Vickers, A. J. Wellings, and B.R. Whittle. Technology transfer: An integrated 'culture-friendly' approach. In *Briefing Document Technology Transfer Workshop*, 1996.

16 Guillem Bernat, Alan Burns, and Martin Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Computing*, 1(2):179–194, 2005.

17 Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems. In *3rd International Workshop on Worst-Case Execution Time Analysis*, pages 21–38, 2003.

18 Guillem Bernat, Antoine Colin, and Steffan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium*, pages 279–288, 2002.

19 Guillem Bernat, Robert I. Davis, Nicholas Merriam, John Tuffen, A. Gardner, Michael Bennett, and D. Armstrong. Identifying opportunities for worst-case execution time reduction in an avionics system. *Ada User Journal*, 28(3):189–194, 9 2007.

20 Pam Binns. A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach. In *20th Digital Avionics Systems Conference (DASC)*, volume 1, pages 1B6/1–1B6/12 vol.1, Oct 2001. `doi:10.1109/DASC.2001.963309`.

21 Bosch. Can specification version 2.0. Technical report, Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart, 1991.

22 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Computer Surveys*, 50(6):1–37, 2017.

23 Alan Burns, N. Hayes, and M.F. Richardson. Generating feasible cyclic schedules. *Control Engineering Practice*, 3(2):151–162, 1995.

24 Alan Burns and Andy J. Wellings. Engineering a hard real-time system: From theory to practice. *Softw., Pract. Exper.*, 25(7):705–726, 1995. `doi:10.1002/spe.4380250702`.

25 Lennart Casparsson, Antal Rajnak, Ken Tindell, and Peter Malmberg. Volcano - a revolution in on-board communications. Technical report, Volvo, 1998.

26 Certification Authorities Software Team CAST. Addressing cache in airborne systems and equipment – cast-20, June 2003.

27 Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *14th Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

28 Antoine Colin and Stefan M. Petters. Experimental evaluation of code properties for WCET analysis. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 190–199, 2003.

29 Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an RTOS can stay within on-chip memory limits. In *Work in Progress and Industrial Experience Sessions, 12th EuroMicro Conference on Real-Time Systems.*, 2000.

30 Robert I. Davis. Impact case study: Guaranteeing the real-time performance of in-vehicle networks. Technical report, University of York, 2015. URL: `https://www-users.cs.york.ac.uk/~robdavis/papers/ImpactCaseStudyVolcano.pdf`.

31 Robert I. Davis, Guillem Bernat, Ian Broster, and Antoine Colin. Impact case study: How long does your real-time software take to run? Technical report, University of York, 2015. URL: `https://www-users.cs.york.ac.uk/~robdavis/papers/ImpactCaseStudyRapiTime.pdf`.

32 Robert I. Davis and Alan Burns. Robust priority assignment for messages on controller area network (CAN). *Real-Time Systems*, 41(2):152–180, 2009.

33 Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

34 Robert I. Davis, Alan Burns, Victor Pollex, and Frank Slomka. On priority assignment for controller area network when some message identifiers are fixed. In *23rd International Conference on Real Time Networks and Systems, RTNS*, pages 279–288, 2015.

35 Robert I. Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. Controller area network (CAN) schedulability analysis with FIFO queues. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 45–56, 2011.

36 Robert I. Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. Schedulability analysis for controller area network (CAN) with FIFO queues priority queues and gateways. *Real-Time Systems*, 49(1):73–116, 2013.

37 Robert I. Davis and Nicolas Navet. Controller area network (CAN) schedulability analysis for messages with arbitrary deadlines in FIFO and work-conserving queues. In *9th IEEE International Workshop on Factory Communication Systems, (WFCS)*, pages 33–42, 2012.

38 Robert I. Davis and Nigel Tracey. Impact case study: The world's smallest automotive real-time operating system. Technical report, University of York, 2015. URL: `https://www-users.cs.york.ac.uk/~robdavis/papers/ImpactCaseStudyRTOS.pdf`.

39 Paul Emberson and Iain J. Bate. Minimising task migrations and priority changes in mode transitions. In *13th IEEE Real-Time And Embedded Technology and Applications Symposium*, pages 158–167, 2007.

40 Paul Emberson and Iain J. Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *IEEE Transactions on Software Engineering*, 36(5):704–718, 2010.

41 ETAS. RTA-OS RH850/WR Port Guide. `https://www.etas.com/download-center-files/products_RTA_Software_Products/RTA-OS_RH850WR_Port_Guide_V2.0.5.pdf`, 2017. Accessed: 2018-02-27.

42 ETAS. RTA software products. `https://www.etas.com/en/products/rta_software_products.php`, 2018. Accessed: 2018-02-27.

43 Mentor Graphics. Volcano in-vehicle embedded software. `http://www.mentor.com/products/vnd/in-vehicle_software/`. Accessed: 2018-02-27.

44 Mentor Graphics. Volcano Network Architect (vna). `http://www.mentor.com/products/vnd/communication-management/vna/`. Accessed: 2018-02-27.

45 Mentor Graphics. Mentor graphics strengthens its automotive solutions portfolio with the acquisition of volcano communications technologies. `https://www.mentor.com/company/news/volcano_acquisition`, 2005. Accessed: 2018-02-27.

46 Mentor Graphics. Shanghai automotive industries adopts mentor graphics volcano automotive network design tools. `http://www.mentor.com/products/vnd/news/saic_sdopts_volcano`, 2006. Accessed: 2018-02-27.

47 Mentor Graphics. Volcano target package datasheet. `http://www.mentor.com/products/vnd/communication-management/vna/upload/VNA_Datasheet.pdf`, 2006. Accessed: 2018-02-27.

48 Mentor Graphics. Volcano target package datasheet. `http://www.mentor.com/products/vnd/in-vehicle_software/volcano_target_package/upload/vtp-ds.pdf`, 2010. Accessed: 2018-02-27.

49 Mentor Graphics. Volcano network architect from mentor graphics verifies and improves network bandwidth usage at mazda. `http://www.mentor.com/products/vnd/news/mentor-vnd-mazda`, 2012. Accessed: 2018-02-27.

50 Paul Graydon and Iain J. Bate. Realistic safety cases for the timing of systems. *The Computer Journal*, 57(5):759–774, 2014.

51 S. Hutchesson and N. Hayes. Technology transfer and certification issues in safety critical real-time systems. In *Digest of the IEE Colloquium on Real-Time Systems*, page 98/306, 1998.

52 ISO. ISO 17356-3:2005 preview road vehicles – open interface for embedded automotive applications – part 3: Osek/vdx operating system (os). Technical report, ISO, 2005. URL: `https://www.iso.org/standard/40079.html`.

53 Tim Kelly, Iain J. Bate, John McDermid, and Alan Burns. Building a preliminary safety case: An example from aerospace. In *Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, 1997.

54 Dawood Ashraf Khan, Robert I. Davis, and Nicolas Navet. Schedulability analysis of CAN with non-abortable transmission requests. In *16th IEEE Conference on Emerging Technologies & Factory Automation, (ETFA)*, pages 1–8, 2011.

55 Motorola. MSCAN block guide. Technical report, Motorola, , Document No. S12MSCANV2/D., 2004. URL: `http://application-notes.digchip.com/314/314-67565.pdf/`.

56 Antal Rajnak. Volcano technology: Enabling correctness by design. In *The Industrial Communication Technology Handbook*, chapter 32. CRC Press, 2009.

57 RTCA-EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification DO-178B/ED-12B*. RTCA, Inc, December 1992.

58 Rapita Systems. Capturing worst case timing and stack usage data for do-178b level a embraer flight control systems. `https://www.rapitasystems.com/downloads/do-178b-level-embraer-fcs`. Accessed: 2018-02-27.

59 Rapita Systems. Verifying the timing correctness of infineon's safetcore safety drivers. `https://www.rapitasystems.com/downloads/infineon-safetcore-drivers`. Accessed: 2018-02-27.

60 Rapita Systems. Rapitime worst-case execution time optimization on the bae systems hawk mission computer. `https://www.rapitasystems.com/downloads/bae-systems-hawk-mission-computer`, 2006. Accessed: 2018-02-27.

61 Rapita Systems. Flight control system execution timing analyzed cheaper, faster with rapitime. `https://www.rapitasystems.com/downloads/wide-body-jet-flight-control-system`, 2009. Accessed: 2018-02-27.

62 Rapita Systems. Proving and Improving Worst-Case Execution Times on the Alenia Aermacchi M-346. `https://www.rapitasystems.com/downloads/alenia-aermacchi-m-346`, 2010. Accessed: 2018-02-27.

63 Rapita Systems. Qualification of RapiCover for MC/DC coverage of DO-178B level-A software. `https://www.rapitasystems.com/system/files/downloads/mc-cs-009_alenia_aermacchi_m346_case_study_v2.pdf`, 2010. Accessed: 2018-03-26.

64 Rapita Systems. Danlaw acquires rapita systems. `https://www.rapitasystems.com/news/danlaw-acquires-rapita-systems`, 2016. Accessed: 2018-02-27.

65 Ken Tindell and Alan Burns. Guaranteeing message latencies on controller area network (CAN). In *1st international CAN conference*, pages 1–11, 1994.

66 Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

67 Ken Tindell, Alan Burns, and Andy J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, 1995.

68 Ken Tindell, Alan Burns, and Andy J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

69 Ken Tindell, H. Hanssmon, and Andy J. Wellings. Analysing real-time communications: Controller area network (CAN). In *15th IEEE Real-Time Systems Symposium*, pages 259–263, 1994.

70 Volvo. Volvo annual reports. `https://group.volvocars.com/sustainability/publication-list`, 2018. Accessed: 2018-02-27.

71 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem; overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. `doi:10.1145/1347375.1347389`.

72 Yaba. Rapita systems flies high. `http://www.rapitasystems.com/system/files/yabawinter05news.2.pdf`, 2005. Accessed: 2018-02-27.

73 Patrick Meumeu Yomsi, Dominique Bertrand, Nicolas Navet, and Robert I. Davis. Controller area network (CAN): response time analysis with offsets. In *9th IEEE International Workshop on Factory Communication Systems, (WFCS)*, pages 43–52, 2012.

Push Forward: Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems

Jian-Jia Chen
TU Dortmund University, Germany
jian-jian.chen@tu-dortmund.de
https://orcid.org/0000-0001-8114-9760

Georg von der Brüggen
TU Dortmund University, Germany
georg.von-der-brueggen@tu-dortmund.de
https://orcid.org/0000-0002-8137-3612

Niklas Ueter
TU Dortmund University, Germany
niklas.ueter@tu-dortmund.de
https://orcid.org/0000-0002-6722-4805

Abstract

The sporadic task model is often used to analyze recurrent execution of tasks in real-time systems. A sporadic task defines an infinite sequence of task instances, also called jobs, that arrive under the minimum inter-arrival time constraint. To ensure the system safety, timeliness has to be guaranteed in addition to functional correctness, i.e., all jobs of all tasks have to be finished before the job deadlines. We focus on analyzing arbitrary-deadline task sets on a homogeneous (identical) multiprocessor system under any given global fixed-priority scheduling approach and provide a series of schedulability tests with different tradeoffs between their time complexity and their accuracy. Under the arbitrary-deadline setting, the relative deadline of a task can be longer than the minimum inter-arrival time of the jobs of the task. We show that global deadline-monotonic (DM) scheduling has a speedup bound of $3 - 1/M$ against any optimal scheduling algorithms, where M is the number of identical processors, and prove that this bound is asymptotically tight.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases global fixed-priority scheduling, schedulability analyses, speedup bounds

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.8

Related Version A full version of this paper is available at [21], `http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2018-chen-ecrts-push-forward-full.pdf`.

Supplement Material ECRTS Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.4.2.6`

Funding This paper is supported by DFG, as part of the Collaborative Research Center SFB876 (`http://sfb876.tu-dortmund.de/`), project B2.

1 Introduction

The sporadic task model is the basic task model in real-time systems, where each task τ_i releases an infinite number of *task instances* (*jobs*) under its *minimum inter-arrival time* (*period*) T_i and is further characterized by its *relative deadline* D_i and its *worst-case execution time* C_i. The sporadic task model has been widely adopted in real-time systems. A sporadic task defines an infinite sequence of task instances, also called *jobs*, that arrive under the minimum inter-arrival time constraint, i.e., any two consecutive releases of jobs of task τ_i are temporally separated by at least T_i. When a job of task τ_i arrives at time t, it must finish no later than its *absolute deadline* $t + D_i$. If all tasks release their jobs strictly periodically with period T_i, the task model is the well-known Liu and Layland task model [33]. A sporadic task set is called with 1) *implicit deadlines*, if the relative deadlines are equal to their minimum inter-arrival times, 2) *constrained deadlines*, if the minimum inter-arrival times are no less than their relative deadlines, and 3) *arbitrary deadlines*, otherwise.

To schedule such task sets on a multiprocessor platform, three paradigms have been widely adopted: partitioned, global, and semi-partitioned multiprocessor scheduling. The *partitioned* scheduling approach partitions the tasks statically among the available processors, i.e., a task executes all its jobs on the assigned processor. The *global* scheduling approach allows a job to migrate from one processor to another at any time. The *semi-partitioned* scheduling approach decides whether a task is divided into subtasks statically and how each task/subtask is then assigned to a processor. A comprehensive survey of multiprocessor scheduling for real-time systems can be found in [23].

We focus on *global fixed-priority preemptive scheduling* on M identical processors, i.e., unique fixed priority levels are statically assigned to the tasks and at any point in time the M highest-priority jobs in the ready queue are executed. Hence, the schedule is *workload-conserving.* The response time of a job is defined as its finish time minus its arrival time. The worst-case response time of a task is an upper bound on the response times of all the jobs of the task and can be derived by a *(worst-case) response time analysis* for a sporadic task under a given scheduling algorithm. Verifying whether a set of sporadic tasks can meet their deadlines by a scheduling algorithm is called a *schedulability test*, i.e., verifying if the *(worst-case) response time* is smaller than or equal to the *relative deadline.*

1.1 Related Work

For uniprocessor systems, i.e, M=1, the exact schedulability test and the (tight) worst-case response time analysis by using *busy intervals* were provided by Lehoczky [32]. Several approaches have been proposed to reduce the time complexity, e.g., [35]. Bini and Buttazzo [12] proposed a framework of schedulability tests that can be tuned to balance the time complexity and the acceptance ratio of the schedulability test for uniprocessor sporadic task systems. To achieve polynomial-time schedulability tests and response time analyses, Lehoczky [32] proposed a utilization upper bound for a set of sporadic arbitrary-deadline tasks under fixed-priority scheduling. The linear-time response-time bound for fixed-priority systems was first proposed by Davis and Burns [22], and later improved by Bini et al. [14, 15] and Chen et al. [18]. The computational complexity of the schedulability test problem and the worst-case response time analysis in uniprocessor systems for different variances can be found in [16, 25, 24, 27, 26].

In this paper, we will implicitly assume multiprocessor systems, i.e., $M \geq 2$. Many results are known for constrained-deadline ($D_i \leq T_i$) and implicit-deadline task systems ($D_i = T_i$) on identical multiprocessor platforms, e.g., [2, 5, 30, 1, 7, 18]. For details, please refer to the survey by Davis and Burns [23]. Unfortunately, deriving exact schedulability tests under multiprocessor global scheduling is much harder than deriving them for uniprocessor systems due to the lack of concrete worst-case scenarios that can be constructed efficiently. Most results in the literature focus on sufficient schedulability tests. Exceptions are the exhaustive search under discrete time parameters by Baker and Cirinei [4], finite automata under discrete time parameters by Geeraerts et al. [29], and hybrid finite automata by Sun and Lipari [36]. Specifically, Geeraerts et al. [29] showed that the schedulability test formulation by Baker and Cirinei [4] is PSPACE-Complete.

Regarding global fixed-priority scheduling for arbitrary-deadline task systems, several sufficient schedulability tests and safe worst-case response time analyses have been proposed, e.g., [3, 4, 8, 9, 30, 37, 31]. Baker [3] designed a test based on certain properties to characterize a *problem window*. Baruah and Fisher [8, 9] used different annotations to extend the analysis window and derived corresponding exponential-time schedulability tests. The first worst-case response-time analysis for arbitrary-deadline task systems was proposed by Guan et al. [30], where the authors used the insight proposed by Baruah [5] to limit the number of carry-in jobs, and then apply the workload function proposed by Bertogna et al. [11] to quantify the requested demand of higher-priority tasks. Unfortunately, it has recently been shown by Sun et al. [37] that this analysis in [30] is optimistic. In addition, Sun et al. [37] derived a complex carry-in workload function for the response time analysis where all possible combinations of carry-in and non-carry-in functions have to be explicitly enumerated. However, their method is computationally intractable since the time complexity is exponential. Huang and Chen [31] proposed a more precise quantification for the number of carry-in jobs of a task than the bounds used in the tests provided in [3, 9]. They also presented a response time bound for arbitrary-deadline tasks under global scheduling in multiprocessor systems with linear-time complexity.

1.2 Our Contribution

We consider arbitrary-deadline sporadic task systems, which is the most general case of the sporadic real-time task model. To quantify the performance loss due to efficient schedulability tests and the non-optimality of scheduling algorithms, we will adopt the notion of speedup factors/bounds, also known as resource augmentation factors/bounds. Table 1 summarizes the state-of-the-art speedup bounds for the global deadline-monotonic (DM) scheduling, one specific global fixed-priority scheduling algorithm. Under global DM, a task τ_i has higher priority than task τ_j if $D_i \leq D_j$, in which ties are broken arbitrarily. The authors note that the proof by Lundberg [34] seems incomplete. However, the concrete task set in [34] provides the lower bound 2.668 of the speedup factors for global DM. Moreover, Andersson [1] showed that global slack monotonic scheduling has a speedup bound of $\frac{3+\sqrt{5}}{2} \approx 2.6181$ for implicit-deadline task systems. However, no better global fixed-priority scheduling algorithms with respect to speedup factors are known for constrained-deadline and arbitrary-deadline task systems.

Our Contributions. Table 1 summarizes the related results and the contribution of this paper for multiprocessor global fixed-priority preemptive scheduling. We improve the best known results by Baruah and Fisher [8] with respect to the speedup bounds. Our contributions are:

- For *any* global fixed-priority preemptive scheduling, we provide a series of schedulability tests with different tradeoffs between time complexity and accuracy in Section 3 and Section 4.

Table 1 Speedup bounds of the global deadline-monotonic (DM) scheduling algorithm for sporadic task systems.

		implicit deadlines	constrained deadlines	arbitrary deadlines
Global DM	upper bounds	2.668 [34] (poly.-time)	$3 - 1/M$ [7] (expo.-time)	$\frac{2(M-1)}{4M-1-\sqrt{12M^2-8M+1}} \leq 3.73$ [8] (expo.-time)
		2.823 [18] (poly.-time)	$3 - 1/M$ [18] (poly.-time)	$3 - \frac{1}{M}$ (*this paper*) (poly.-time)
	lower bounds	2.668 [34]	2.668 [34]	2.668 [34]
				$3 - \frac{3}{M+1}$ (this paper)

- We show that the global deadline-monotonic scheduling algorithm has a speedup factor $3 - 1/M$ with respect to the optimal multiprocessor scheduling policies when considering task systems with arbitrary deadlines. This improves the analyses by Fisher and Baruah with respect to the speedup bounds, i.e., $4 - 1/M$ [9] and 3.73 [8].
- We show that all the schedulability tests we provide in this paper analytically dominate the tests by Baruah and Fisher [8] for global DM. We also show that global DM has a speedup lower bound of $3 - 3/(M+1)$, which shows that our schedulability analyses are asymptotically tight with respect to the speedup factors.

2 System Model, Definitions, and Assumptions

We consider an arbitrary-deadline sporadic task set $\mathbf{T}$ with N tasks executed on $M \geq 2$ identical processors based on global fixed-priority preemptive scheduling. We assume that the priority levels of the tasks are unique (and given) and that τ_i has higher priority than task τ_j if $i < j$. When there is only one processor, i.e., $M = 1$, the existing results discussed in Section 1.1 can be adopted, and our analysis here cannot be applied. We will implicitly use the assumption $M \geq 2$ in the paper.

By definition, M is an integer. In addition to C_i, T_i, D_i, we also define the utilization U_i task τ_i as C_i/T_i. We will implicitly assume that $D_i > 0$, $C_i > 0$, $T_i > 0$, $C_i/D_i \leq 1$, and $U_i \leq 1$ $\forall \tau_i$ in this paper. Moreover, *intra-task parallelism* is not allowed. *At most one job* of task τ_i can be executed on at most one processor at each instant in time, *regardless of the number of the jobs of task* τ_i *awaiting for execution and the number of idle processors.* We denote the set of natural numbers as $\mathbb{N}$.

2.1 Resource Augmentation

We assume the original platform speed is 1. Therefore, running the platform at speed s implies that the worst-case execution time of task τ_i becomes C_i/s. A scheduling algorithm $\mathcal{A}$ has a *speedup bound* s with respect to the optimal schedule, if it guarantees to always produce a feasible solution when 1) each processor is sped up to run at s times of the original speed of the platform and 2) the task set $\mathbf{T}$ can be feasibly scheduled on the original M identical processors, i.e., running at speed 1.

We will use the negation of the above definition to quantify the failure of algorithm $\mathcal{A}$: *If* $\mathcal{A}$ *fails to ensure that all the tasks in* $\boldsymbol{T}$ *meet their deadlines, then no feasible multiprocessor schedule exists when each processor is slowed down to run at speed* $1/s$*.*

2.2 Definitions and Necessary Condition

We define the following notation according to the task system and the priority assignment:

- density δ_i of task τ_i: $\delta_i = C_i/\min\{D_i, T_i\}$
- maximum density $\delta_{\max}(k)$ among the first k tasks: $\delta_{\max}(k) = \max_{i=1}^{k} \delta_i$

- maximum between the utilization of the higher-priority tasks and the density of task τ_k: $U^{\max}_{\delta,k} = \max\{\max_{i=1}^{k-1} U_i, \delta_k\}$
- demand bound function [10] $\text{DBF}(\tau_i, t)$ of task τ_i, further explained in Definition. 2.1
- load $\text{LOAD}(k)$ of the first k tasks: $\text{LOAD}(k) = \max_{t>0} \frac{\sum_{i=1}^{k} \text{DBF}(\tau_i,t)}{t}$

▶ **Definition 2.1** (demand bound function (DBF) by Baruah [10])**.** For any $t \geq 0$

$$\text{DBF}(\tau_i, t) = \max\left\{0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right\} \tag{1}$$

The demand bound function $\text{DBF}(\tau_i, t)$ defines the execution time task τ_i must finish for any interval length t to ensure its timing correctness.

Since $\delta_i \geq U_i$ by definition, we know that $U^{\max}_{\delta,k} \leq \delta_{\max}(k)$. As we assume $C_i/D_i \leq 1$ and $U_i \leq 1$ we know that $\delta_i \leq 1$. In addition to DBFs, we will heavily use the following workload function:

▶ **Definition 2.2** (Workload function)**.** Let $work_i(t)$ be a workload function, representing the maximum amount of time for *sequentially* executing the jobs of task τ_i released in time interval $[a, a+t)$, i.e., jobs released before a are not considered. For any $t \geq 0$

$$work_i(t) = \left\lfloor \frac{t}{T_i} \right\rfloor C_i + \min\left\{C_i, t - \left\lfloor \frac{t}{T_i} \right\rfloor T_i\right\}. \tag{2}$$

For notational brevity, we set $work_i(t)$ to $-\infty$ if $t < 0$.

The workload function $work_i(t)$ defined above is a piecewise function, i.e., linear in intervals $[\ell T_i, \ell T_i + C_i]$ with a slope 1 and constant, $(\ell+1)C_i$, in intervals $[\ell T_i + C_i, (\ell+1)T_i]$ for any non-negative integer ℓ. Two examples of the workload function are illustrated in Figure 2 in Section 3. To prove the speedup bound, we will utilize the following necessary condition.

▶ **Lemma 2.3.** *A task set $\boldsymbol{T}$ with N tasks is not schedulable by any multiprocessor scheduling algorithm when the M processors are running at any speed s, if*

$$\max\left\{\max_{t>0} \frac{\sum_{\tau_i \in \boldsymbol{T}} \text{DBF}(\tau_i, t)}{Mt}, \frac{\sum_{\tau_i \in \boldsymbol{T}} U_i}{M}, \delta_{\max}(N)\right\} > s. \tag{3}$$

Proof. This is widely used based on a reformulation in the literature, e.g., [8, 9]. ◀

2.3 Analysis Based on DBFs

Baruah and Fisher in [8] provided a schedulability test for task τ_k under global deadline-monotonic (DM) scheduling that is based on the Demand Bound Functions (DBF), assuming that the tasks are sorted according to DM order already, i.e., $D_1 \leq D_2 \leq \ldots \leq D_N$:

▶ **Theorem 2.4** (Baruah and Fisher [8], revised in [17])**.** *Let μ_k be defined as $M - (M-1)\delta_{\max}(k)$. Task τ_k is schedulable under global DM if* [1]

$$2\text{LOAD}(k) + (\lceil \mu_k \rceil - 1)\delta_{\max}(k) \leq \mu_k. \tag{4}$$

[1] The original proof by Baruah and Fisher [8] had a mathematical flaw in their Lemma 3, i.e., setting μ_k to $M - (M-1)\delta_k$. It can be fixed by setting μ_k to $M - (M-1)\delta_{\max}(k)$.

3 Schedulability Test by Pushing Forward

In this section, we provide several conditions for the schedulability of task τ_k under a given preemptive global fixed-priority scheduling algorithm. They lead to a sufficient schedulability test for τ_k, assuming that the schedulability of the tasks $\tau_1, \tau_2, \ldots, \tau_{k-1}$ under the given algorithm is already verified. This means that for all tasks τ_i with $i < k$ the worst-case response time is at most D_i. Therefore, the test should be applied for all tasks, i.e., from the highest-priority task to the lowest-priority task, to ensure the schedulability of the task set under the (specified/given) global fixed-priority scheduling. As the test presented here has a high time complexity, we provide more efficient tests in Section 4.

3.1 Analysis Window Extension

We analyze the schedulability of τ_k by looking at the intervals where τ_k is active in the schedule S provided by the global fixed-priority scheduling algorithm according to the following definition:

▶ **Definition 3.1** (active task). For a schedule S, a task τ_i is active at time t, if there is (at least) one job of τ_i that has arrived before or at t and has not finished yet at time t.

The schedulability conditions are proved by using *contrapositive*. Suppose a schedule S produced by the given global fixed-priority scheduling algorithm and that t_d is the earliest (absolute) deadline at which a job of task τ_k misses its deadline. Let t_a be the time instant in S such that τ_k is continuously active in the time interval $[t_a, t_d)$ and is not active *immediately* prior to t_a. By definition, t_a must be the arrival time of a job of task τ_k. Suppose that t_d is the absolute deadline of the ℓ-th job of task τ_k that arrived in the time interval $[t_a, t_d)$. Therefore, as τ_k is a sporadic task, $t_d - t_a \geq (\ell - 1)T_k + D_k$. For notational brevity, we define $D'_k = (\ell - 1)T_k + D_k$ and $C'_k = \ell C_k$.

We remove all the jobs of task τ_k that arrive before t_a and all the jobs with priorities lower than τ_k from the schedule S. The schedule of task τ_k remains unchanged in the resulting (new) schedule S, due to the preemptiveness of the global fixed-priority scheduling algorithm. Let C^*_k be the amount of time that task τ_k is executed from t_a to t_d. Since the ℓ-th job of task τ_k misses its deadline, we know that $C^*_k < \ell C_k = C'_k$. We now introduce three functions that are defined for any $t \leq t_d$.

- Let $E(t, t_d)$ be the amount of workload (sum of the execution times) of the higher-priority jobs, i.e., from $\tau_1, \tau_2, \ldots, \tau_{k-1}$, *executed* in the time interval $[t, t_d)$ in schedule S.
- Let $W(t, t_d)$ be $C^*_k + E(t, t_d)$.
- Let $\Omega(t, t_d)$ be $\frac{W(t,t_d)}{t_d - t}$.

Those definitions and the deadline miss of task τ_k at time t_d lead to the following lemma.

▶ **Lemma 3.2.** *Since τ_k misses its deadline at t_d in S, the following conditions hold:*

$$E(t_a, t_d) \geq M \times (t_d - t_a - C^*_k) \tag{5}$$

$$W(t_a, t_d) > M \times (t_d - t_a) - (M - 1)C'_k \tag{6}$$

$$\Omega(t_a, t_d) > M - (M - 1) \times \frac{C'_k}{D'_k} \tag{7}$$

Proof. Since task τ_k is active from t_a to t_d and is only executed for *exactly* C^*_k amount of time, we know that all M processors must be busy executing other higher-priority jobs for at least $t_d - t_a - C^*_k$ amount of time. Therefore, the amount of workload $E(t_a, t_d)$ of the

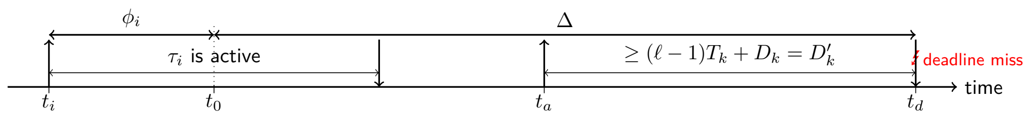


Figure 1 The notation used in Section 3: 1) task τ_k is continuously active from t_a to t_d with a deadline miss at time t_d; 2) time instant t_0 is the smallest value of $t \leq t_a$ such that $\Omega(t, t_d) \geq \mu_k$; 3) time instant t_i is the arrival time of a higher-priority carry-in task τ_i if τ_i is continuously active in time interval $[t_i, t_0 + \varepsilon]$, where $t_i < t_0$ and $\varepsilon > 0$ is an arbitrarily small number; 4) ϕ_i is $t_0 - t_i$ and Δ is $t_d - t_0$.

higher-priority jobs executed in the time interval $[t_a, t_d)$ must be at least $M \times (t_d - t_a - C^*_k)$, i.e., Eq. (5) must hold.[2] Therefore, since $W(t_a, t_d)$ is defined as $E(t_a, t_d) + C^*_k$, we have

$$W(t_a, t_d) \geq M \times (t_d - t_a - C^*_k) + C^*_k > M \times (t_d - t_a) - (M-1)C'_k,$$

where the last inequality is due to $M \geq 2$ and $C'_k > C^*_k$. This leads to the conditions in Eq. (6). Since $\Omega(t_a, t_d)$ is defined as $\frac{W(t_a,t_d)}{t_d - t_a}$ and $D'_k \leq t_d - t_a$, we have

$$\Omega(t_a, t_d) \geq M - (M-1)\frac{C'_k}{t_d - t_a} \geq M - (M-1)\frac{C'_k}{D'_k},$$

i.e., the condition in Eq. (7). ◀

Although the interval $[t_a, t_d)$ can already be used for constructing the schedulability tests, researchers have tried to push the interval of interest towards $[t_0, t_d)$ for some $t_0 \leq t_a$ based on certain properties, e.g., [31, 9, 8]. Such extensions have been shown to provide better quantifications of the interfering workload from the higher-priority tasks. In our analysis, we will use a similar extension strategy as suggested by Baruah and Fisher [8] based on a user-specified parameter ρ.

The following definition and lemmas are from [8]. Figure 1 provides an illustration of our notation based on the above definitions.

▶ **Definition 3.3.** Suppose that $\mu_k = M - (M-1)\rho$ for a certain ρ with $1 \geq \rho \geq \frac{C'_k}{D'_k}$. For the schedule S, let time instant t_0 be the smallest value of $t \leq t_a$ such that $\Omega(t, t_d) \geq \mu_k$. This means, $\Omega(t, t_d) < \mu_k$ for any $t < t_0$.

▶ **Lemma 3.4.** *If τ_k misses its deadline at t_d, for any ρ with $1 \geq \rho \geq \frac{C'_k}{D'_k}$, the time t_0, as defined in Definition 3.3, always exists with $\Omega(t_0, t_d) \geq \mu_k$ and $t_0 \leq t_a$.*

Proof. By Eq. (7) from Lemma 3.2 and $\rho \geq \frac{C'_k}{D'_k}$, we know

$$\Omega(t_a, t_d) > M - (M-1) \times \frac{C'_k}{D'_k} \geq M - (M-1)\rho = \mu_k.$$

Therefore, such a time instant $t_0 \leq t_a$ exists, at least when the system starts. ◀

▶ **Definition 3.5** (carry-in task). A task τ_i is a carry-in task in the schedule S, if τ_i is continuously active in a time interval $[t_i, t_0 + \varepsilon]$, for $t_i < t_0$ and an arbitrarily small $\varepsilon > 0$.

▶ **Lemma 3.6.** *For $1 \geq \rho \geq \frac{C'_k}{D'_k}$, there are at most $\lceil M - (M-1)\rho \rceil - 1$ carry-in tasks at t_0 in schedule S.*

[2] The condition in Eq. (5) is widely used in the form of $E(t_a, t_d) > M \times (t_d - t_a - \ell C_k)$. Here, since we will use C^*_k, the correct form is with $\geq$.

3.2 Analysis Based on Workload Functions

By extending the interval of interest to $[t_0, t_d)$, Baruah and Fisher provided the schedulability test shown in Theorem 2.4 in this paper. However, they analyzed the workload in $[t_0, t_d)$ based on the DBFs by using the function $\text{LOAD}(k)$ as an approximation, which will be shown pessimistic in Corollary 5.1 in Section 5. Moreover, their final analysis can only be applied for global DM. We will carefully analyze the workload executed in $[t_0, t_d)$ to ensure that the analytical accuracy is better preserved and that the analysis can be used for any global fixed-priority preemptive scheduling. We will demonstrate that our analysis dominates the analysis by Baruah and Fisher [8] in Corollary 5.1.

For the analysis before Theorem 3.10, we will assume that ρ is given and t_0 is already defined. According to Lemma 3.6, at time t_0 at most $\lceil M - (M-1)\rho \rceil - 1$ tasks are active in schedule S. We quantify their contribution to the *executed* workload in time interval $[t_0, t_d)$ with two different forms from Lemma 3.7, denoted by $\omega_i^{heavy}(t_d - t_0)$, and from Lemma 3.8, denoted by $\omega_i^{light}(t_d - t_0)$. While Lemma 3.7 can be used in general, Lemma 3.8 only holds if $U_i \leq \rho$. After these workload functions are detailed and explained, we will show their relationship in Lemma 3.9. Then, we will explain how they can be used and detail the constructed schedulability test in Theorem 3.10 based on the above concepts.

▶ **Lemma 3.7.** *If all jobs of a higher-priority task τ_i meet their deadlines, the upper bound $\omega_i^{heavy}(\Delta)$ on the workload of task τ_i executed from t_0 to t_d with $\Delta = t_d - t_0$ in schedule S is at most:*

$$\omega_i^{heavy}(\Delta) = work_i(\Delta + D_i). \tag{8}$$

Proof. Since all jobs of τ_i meet their deadlines, the jobs of τ_i executed in $[t_0, t_d)$ must arrive in the time interval $(t_0 - D_i, t_d)$. Therefore, the workload of task τ_i that can be sequentially executed is upper bounded by the workload function with length $t_d - (t_0 - D_i) = \Delta + D_i$. ◀

The key improvement achieved in this paper is due to the following Lemma 3.8 to safely bound the workload of a light task.

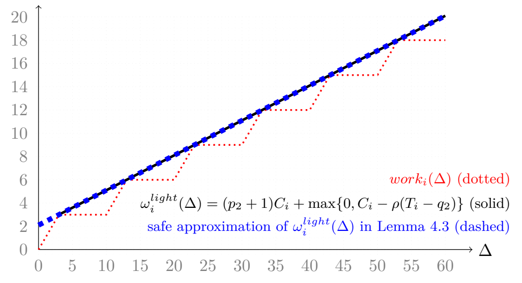
Figure 2 demonstrates the workload function for different cases in Lemma 3.8, together with a linear approximation that will be presented in Lemma 4.3. For the workload function defined in Eq. (9), informally speaking, the workload defined by $(p_2+1)C_i + \max\{0, C_i - \rho(T_i - q_2)\}$ can be imagined as if 1) there is an offset for C_i amount of execution time at beginning of the interval, and 2) the workload in each period starting from $C_i + p_2 T_i$ to $C_i + (p_2+1)T_i$ is pushed to the end of the period with a slope ρ. For example, in Figure 2(b), the offset is 3, the workload increases from 3 at time 7 to 6 at time 13 with a slope $\rho = 0.5$, the workload increases from 6 at time 17 to 9 at time 23 with a slope $\rho = 0.5$, etc.

▶ **Lemma 3.8.** *If all jobs of a higher-priority task τ_i meet their deadlines and $U_i \leq \rho \leq 1$, the upper bound $\omega_i^{light}(\Delta)$ on the workload of task τ_i executed from t_0 to t_d with $\Delta = t_d - t_0$ in schedule S is:*

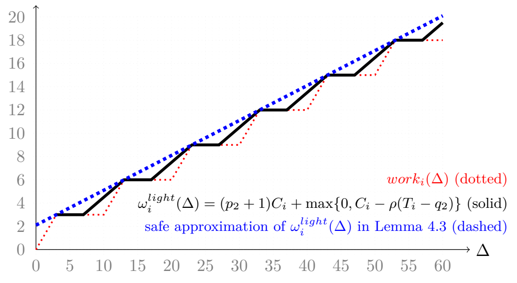
$$\omega_i^{light}(\Delta) = \begin{cases} \Delta & \text{if } 0 < \Delta \leq C_i \\ \max\begin{Bmatrix} work_i(\Delta), \\ (p_2+1)C_i + \max\{0, C_i - \rho(T_i - q_2)\} \end{Bmatrix} & \text{if } \Delta > C_i \end{cases} \tag{9}$$

where $p_2 = \lceil (\Delta - C_i)/T_i \rceil - 1$ and q_2 is $\Delta - C_i - p_2 T_i$.

Proof. As the case when $0 < \Delta \leq C_i$ is due to the definition, let $\Delta > C_i$ for the rest of the proof. Based on the schedule S, let $t_i < t_0$ be the time instant such that task τ_i is



(a) $U_i = 0.3$ and $\rho = 0.3$



(b) $U_i = 0.3$ and $\rho = 0.5$

Figure 2 Two examples for the approximation of $work_i$ for τ_i with $T_i = 10, C_i = 3, D_i = 45$: black curves for $\omega_i^{light}(\Delta)$ defined in Lemma 3.8 and the approximation in Lemma 4.3 (blue curves).

continuously active in the time interval $[t_i, t_0]$ and task τ_i is not active *immediately* prior to t_i. If t_i does not exist, then task τ_i does not have workload released before t_0 that is still active. Therefore, the worst-case workload is $work_i(\Delta)$ in this case.

Let ϕ_i be $t_0 - t_i$. By the definition of t_i, if it exists, there are at most $\left\lceil \frac{\phi_i}{T_i} \right\rceil$ jobs of task τ_i executed in time interval $(t_i, t_0]$. For the rest of the proof, we only consider that t_i exists and that $\Delta > C_i$. By definition, t_i must be the arrival time of a job of task τ_i. Moreover, due to the definition of t_0 in Definition 3.3, we know that $\Omega(t_i, t_d) < M - (M-1)\rho$. Since $\Omega(t_i, t_d) < M - (M-1)\rho$ and $\Omega(t_0, t_d) \geq M - (M-1)\rho$, we have

$$W(t_0, t_d) = \Omega(t_0, t_d) \cdot (t_d - t_0) \geq (t_d - t_0)\mu_k = \Delta\mu_k \tag{10}$$

$$W(t_i, t_d) = \Omega(t_i, t_d) \cdot (t_d - t_i) < (t_d - t_i)\mu_k = (\Delta + \phi_i)\mu_k \tag{11}$$

Substracting Eq. (11) by Eq. (10), we have $W(t_i, t_d) - W(t_0, t_d) < \phi_i\mu_k$, i.e., in schedule S the workload executed in time interval $[t_i, t_0)$ is *strictly less* than $\phi_i\mu_k$. Suppose that y_i is the amount of time that task τ_i is executed in time interval $[t_i, t_0)$, i.e., task τ_i is active but blocked by other higher-priority jobs for $\phi_i - y_i$ amount of time in this time interval. When task τ_i is blocked in global fixed-priority scheduling, all the M processors are executing other jobs. The workload executed in time interval $[t_i, t_0)$ is at least $M(\phi_i - y_i) + y_i$. Therefore, by the above discussions, we know that

$$M(\phi_i - y_i) + y_i < \phi_i\mu_k = \phi_i(M - (M-1)\rho) \Rightarrow y_i > \rho\phi_i, \tag{12}$$

since $M \geq 2$. At time t_0, the remaining execution time of the jobs of task τ_i that arrived before t_0 in schedule S is at most $\lceil \phi_i/T_i \rceil C_i - \rho\phi_i$. Note that the existence of t_i in our definition means that $\lceil \phi_i/T_i \rceil C_i - y_i > 0$, i.e., $\lceil \phi_i/T_i \rceil C_i - \rho\phi_i > 0$.

The workload of task τ_i that is executed in the time interval $[t_i, t_d)$ in schedule S is at most $work_i(t_d - t_i) = work_i(\Delta + \phi_i)$. The workload of task τ_i that is executed in the time interval $[t_i, t_0)$ is at least $y > \rho\phi_i$. Therefore, the workload of task τ_i that is executed in the time interval $[t_0, t_d)$ in schedule S is upper bounded by $work_i(\Delta + \phi_i) - \rho\phi_i$.

The rest of the proof is to provide an upper bound of $work_i(\Delta + \phi_i) - \rho\phi_i$ for any arbitrary $\phi_i > 0$. The proof involves some detailed manipulations of the workload function. Before proceeding, we explain two basic properties of the workload function here by inspecting the periodicity of the workload function $work_i(t)$ where $p = \lfloor t/T_i \rfloor$, a non-negative integer:

- For $t = pT_i + x$ with $0 \leq x$, the recursion $work_i(pT_i + x) = pC_i + work_i(x)$ holds.
- For $t = pT_i + x$ with $0 \leq x \leq C_i$, the simplification $work_i(pT_i + x) = pC_i + x$ holds.

To identify the exact value of $work_i(\Delta + \phi_i)$, we define the following variables p_1, p_2, q_1, and q_2 for brevity:

- Let p_1 be $\lceil \phi_i / T_i \rceil - 1$ and q_1 be $\phi_i - p_1 T_i$, i.e., $p_1 + 1$ is the number of jobs of task τ_i that can be released in $[t_i, t_0]$. By definition $\phi_i > 0$, which implies that p_1 is a non-negative integer, $0 < q_1 \leq T_i$, and $\phi_i = p_1 T_i + q_1$.
- Let p_2 be $\lceil (\Delta - C_i)/T_i \rceil - 1$ and q_2 be $\Delta - C_i - p_2 T_i$, i.e., $p_2 + 1$ is the number of jobs of task τ_i that can be released in $[t_0 + C_i, t_d]$. Due to the assumption $\Delta > C_i$, we know that p_2 is a non-negative integer, $0 < q_2 \leq T_i$, and $\Delta - C_i = p_2 T_i + q_2$.

By the above definition, we achieve $\phi_i + \Delta = (p_1 + p_2)T_i + q_1 + q_2 + C_i$, and

$$\begin{aligned}
&work_i(\Delta + \phi_i) - \rho\phi_i \\
&= work_i((p_1 + p_2)T_i + q_1 + q_2 + C_i) - \rho(p_1 T_i + q_1) \\
&= work_i(p_2 T_i + q_1 + q_2 + C_i) + p_1 C_i - \rho(p_1 T_i + q_1) \\
&= work_i(p_2 T_i + q_1 + q_2 + C_i) + p_1 U_i T_i - \rho(p_1 T_i + q_1) \\
&\leq work_i(p_2 T_i + q_1 + q_2 + C_i) - \rho q_1
\end{aligned} \tag{13}$$

where the inequality is due to the assumption that $0 \leq U_i \leq \rho$. We will prove that the right-hand side of Eq. (9) is a safe upper bound on the condition in Eq. (13). By the definition of q_1 and q_2, we know that $0 \leq q_1 + q_2 \leq 2T_i$, i.e., $C_i \leq p_2 T_i + q_1 + q_2 + C_i \leq 2T_i + C_i$. Depending on the value of $q_1 + q_2$, there are four cases for different (linear or constant) segments of $work_i(p_2 T_i + q_1 + q_2 + C_i)$ to be analyzed:

- **Case 1:** $0 \leq q_1 + q_2 \leq T_i - C_i$: That is, $p_2 T_i + C_i \leq p_2 T_i + q_1 + q_2 + C_i \leq p_2 T_i + T_i$. Therefore, $work_i(p_2 T_i + C_i) \leq work_i(p_2 T_i + q_1 + q_2 + C_i) \leq work_i(p_2 T_i + T_i)$. Since $work_i(p_2 T_i + C_i) = work_i(p_2 T_i + T_i) = (p_2 + 1)C_i$, we have

  $$\text{RHS. of Eq. (13)} = (p_2 + 1)C_i - \rho q_1 \leq work_i(p_2 T_i + C_i + q_2) = work_i(\Delta),$$

  where $\leq$ is due to $\rho \geq 0$ and $q_1 > 0$.
- **Case 2:** $T_i - C_i < q_1 + q_2 \leq T_i$: By definition, when p_2 is a nonnegative integer and $0 < x \leq C_i$, $work_i((p_2+1)T_i + x) = (p_2+1)C_i + x$. By $T_i - C_i < q_1 + q_2 \leq T_i$, we know that $(p_2+1)T_i < p_2 T_i + q_1 + q_2 + C_i \leq (p_2+1)T_i + C_i$. Therefore, $work_i(p_2 T_i + q_1 + q_2 + C_i) = (p_2 + 1)C_i + (p_2 T_i + q_1 + q_2 + C_i - (p_2 + 1)T_i) = (p_2 + 1)C_i + (q_1 + q_2 + C_i - T_i)$. Let η be $T_i - (q_1 + q_2)$. By definition $\eta \geq 0$. Therefore,

  $$\begin{aligned}
  \text{RHS. of Eq. (13)} =& (p_2 + 1)C_i + (C_i - \eta) - \rho(T_i - q_2 - \eta) \\
  =& (p_2 + 1)C_i + (C_i - \rho(T_i - q_2)) + \eta(\rho - 1) \\
  \leq& (p_2 + 1)C_i + \max\{0, C_i - \rho(T_i - q_2)\},
  \end{aligned}$$

  where $\leq$ is due to $0 \leq \rho \leq 1$ and $\eta \geq 0$.
- **Case 3:** $T_i < q_1 + q_2 \leq 2T_i - C_i$: Thus, $work_i(p_2 T_i + q_1 + q_2 + C_i) = (p_2 + 2)C_i$, and

  $$\text{RHS. of Eq. (13)} = (p_2 + 1)C_i + C_i - \rho q_1 \leq (p_2 + 1)C_i + \max\{0, C_i - \rho(T_i - q_2)\},$$

  where $\leq$ is due to $\rho \geq 0$ and $q_1 + q_2 > T_i$.
- **Case 4:** $2T_i - C_i < q_1 + q_2 \leq 2T_i$: In this case $work_i(p_2 T_i + q_1 + q_2 + C_i)$ is equal to $(p_2 + 2)C_i + (q_1 + q_2 + C_i - 2T_i)$, similar to the analysis in Case 2. Let η be $2T_i - (q_1 + q_2)$. By definition $\eta \geq 0$. Therefore,

  $$\begin{aligned}
  \text{RHS. of Eq. (13)} =& (p_2 + 1)C_i + 2C_i - \eta - \rho(2T_i - q_2 - \eta) \\
  =& (p_2 + 1)C_i + C_i + T_i(U_i - \rho) - \eta(1 - \rho) - \rho(T_i - q_2) \\
  \leq& (p_2 + 1)C_i + \max\{0, C_i - \rho(T_i - q_2)\},
  \end{aligned}$$

where $\leq$ is due to $0 < U_i \leq \frac{C'_k}{D'_k} \leq \rho \leq 1$ and $\eta \geq 0$, i.e., $U_i - \rho \leq 0$ and $-\eta(1-\rho) \leq 0$. Since $0 < q_1 + q_2 \leq 2T_i$, we know that $work_i(\Delta)$ is a safe upper bound for **Case 1** and that $(p_2+1)C_i + \max\{0, C_i - \rho(T_i - q_2)\}$ is a safe upper bound for the other cases, and we reach the conclusion of this lemma. ◀

▶ **Lemma 3.9.** *If $U_i \leq \rho$, then $\omega_i^{heavy}(\Delta) \geq \omega_i^{light}(\Delta)$ for all $\Delta > 0$.*

Proof. This inequality can be proved formally, but can also be derived by following the definitions. When $0 < \Delta \leq C_i$, the inequality holds naturally. In the proof of Lemma 3.8, *the workload of task τ_i that is executed in the time interval $[t_i, t_d)$ in schedule S is at most $work_i(t_d - t_i) = work_i(\Delta + \phi_i)$.* Since $\phi_i \leq D_i$, we know that $\omega_i^{light}(\Delta) \leq work_i(\Delta + \phi_i) \leq work_i(\Delta + D_i) = \omega_i^{heavy}(\Delta)$. ◀

Here is a short summary of the information provided by Lemmas 3.6, 3.7, and 3.8.

- According to Lemma 3.6, at time t_0, there are at most $\lceil M - (M-1)\rho \rceil - 1 = \lceil \mu_k \rceil - 1$ carry-in tasks.
- Among the $\lceil \mu_k \rceil - 1$ carry-in tasks, there are two types of carry-in tasks, i.e., *heavy* and *light* tasks. A light carry-in task τ_i can be described by $\omega_i^{light}(\Delta)$ from Eq. (9) if the utilization is no more than ρ and a heavy carry-in task τ_i can be described by $\omega_i^{heavy}(\Delta)$ from Eq. (8). By observing the conditions in Eqs. (8) and (9), we know that $work_i(\Delta) \leq \omega_i^{light}(\Delta) \leq \omega_i^{heavy}(\Delta)$.
- Since ρ is a user-defined parameter, a smaller ρ implies a larger μ_k, i.e., potentially more carry-in tasks and more heavy carry-in tasks. By constrast, a larger ρ implies a smaller μ_k, i.e., potentially less carry-in tasks and more light carry-in tasks. Therefore, *a larger ρ is better for minimizing the carry-in workload.*
- However, the window of interest $[t_0, t_d)$ is defined by the condition $\Omega(t_0, t_d) \geq M - (M-1)\rho$. The window of interest is smaller when ρ is larger. As a result, there is no monotonicity with respect to the schedulability test for setting the value of ρ.

▶ **Theorem 3.10.** *Task τ_k is schedulable by the given global fixed-priority scheduling if*

$$\forall \ell \in \mathbb{N}, \exists 1 \geq \rho \geq \ell C_k/((\ell-1)T_k + D_k), \forall \Delta \geq (\ell-1)T_k + D_k$$

$$\ell C_k + \sum_{\tau_i \in \boldsymbol{T}^{carry}} \omega_i^{diff}(\Delta, \rho) + \sum_{i=1}^{k-1} work_i(\Delta) \leq \Delta \cdot \mu_k \tag{14}$$

holds, where $\mu_k = M - (M-1)\rho$,

$$\omega_i^{diff}(\Delta, \rho) = \begin{cases} \omega_i^{heavy}(\Delta) - work_i(\Delta) & \text{if } U_i > \rho \\ \omega_i^{light}(\Delta) - work_i(\Delta) & \text{if } U_i \leq \rho \end{cases} \tag{15}$$

and $\boldsymbol{T}^{carry}$ is the set of the $\lceil \mu_k \rceil - 1$ tasks among the $k-1$ higher-priority tasks with the largest values of $\omega_i^{diff}(\Delta, \rho)$. If $D_k \leq T_k$, we only need to consider $\ell = 1$.

Proof. We prove this theorem by contrapositive, i.e., task τ_k misses its deadline first at time t_d in a global fixed-priority preemptive schedule S. We know that t_a can be defined for schedule S, and t_0, i.e., $\Omega(t_0, t_d) \geq M - (M-1) \times \frac{C'_k}{D'_k}$ in Definition 3.3 can be defined for any ρ with $1 \geq \rho \geq \ell C_k/((\ell-1)T_k + D_k)$ due to Lemma 3.4.

By the existence of t_d, the choice of ρ, and the definition of t_0 in Definition 3.3, we know that the deadline miss of task τ_k at time t_d in the schedule S implies

$$\exists \ell \in \mathbb{N}, \forall 1 \geq \rho \geq \ell C_k/((\ell-1)T_k + D_k), \exists \Delta = t_d - t_0, \qquad \Omega(t_0, t_d) \geq M - (M-1)\rho \tag{16}$$

By the fact that $C_k^* < C_k' = \ell C_k$ and the definition of $\Omega()$, we have

$$\Omega(t_0, t_d) = \frac{C_k^* + E(t_0, t_d)}{t_d - t_0} < \frac{\ell C_k + E(t_0, t_d)}{t_d - t_0} \tag{17}$$

By Lemma 3.6, for a specific ρ, there are at most $\lceil M - (M-1)\rho \rceil - 1 = \lceil \mu_k \rceil - 1$ higher-priority carry-in tasks at time t_0 and the other higher-priority tasks do not have any unfinished job at time t_0. Suppose that $\mathbf{T}^{heavy}$ and $\mathbf{T}^{light}$ are the sets of the heavy and light carry-in tasks at time t_0, respectively. By Lemma 3.6, $|\mathbf{T}^{heavy}| + |\mathbf{T}^{light}| \leq \lceil \mu_k \rceil - 1$. Therefore, by using Lemmas 3.7 and 3.8 and 3.9, we have

$$\begin{aligned} E(t_0, t_d) &\leq \sum_{\tau_i \in \mathbf{T}^{heavy}} \omega_i^{heavy}(\Delta) + \sum_{\tau_i \in \mathbf{T}^{light}} \omega_i^{light}(\Delta) \\ &= \sum_{\tau_i \in \mathbf{T}^{heavy}} \left(\omega_i^{heavy}(\Delta) - work_i(\Delta)\right) + \sum_{\tau_i \in \mathbf{T}^{light}} \left(\omega_i^{light}(\Delta) - work_i(\Delta)\right) + \sum_{i=1}^{k-1} work_i(\Delta) \\ &\leq \sum_{\tau_i \in \mathbf{T}^{carry}} \omega_i^{diff}(\Delta, \rho) + \sum_{i=1}^{k-1} work_i(\Delta) \end{aligned} \tag{18}$$

where $\omega_i^{diff}(\Delta, \rho)$ is defined in Eq. (15), and $\mathbf{T}^{carry}$ is defined in the statement of the theorem.

By Eqs. (16), (17), and (18), and the fact $t_d - t_a \geq D_k' = (\ell - 1)T_k + D_k$, the deadline miss of task τ_k at t_d implies

$$\begin{aligned} &\exists \ell \in \mathbb{N}, \forall 1 \geq \rho \geq \ell C_k / ((\ell - 1)T_k + D_k), \exists \Delta \geq (\ell - 1)T_k + D_k \\ &\ell C_k + \sum_{\tau_i \in \mathbf{T}^{carry}} \omega_i^{diff}(\Delta, \rho) + \sum_{i=1}^{k-1} work_i(\Delta) > \Delta \cdot \mu_k \end{aligned} \tag{19}$$

Therefore, the negation of the above necessary condition for the deadline miss of task τ_k at time t_d is a safe sufficient schedulability test. We reach the conclusion of the schedulability test.

When $D_k \leq T_k$, since t_d is the earliest moment in the schedule S with a deadline miss of task τ_k, we know that t_a is by definition $t_d - D_k$ and ℓ is 1. Therefore, we only have to consider $\ell = 1$ when $D_k \leq T_k$. ◀

The schedulability test described in Theorem 3.10 can be informally explained as follows: 1) it requires to test all the possible positive integers for ℓ, like the busy-window concept, 2) it has to find a ρ value in the specified range, and 3) for the specified combination of ℓ and ρ, we have to test whether the condition in Eq. (14) holds for every $\Delta \geq (\ell - 1)T_k + D_k$.

3.3 Remarks on Implementing Theorem 3.10

Unfortunately, due to the following issues, implementing the schedulability test in Theorem 3.10 directly would lead to a high time complexity:

- **Issue 1 due to Δ:** For specific ℓ and ρ, testing the schedulability condition in Eq. (14) requires to evaluate all $\Delta \geq (\ell - 1)T_k + D_k$. Suppose that $HP(k)$ is the hyper-period of $\{\tau_1, \tau_2, \ldots, \tau_{k-1}\}$, i.e., the least common multiple of the periods of $\tau_1, \tau_2, \ldots, \tau_{k-1}$. Since $work_i(\Delta) + HP(k)U_i = work_i(\Delta + HP(k))$, $\omega_i^{light}(\Delta) + HP(k)U_i = \omega_i^{light}(\Delta + HP(k))$, and $\omega_i^{heavy}(\Delta) + HP(k)U_i = \omega_i^{heavy}(\Delta + HP(k))$, we only have to test $\Delta \in [(\ell - 1)T_k + D_k, (\ell - 1)T_k + D_k + HP(k)]$, as long as $\sum_{i=1}^{k-1} U_i \leq \mu_k$. However, the time complexity can still be exponential. We will explain how to reduce this complexity by using safe upper bounds in Section 4.

- **Issue 2 due to ρ**: For a specific ℓ, the schedulability condition in Eq. (14) is dependent on the selection of ρ. If ρ is smaller, then μ_k is larger, and vice versa. A smaller ρ increases the right-hand side in the schedulability test in Eq. (14), but it also increases the left-hand side, since there are potentially more carry-in tasks. One simple strategy to find a suitable ρ instead of searching for all values of ρ is to start from $\rho = \ell C_k/((\ell-1)T_k + D_k$ and increase ρ to the next (higher) U_i for certain higher-priority task τ_i if necessary. Therefore, in the worst case, we only have to consider k different ρ values. We will deal with this in Theorems 4.4 and 4.5 in Section 4.
- **Issue 3 due to ℓ**: We need to consider all positive integer values of ℓ in the schedulability condition in Eq. (14), as the test is only valid when the condition holds for all $\ell \in \mathbb{N}$. Therefore, if we only test some ℓ, it is necessary to prove that the other ℓ configurations are also covered even though they are not tested. We will explain how to deal with this in Theorems 4.6 and 4.7 in Section 4.

4 Efficient Schedulability Tests

In this section we provide several schedulability tests based on approximate workload functions to test the schedulability of task τ_k more efficiently. The following three lemmas approximate the *piecewise linear* workload function $work_i(\Delta)$, $\omega_i^{heavy}(\Delta)$ and $\omega_i^{light}(\Delta)$ by *linear* functions with respect to Δ for any $\Delta \geq 0$.

▶ **Lemma 4.1.** *When $0 \leq U_i \leq 1$, for any $\Delta \geq 0$,*

$$work_i(\Delta) \leq C_i - C_iU_i + U_i\Delta. \tag{20}$$

Proof. This inequality was already stated in Eq. (5) by Bini et al. [14] as a fact. Here, we provide the proof for completeness. Suppose that Δ is $p_3T_i + q_3$, where p_3 is $\left\lfloor \frac{\Delta}{T_i} \right\rfloor$ and q_3 is $\Delta - \left\lfloor \frac{\Delta}{T_i} \right\rfloor T_i$. Therefore, we know $U_i\Delta = p_3C_i + q_3U_i$ and $work_i(\Delta) = p_3C_i + \min\{C_i, q_3\}$. We have to consider two cases:

- If $q_3 \leq C_i$: we have

$$\begin{aligned} work_i(\Delta) = &\; p_3C_i + q_3 \leq p_3C_i + C_i - (C_i - q_3) \\ \leq_1 &\; p_3C_i + C_i - (C_i - q_3)U_i = C_i - C_iU_i + U_i\Delta, \end{aligned}$$

  where $\leq_1$ is due to $0 \leq U_i \leq 1$ and $C_i - q_3 \geq 0$.
- If $q_3 > C_i$: we have

$$work_i(\Delta) = \; p_3C_i + C_i \leq p_3C_i + C_i + (q_3 - C_i)U_i = C_i - C_iU_i + U_i\Delta,$$

  where $\leq$ is due to $0 \leq U_i \leq 1$ and $q_3 - C_i > 0$. ◀

▶ **Lemma 4.2.** *For any $\Delta \geq 0$,*

$$\omega_i^{heavy}(\Delta) \leq C_i + U_iD_i - C_iU_i + U_i\Delta. \tag{21}$$

Proof. Due to Lemma 3.7 and Lemma 4.1, the inequality holds. ◀

▶ **Lemma 4.3.** *If $U_i \leq \rho \leq 1$, for any $\Delta \geq 0$,*

$$\omega_i^{light}(\Delta) \leq C_i - C_iU_i + U_i\Delta. \tag{22}$$

Proof. We consider the three upper bounds in Lemma 3.8 individually. When $\Delta \leq C_i$, this follows from Lemma 4.1 directly. When $\Delta > C_i$ and $\omega_i^{light}(\Delta) = work_i(\Delta)$, it holds due to Lemma 4.1 as well.

For the last case we have to bound $(p_2+1)C_i + \max\{0, C_i - \rho(T_i - q_2)\}$, as defined in Lemma 3.8. By the definition of p_2 and q_2, i.e., $\Delta - C_i = p_2T_i + q_2$, in the statement of Lemma 3.8, we have $p_2 + 1 = \lceil(\Delta - C_i)/T_i\rceil$ and $(p_2+1)C_i = work_i(p_2T_i + C_i) = work_i(\Delta - q_2)$. Therefore, for any $\Delta > C_i$, if $C_i - \rho(T_i - q_2) \geq 0$, we get

$$\begin{aligned}
\omega_i^{light}(\Delta) =\ & (p_2+1)C_i + C_i - \rho(T_i - q_2)\\
=\ & work_i(\Delta - q_2) + C_i - \rho(T_i - q_2)\\
\leq_1\ & C_i - C_iU_i + U_i(\Delta - q_2) + C_i - \rho(T_i - q_2)\\
=\ & C_i - C_iU_i + U_i\Delta - q_2(U_i - \rho) - T_i(\rho - U_i)\\
=\ & C_i - C_iU_i + U_i\Delta + (T_i - q_2)(U_i - \rho)\\
\leq_2\ & C_i - C_iU_i + U_i\Delta,
\end{aligned}$$

where $\leq_1$ is due to Lemma 4.1 and $\leq_2$ is due to $q_2 \leq T_i$ and $U_i \leq \rho$. For any $\Delta > C_i$, if $C_i - \rho(T_i - q_2) < 0$, similarly, we have

$$\begin{aligned}
\omega_i^{light}(\Delta) =& (p_2+1)C_i = work_i(p_2T_i + C_i) \leq work_i(p_2T_i + C_i + q_2) = work_i(\Delta)\\
\leq& C_i - C_iU_i + U_i\Delta.
\end{aligned}$$

Therefore, we reach the conclusion. ◀

With the help of the above lemmas for safe approximations, we can now safely and efficiently handle the schedulability test for specific ℓ and ρ in the following theorem. This handles **Issue 1** explained at the end of Section 3.

▶ **Theorem 4.4.** *Task τ_k is schedulable by the given global fixed-priority scheduling if*

$$\forall \ell \in \mathbb{N}, \exists 1 \geq \rho \geq \ell C_k/((\ell-1)T_k + D_k)$$
$$\frac{\ell C_k}{D'_k} + \sum_{\tau_i \in \boldsymbol{T}^{carry-approx}} \frac{\gamma_i U_i D_i}{D'_k} + \sum_{i=1}^{k-1}\left(\frac{C_i - C_iU_i}{D'_k} + U_i\right) \leq \mu_k, \tag{23}$$

where $\mu_k = M - (M-1)\rho$ with $1 \geq \rho \geq \ell C_k/((\ell-1)T_k + D_k)$, D'_k is $(\ell-1)T_k + D_k$,

$$\gamma_i = \begin{cases} 1 & \text{if } U_i > \rho \\ 0 & \text{if } U_i \leq \rho \end{cases} \tag{24}$$

and $\boldsymbol{T}^{carry-approx}$ is the set of the $\lceil\mu_k\rceil - 1$ tasks among the $k-1$ higher-priority tasks with the largest values of $\gamma_i U_i D_i$. Note that $|\boldsymbol{T}^{carry-approx}|$ can be smaller than $\lceil\mu_k\rceil - 1$ if the number of tasks with $U_i > \rho$ is less than $\lceil\mu_k\rceil - 1$. If $D_k \leq T_k$, we only need to consider $\ell = 1$.

Proof. We prove that the condition in this theorem is a safe upper bound of that in Theorem 3.10. For specific ℓ, ρ, Δ, we can find $\mathbf{T}^{carry}$ as defined in Theorem 3.10. By Lemmas 4.1, 4.2, and 4.3 and the assumptions $\Delta \geq (\ell-1)T_k + D_k = D'_k$ and $0 < U_i \leq 1 \forall \tau_i$, we have

$$\begin{aligned}
&\ell C_k + \sum_{\tau_i \in \mathbf{T}^{carry}} \omega_i^{diff}(\Delta, \rho) + \sum_{i=1}^{k-1} work_i(\Delta) \\
\leq & \ell C_k + \sum_{\tau_i \in \mathbf{T}^{carry}} \gamma_i U_i D_i + \sum_{i=1}^{k-1} (C_i - C_i U_i + U_i \Delta) && (25)\\
\leq & \ell C_k + \sum_{\tau_i \in \mathbf{T}^{carry-approx}} \gamma_i U_i D_i + \sum_{i=1}^{k-1} (C_i - C_i U_i + U_i \Delta) && (26)\\
\leq & \Delta \cdot \left(\frac{\ell C_k}{D'_k} + \sum_{\tau_i \in \mathbf{T}^{carry-approx}} \frac{\gamma_i U_i D_i}{D'_k} + \sum_{i=1}^{k-1} \left(\frac{C_i - C_i U_i}{D'_k} + U_i \right) \right) && (27)
\end{aligned}$$

Therefore, the test in Theorem 3.10 can be safely over-approximated as follows:

$$\forall \ell \in \mathbb{N}, \exists 1 \geq \rho \geq \ell C_k/((\ell-1)T_k + D_k)T_k + D_k$$
$$\frac{\ell C_k}{D'_k} + \left(\sum_{\tau_i \in \mathbf{T}^{carry-approx}} \frac{\gamma_i U_i D_i}{D'_k} \right) + \sum_{i=1}^{k-1} \left(\frac{C_i - C_i U_i}{D'_k} + U_i \right) \leq \mu_k \tag{28}$$

◀

Theorem 4.4 provides two interesting implications to handle **Issue 2**. Firstly, if $U_i \leq \rho$, the linear approximation of $work_i(\Delta)$ by considering task τ_i as a non-carry-in task in Lemma 4.1 is the same as the linear approximation of $\omega_i^{light}(\Delta)$ by considering task τ_i as a carry-in task in Lemma 4.3. Therefore, the carry-in tasks are only effective for those tasks τ_i with $U_i > \rho$. Secondly, for a specific ℓ, deciding whether a specific ρ exists to pass the test in Eq. (23) can be done by only testing a finite number of ρ values, i.e. by starting from $\rho = \ell C_k/((\ell-1)T_k + D_k$ and increasing ρ to the next (higher) values where $\mathbf{T}^{carry-approx}$ changes. This means either 1) $\rho = U_i$ for certain higher-priority task τ_i, i.e., the summation can be larger with the same number of summands; or 2) $\mu_k = M - (M-1)\rho$ is an integer, i.e., the number of summands increases. This only has time complexity $O((k+M)\log(k+M))$, mainly due to the sorting, when proper data structures are used. Details can be found in Appendix of the full version [21].

4.1 Linear-Time Schedulability Tests

The time complexity of Theorem 4.4 is due to the search of possible ρ values. Nevertheless, we can directly set ρ to $U_{\delta,k}^{\max}$ which implies that there is no carry-in task in the linear-approximation form. With this simplification, we can conclude different schedulability tests in Theorems 4.5, 4.6, and 4.7. Although these tests are not superior to Theorem 4.4, our main target is the test in Theorem 4.7, which will be used *mainly to derive the speedup bounds later in Theorem 5.2.*

▶ **Theorem 4.5.** *Task τ_k is schedulable by the given global fixed-priority scheduling if* $\forall \ell \in \mathbb{N}$

$$\frac{\ell C_k}{D'_k} + \sum_{i=1}^{k-1} \left(\frac{C_i - C_i U_i}{D'_k} + U_i \right) \leq (M - (M-1) U_{\delta,k}^{\max}) \tag{29}$$

holds, where D'_k is $(\ell-1)T_k + D_k$.

Proof. This comes directly from Theorem 4.4 by setting ρ to $U^{\max}_{\delta,k}$ and the facts that $U^{\max}_{\delta,k} \geq U_i$ for $i = 1, 2, \dots, k-1$ and $U^{\max}_{\delta,k} \geq \delta_k \geq \ell C_k/((\ell-1)T_k + D_k)$ by definition. ◀

▶ **Theorem 4.6.** *Suppose that $D_k > T_k$. Let b be $\frac{D_k - T_k}{T_k}$. Task τ_k is schedulable by the given global fixed-priority scheduling algorithm if:*

$$\sum_{i=1}^{k} U_i \leq (M - (M-1)U^{\max}_{\delta,k}), \qquad \textit{when } bU_k - \sum_{i=1}^{k-1} \frac{C_i - C_iU_i}{T_k} > 0 \tag{30}$$

$$\frac{C_k}{D_k} + \sum_{i=1}^{k-1}\left(\frac{C_i - C_iU_i}{D_k} + U_i\right) \leq (M - (M-1)U^{\max}_{\delta,k}), \qquad \textit{otherwise} \tag{31}$$

Proof. For a given ℓ, the left-hand side in Eq. (29) can be rephrased as:

$$F(\ell) = \frac{\ell C_k}{D'_k} + \sum_{i=1}^{k-1}\left(\frac{C_i - C_iU_i}{D'_k} + U_i\right) = \frac{\ell U_k + \sum_{i=1}^{k-1} \frac{C_i - C_iU_i}{T_k}}{\ell + b} + \sum_{i=1}^{k-1} U_i \tag{32}$$

The first order derivative of $F(\ell)$ with respect to ℓ is:

$$\frac{\partial F(\ell)}{\partial \ell} = \frac{bU_k - \sum_{i=1}^{k-1} \frac{C_i - C_iU_i}{T_k}}{(\ell + b)^2}. \tag{33}$$

We have to consider two cases:

- Case 1: if $bU_k - \sum_{i=1}^{k-1} \frac{C_i - C_iU_i}{T_k} > 0$, then $F(\ell)$ is an increasing function with respect to ℓ. Therefore, $F(\ell)$ is maximized when $\ell \to \infty$, i.e., $F(\ell) \leq \sum_{i=1}^{k} U_i$.
- Case 2: if $bU_k - \sum_{i=1}^{k-1} \frac{C_i - C_iU_i}{T_k} \leq 0$, then $F(\ell)$ is a non-increasing function with respect to ℓ. Therefore, $F(\ell)$ is maximized when $\ell \to 1$, i.e., $F(\ell) \leq \frac{C_k}{D_k} + \sum_{i=1}^{k-1}(\frac{C_i - C_iU_i}{D_k} + U_i)$. ◀

▶ **Theorem 4.7.** *Task τ_k is schedulable by the given global fixed-priority scheduling if*

$$\delta_k + \sum_{i=1}^{k-1}\left(\frac{C_i - C_iU_i}{D_k} + U_i\right) \leq M - (M-1)U^{\max}_{\delta,k} \tag{34}$$

Proof. Based on Theorem 4.5 and the two facts that $D'_k = (\ell - 1)T_k + D_k \geq D_k$ and $\delta_k \geq \ell C_k/((\ell-1)T_k + D_k)$ for all $\ell \in \mathbb{N}$, we reach the conclusion. ◀

4.2 Dominance

We now show analytical dominance among the tests presented above and in Theorem 3.10 in the following corollary. A test $\mathcal{B}_1$ analytically dominates another test $\mathcal{B}_2$ if the schedulability condition in $\mathcal{B}_1$ always dominates that in $\mathcal{B}_2$. This means, if task τ_k is deemed schedulable by $\mathcal{B}_2$, task τ_k is also deemed schedulable by $\mathcal{B}_1$.

▶ **Corollary 4.8.** *For arbitrary-deadline sporadic real-time systems under global fixed-priority scheduling, the schedulability tests have the following dominance relations.*

- *Theorem 3.10 analytically dominates Theorem 4.4.*
- *Theorem 4.4 analytically dominates Theorem 4.5.*
- *Theorem 4.5 is equivalent to the test in Theorem 4.6.*
- *Theorem 4.6 analytically dominates Theorem 4.7.*

Proof. They follow directly from the above analyses. The reason why Theorems 4.5 and 4.6 are equivalent is because the conditions in Theorem 4.6 represent exactly the worst-case ℓ selection in Theorem 4.5. The other cases are obvious. ◀

Although we will show in Theorem 5.3 that all the above schedulability tests have the same speedup bound for global DM, the *performance* of the schedulability tests in this section can be very different *in practice*. Chen et al. [19] have recently shown that "*Speedup factors … often lack the power to discriminate between the performance of different scheduling algorithms and schedulability tests even though the performance of these algorithms and tests may be very different when viewed from the perspective of empirical evaluation.*" To avoid concluding an algorithm with a reasonable speedup bound but practically not useful, we performed a series of experiments and present the results in Section 6. Moreover, according to the experimental results, the domination relations among Theorems 4.4, 4.5, and 4.7 are strict, i.e., there is a concrete input instance that is deemed schedulable by a dominating schedulability test but is not deemed schedulable by a dominated schedulability test.

5 Global Deadline-Monotonic (DM) Scheduling

After presenting the schedulability tests for any global fixed-priority scheduling algorithms, we focus ourselves on global DM in this section. We will discuss the speedup upper bound and the speedup lower bound. Baruah and Fisher [8] showed that global DM has a speedup upper bound of $2 + \sqrt{3} \approx 3.73$ compared to the optimal schedules, based on the test restated in Theorem 2.4. This is the best known upper bound on speedup factors for arbitrary-deadline sporadic task systems under global fixed-priority scheduling. Evaluating $\textsc{load}(k)$ in Theorem 2.4 requires to calculate $\sum_{i=1}^{k} \text{DBF}(\tau_i, t)/t$ at all time points t. This means, the naïve implementation has an exponential-time complexity. There are more efficient methods, as discussed by Baruah and Bini [6], but the time complexity remains exponential. Although it is possible to approximate $\textsc{load}(k)$ by using approximate demand bound functions in polynomial time, this is at a price of higher $\textsc{load}(k)$. We show that the test in Theorem 2.4 is over-pessimistic and is analytically dominated by our linear-time schedulability test in Theorem 4.7 under global DM.

▶ **Corollary 5.1.** *For global DM, the schedulability test in Theorem 4.7 analytically dominates the schedulability test in Theorem 2.4 proposed by Baruah and Fisher [8].*

Proof. This is due to the following facts:
- By definition, $\textsc{load}(k) \geq \text{limit}_{t\to\infty} \sum_{i=1}^{k} \text{DBF}(\tau_i, t)/t = \sum_{i=1}^{k} U_i$.
- Since $D_i \leq D_k$ in global DM for $i = 1, 2, \ldots, k-1$, we know that $\frac{\sum_{i=1}^{k} \text{DBF}(\tau_i, D_k)}{D_k} \geq \sum_{i=1}^{k} \frac{C_i}{D_k}$. Therefore, $\textsc{load}(k) \geq \sum_{i=1}^{k} \frac{C_i}{D_k}$.

Combining these facts, we get

$$\delta_k + \sum_{i=1}^{k-1} \left(\frac{C_i - C_i U_i}{D_k} + U_i \right) \leq \sum_{i=1}^{k} \frac{C_i}{D_k} + U_i \leq 2\textsc{load}(k). \tag{35}$$

Since we know that the right-hand side in Eq. (4), i.e., $M - (M-1)\delta_{\max}(k)$, is less than or equal to $M - (M-1)U_{\delta,k}^{\max}$ in Eq. (34), we reach the conclusion. ◀

▶ **Theorem 5.2.** *Global DM has a speedup bound of $3 - \frac{1}{M}$, with respect to the optimal schedule, when $M \geq 2$.*

Proof. We only prove the speedup bound by using the schedulability test in Theorem 4.7. Due to the dominance properties in Corollary 4.8, such a bound also holds for the schedulability tests from Theorems 3.10, 4.4, 4.5, and 4.6.

Suppose that task τ_k is not schedulable by global DM. Since $D_i \leq D_k$ for any $i = 1, 2, \dots, k-1$ under global DM, we know $\text{DBF}(\tau_i, D_k) \geq C_i$. Therefore, under global DM, $\sum_{i=1}^{k} \frac{C_i}{MD_k} \leq \sum_{i=1}^{k} \frac{\text{DBF}(\tau_i,D_k)}{MD_k} \leq \frac{\sum_{\tau_i \in \mathbf{T}} \text{DBF}(\tau_i,D_k)}{MD_k} \leq \max_{t>0} \frac{\sum_{\tau_i \in \mathbf{T}} \text{DBF}(\tau_i,t)}{Mt}$.

By the assumption that task τ_k is also deemed not schedulable by Theorem 4.7, we have

$$\delta_k + \sum_{i=1}^{k-1} \left(\frac{C_i - C_i U_i}{D_k} + U_i \right) > M - (M-1) U_{\delta,k}^{\max}$$

$$\Rightarrow \quad \sum_{i=1}^{k} \frac{C_i}{MD_k} + \sum_{i=1}^{k} \frac{U_i}{M} > 1 - \left(1 - \frac{1}{M}\right) U_{\delta,k}^{\max}$$

$$\Rightarrow \quad \sum_{i=1}^{k} \frac{C_i}{MD_k} + \sum_{i=1}^{k} \frac{U_i}{M} + \left(1 - \frac{1}{M}\right) U_{\delta,k}^{\max} > 1$$

$$\overset{x+x+(1-1/M)x>1}{\Rightarrow} \quad \max\left\{ \sum_{i=1}^{k} \frac{C_i}{MD_k}, \sum_{i=1}^{k} \frac{U_i}{M}, U_{\delta,k}^{\max} \right\} > \frac{1}{3 - 1/M} \tag{36}$$

Therefore, either $\max_{t>0} \frac{\sum_{\tau_i \in \mathbf{T}} \text{DBF}(\tau_i,t)}{Mt} \geq \sum_{i=1}^{k} \frac{C_i}{MD_k} > \frac{1}{3-1/M}$, or $\sum_{i=1}^{k} \frac{U_i}{M} > \frac{1}{3-1/M}$, or $\delta_{\max}(k) \geq U_{\delta,k}^{\max} > \frac{1}{3-1/M}$. By Lemma 2.3, we reach the conclusion of the speedup bound for global DM with respect to the optimal schedule. ◀

▶ **Theorem 5.3.** *For global DM, the schedulability tests in Theorems 3.10, 4.4, 4.5, 4.6, and 4.7 have a speedup bound of* $3 - \frac{1}{M}$*, with respect to the optimal schedule, when* $M \geq 2$.

Proof. This is due to Theorem 5.2 and Corollary 4.8, because all of the tests in Theorems 3.10, 4.4, 4.5, 4.6 dominate the test in Theorem 4.7 as presented in Corollary 4.8. ◀

▶ **Theorem 5.4.** *The speedup bound of global DM for arbitrary-deadline task systems is at least* $3 - \frac{3}{M+1}$.

Proof. The proof is based on a concrete task set. We specifically use the following task set $\mathbf{T}^{ad}$ with $N = 2M+1$ tasks. Let ε be an arbitrarily small positive real number such that $1/\varepsilon$ is an integer. Let $\eta \ll \varepsilon$ be an arbitrarily small positive number, that is used to enforce the priority assignment under global DM:

- $C_i = \frac{\varepsilon}{3}$, $T_i = \varepsilon$, $D_i = 1$, for $i = 1, 2, \dots, M$.
- $C_i = \frac{1}{3}$, $T_i = \infty$, $D_i = 1 + \eta$, for $i = M+1, M+2, \dots, 2M$.
- $C_i = \frac{1+\varepsilon}{3}$, $T_i = \infty$, $D_i = 1 + 2\eta$, for $i = 2M+1$

As the setting of $\eta \ll \varepsilon$ is just to enforce the indexing, *we will directly take* $\eta \to 0$ *here.* In the Appendix, we prove two properties: 1) $\mathbf{T}^{ad}$ is not schedulable by global DM under a concrete instance which releases all the tasks at time 0 and the subsequent jobs periodically. 2) There exists a feasible schedule for task set $\mathbf{T}^{ad}$ at any speed no lower than $\frac{1+\varepsilon}{3} + \frac{1+\varepsilon}{3M}$ under a concrete semi-partitioned multiprocessor schedule, i.e., $\{\tau_m, \tau_{m+M}\}$ assigned to processor m for $m = 1, 2, \dots, M$ and task τ_{2M+1} executed partially on each of the M processors. Therefore, a lower bound on the speedup bound of global DM is:

$$\lim_{\varepsilon \to 0} \frac{1}{\frac{1+\varepsilon}{3} + \frac{1+\varepsilon}{3M}} = \lim_{\varepsilon \to 0} \frac{3M}{(1+\varepsilon) \times (M+1)} = \frac{3M}{M+1} = 3 - \frac{3}{M+1}.$$ ◀

By Theorems 5.2 and 5.4, we can reach the conclusion that all the schedulability tests from Theorems 3.10, 4.4, 4.5, 4.6, and 4.7 are asymptotically tight with respect to speedup bounds. However, these tests have different performance with respect to the schedulability.

6 Evaluation

We evaluated the scheduling tests provided in this paper by comparing their acceptance ratio to the acceptance ratio of other algorithms, i.e., comparing the percentage of task sets accepted for the different schedulability tests, using different settings for the number of processors, the scheduling policy, and the ratio of the relative deadline to the period.

Evaluation Setup. We conducted evaluations for homogeneous multiprocessor systems with $M = 4$, $M = 8$, and $M = 16$ processors. We generated 100 task sets with cardinality of both $N = 5 \times M$ and $N = 10 \times M$, and utilization ranging from $M \times 5\%$ to $M \times 100\%$ in steps of $M \times 5\%$. The UUniFast-Discard method [13] was adopted to generate the utilization values of a set of N tasks under the target utilization. As suggested by Emberson et al. [28], the periods were generated according to a log-uniform distribution, with 1, 2, and 3 orders of magnitude, i.e., $[1ms-10ms]$, $[1ms-100ms]$, and $[1ms-1000ms]$. For each task, the relative deadline was set to the period multiplied with a value randomly drawn under a uniform distribution from a given interval I. We conducted evaluations using different interval, i.e., I was $[0.8, 2]$, $[0.8, 5]$, $[0.8, 10]$, $[1, 2]$, $[1, 5]$, or $[1, 10]$. To schedule the task sets, we applied global deadline-monotonic (DM) and global slack-monotonic (SM) [1] scheduling.

Whether the task set is schedulable under the given scheduling approach or not was tested using the following schedulability tests:

- LOAD: The load-based analysis by Baruah and Fisher in [9], only for DM scheduling.
- BAK: The test by Baker in Theorem 11 in [3].
- HC: The sufficient test in Corollary 2 by Huang and Chen in [31].
- OUR-4.4: The sufficient test in Theorem 4.4 in this paper.
- OUR-4.6: The sufficient test in Theorem 4.6 in this paper.
- OUR-4.7: The sufficient test in Theorem 4.7 in this paper.

We also checked if a task set was schedulable according to at least one of the tests, denoted as *ALL. We only present a small set of the conducted tests here. The diagrams of all conducted evaluations can be found in [20].*

Evaluation Results. Figure 3 shows the evaluations under the setting used in the paper by Huang and Chen [31]. They used DM scheduling on $M = 8$ processors, a task set containing 40 tasks and ratios of $\frac{D_i}{T_i} \in [0.8, 2]$ and analyzed the schedulability for T_i values that differ up to 1, 2, and 3 orders of magnitude, i.e., T_i in a range of $[1ms, 10ms]$, $[1ms, 100ms]$, or $[1ms, 1000ms]$. The test by Baruah and Fisher [9] is clearly outperformed by Theorem 4.6, Theorem 4.7, and Baker's test [3], which provide similar acceptance ratios. The test by Huang and Chen [31] outperforms those three tests and is worse than the test in Theorem 4.4 in these settings. However, there is no dominance relation between Theorem 4.4 and the test by Huang and Chen [31], as some task sets are schedulable under the test by Huang and Chen [31] but not schedulable under Theorem 4.4 and vise versa.

There are other configurations where the test by Huang and Chen [31] performs better than Theorem 4.4. One example is shown in Figure 4, analyzing the impact of the number of processors. Here Theorem 4.4 performs compatible to Theorem 4.6, Theorem 4.7, and Baker's test [3] for $M = 4$. When the number of processors increases, Theorem 4.4 performs better. The gap to Huang and Chen [31] is smaller for 8 processors and Theorem 4.4 has a higher acceptance rate when the utilization level is $80\% \times M$. For $M = 16$ processors Theorem 4.4 accepts more task sets than Huang and Chen [31] when the utilization level is

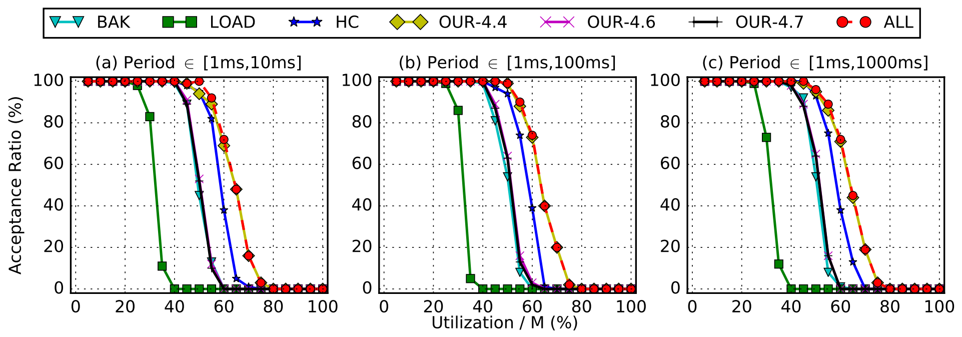


Figure 3 Comparison of the tests presented in Theorem 4.4, 4.6, and 4.7 with the methods from Baruah and Fisher (LOAD) [9], Baker [3], and Huang and Chen [31] for different ranges of period. The evaluation setup is the same as in [31], i.e., DM, $M = 8$, $N = 40$, $\frac{D_i}{T_i} \in [0.8, 2]$.

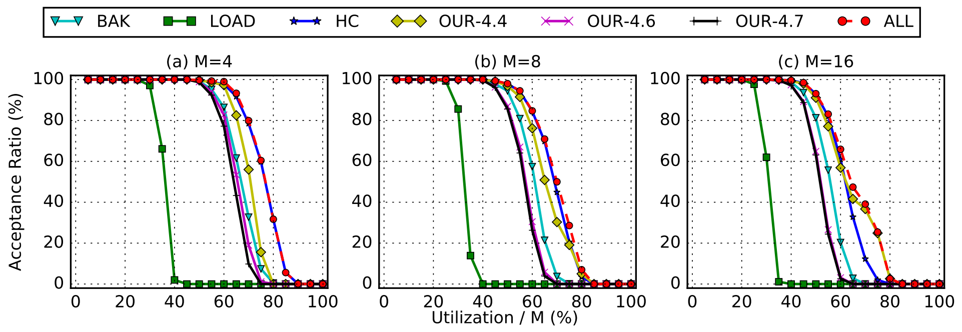


Figure 4 Comparison of the tests presented in Theorem 4.4, 4.6, and 4.7 with the methods from Baruah and Fisher (LOAD) [9], Baker [3], and Huang and Chen [31] for different M values. The other parameters are fixed, i.e., DM, $N = 5 \times M$, $T_i \in [1ms, 10ms]$, and $\frac{D_i}{T_i} \in [0.8, 10]$.

$\geq 65\% \times M$. In addition, it is possible that the number of task sets that is accepted by at least one algorithm is not close to the number of task sets accepted by Huang and Chen [31] or Theorem 4.4 as can be seen for the utilization level $75\% \times M$ in the case where $M = 8$.

Furthermore, we tracked if the test by Baker [3] accepted some task sets that were not accepted by Huang and Chen [31] or Theorem 4.4, which happened occasionally. Therefore, we conclude that there is no dominance relation between any of those three tests, i.e., Theorem 4.4, and the tests by Baker [3] and by Huang and Chen [31]. As these tests can all be implemented with polynomial-time complexity, all three should be applied.

7 Conclusion

We present a series of schedulability tests for multiprocessor systems under any given fixed-priority scheduling approach. Those schedulability tests have different tradeoffs between their accuracy and their time complexity. All those schedulability tests dominate the approach by Baruah and Fisher [9], both with respect to speedup bounds and schedulability analysis. Theorem 3.10 is the most powerful schedulability test in this paper. However, we do not reach any concrete implementation with affordable time complexity. In the future work, we will seek for efficient methods to implement the schedulability test in Theorem 3.10.

References

1 Björn Andersson. Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. In *Principles of Distributed Systems, 12th International Conference, OPODIS*, pages 73–88, 2008. doi:10.1007/978-3-540-92221-6_7.

2 Björn Andersson, Sanjoy K. Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 193–202, 2001. doi:10.1109/REAL.2001.990610.

3 Theodore P Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006. doi:10.1007/S11241-005-4686-1.

4 Theodore P. Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *Principles of Distributed Systems, 11th International Conference, OPODIS*, pages 62–75, 2007. doi:10.1007/978-3-540-77096-1_5.

5 Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128, 2007. doi:10.1109/RTSS.2007.35.

6 Sanjoy Baruah and Enrico Bini. Partitioned scheduling of sporadic task systems: an ILP-based approach. In *Proc. DASIP*, 2008.

7 Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010. doi:10.1007/s11241-010-9096-3.

8 Sanjoy K. Baruah and Nathan Fisher. Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, pages 204–216, 2007. doi:10.1007/978-3-540-77096-1_15.

9 Sanjoy K. Baruah and Nathan Fisher. Global fixed-priority scheduling of arbitrary-deadline sporadic task systems. In *Distributed Computing and Networking, 9th International Conference, ICDCN*, pages 215–226, 2008. doi:10.1007/978-3-540-77444-0_20.

10 Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *In Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, 1990. doi:10.1109/REAL.1990.128746.

11 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Principles of Distributed Systems, 9th International Conference, OPODIS*, pages 306–321, 2005. doi:10.1007/11795490_24.

12 Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Trans. Computers*, 53(11):1462–1473, 2004. doi:10.1109/TC.2004.103.

13 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.

14 Enrico Bini, Thi Huyen Chau Nguyen, Pascal Richard, and Sanjoy K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Trans. Computers*, 58(2):279–286, 2009. doi:10.1109/TC.2008.167.

15 Enrico Bini, Andrea Parri, and Giacomo Dossena. A quadratic-time response time upper bound with a tightness property. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 13–22, 2015. doi:10.1109/RTSS.2015.9.

16 Vincenzo Bonifaci, Ho-Leung Chan, Alberto Marchetti-Spaccamela, and Nicole Megow. Algorithms and complexity for periodic real-time scheduling. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1350–1359, 2010. doi:10.1137/1.9781611973075.109.

17 Jian-Jia Chen. Erratum: Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems, 2017. URL: `http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2016-chen-erratum-globalDM.pdf`.

18 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. In *Real-Time Systems Symposium, RTSS*, pages 351–362, 2016. `doi:10.1109/RTSS.2016.041`.

19 Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I. Davis. On the pitfalls of resource augmentation factors and utilization bounds in real-time scheduling. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 9:1–9:25, 2017. `doi:10.4230/LIPIcs.ECRTS.2017.9`.

20 Jian-Jia Chen, Georg von der Brüggen, and Niklas Ueter. Evaluation results: Push forward: Global fixed-priority scheduling of arbitrary-deadline sporadic task systems. URL: `http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/eval_push_forward.zip`.

21 Jian-Jia Chen, Georg von der Brüggen, and Niklas Ueter. Push forward: Global fixed-priority scheduling of arbitrary-deadline sporadic task systems, 2017. URL: `http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2018-chen-ecrts-push-forward-full.pdf`.

22 Robert I. Davis and Alan Burns. Response time upper bounds for fixed priority real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 407–418, Nov 2008. `doi:10.1109/RTSS.2008.18`.

23 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011. `doi:10.1145/1978802.1978814`.

24 Friedrich Eisenbrand and Thomas Rothvoß. Static-priority real-time scheduling: Response time computation is np-hard. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS*, pages 397–406, 2008. `doi:10.1109/RTSS.2008.25`.

25 Friedrich Eisenbrand and Thomas Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1029–1034, 2010. `doi:10.1137/1.9781611973075.83`.

26 Pontus Ekberg and Wang Yi. Uniprocessor feasibility of sporadic tasks remains conp-complete under bounded utilization. In *2015 IEEE Real-Time Systems Symposium, RTSS*, pages 87–95, 2015. `doi:10.1109/RTSS.2015.16`.

27 Pontus Ekberg and Wang Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly coNP-Complete. In *27th Euromicro Conference on Real-Time Systems, ECRTS*, pages 281–286, 2015. `doi:10.1109/ECRTS.2015.32`.

28 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

29 Gilles Geeraerts, Joël Goossens, and Markus Lindström. Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm. *Real-time systems*, 49(2):171–218, 2013. `doi:10.1007/s11241-012-9172-y`.

30 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *IEEE Real-Time Systems Symposium*, pages 387–397, 2009. `doi:10.1109/RTSS.2009.11`.

31 Wen-Hung Huang and Jian-Jia Chen. Response time bounds for sporadic arbitrary-deadline tasks under global fixed-priority scheduling on multiprocessors. In *International Conference on Real Time Networks and Systems, RTNS*, pages 215–224, 2015. `doi:10.1145/2834848.2834849`.

32 John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *proceedings Real-Time Systems Symposium (RTSS)*, pages 201–209, Dec 1990. doi:10.1109/REAL.1990.128748.

33 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

34 Lars Lundberg. Analyzing fixed-priority global multiprocessor scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–153, 2002. doi:10.1109/RTTAS.2002.1137389.

35 Mikael Sjodin and Hans Hansson. Improved response-time analysis calculations. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 399–408, 1998. doi:10.1109/REAL.1998.739773.

36 Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor global fixed-priority scheduling. *Real-Time Systems*, 52(3):323–355, 2016. doi:10.1007/s11241-015-9245-9.

37 Youcheng Sun, Giuseppe Lipari, Nan Guan, and Wang Yi. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–9, 2014. doi:10.1109/RTCSA.2014.6910543.

A Appendix: Additional Proofs

A.1 Proof of Theorem 5.4: Speedup lower bound of global DM for arbitrary-deadline task systems

We will specifically use the following task set $\mathbf{T}^{ad}$ with $N = 2M + 1$ tasks. Let ε be an arbitrarily small positive real number such that $1/\varepsilon$ is an integer. Let $\eta \ll \varepsilon$ be an arbitrarily small positive number, that is used to enforce the priority assignment under global DM:

- $C_i = \frac{\varepsilon}{3}$, $T_i = \varepsilon$, $D_i = 1$, for $i = 1, 2, \ldots, M$.
- $C_i = \frac{1}{3}$, $T_i = \infty$, $D_i = 1 + \eta$, for $i = M + 1, M + 2, \ldots, 2M$.
- $C_i = \frac{1+\varepsilon}{3}$, $T_i = \infty$, $D_i = 1 + 2\eta$, for $i = 2M + 1$

As the setting of $\eta \ll \varepsilon$ is just to enforce the indexing, *we will directly take* $\eta \to 0$ *here.*

▶ **Lemma A.1.** *$\mathbf{T}^{ad}$ is not schedulable by global DM.*

Proof. This can be proved by showing that task τ_N misses its deadline in the following concrete arrival pattern: all tasks release their first jobs at time 0 and the subsequent jobs arrive as early as possible while respecting their minimum inter-arrival times. For this arrival pattern, the jobs of tasks $\tau_1, \tau_2, \ldots, \tau_M$ are executed from time $i\varepsilon$ to time $i\varepsilon + \frac{\varepsilon}{3}$ for $i = 0, 1, 2, \ldots, 1/\varepsilon$. Therefore, these M tasks are executed for in total 1/3 time units from time 0 to time 1. For tasks $\tau_{1+M}, \tau_{2+M}, \ldots, \tau_{2M}$, each of them is executed for 1/3 time units from time 0 to time 1 when the processors do not execute $\tau_1, \tau_2, \ldots, \tau_M$. Task τ_{2M+1} is executed alone without any overlap with the executions of the higher-priority tasks. Therefore task τ_{2M+1} misses its deadline since it needs $\frac{1+\varepsilon}{3}$ time units, but only $\frac{1}{3}$ time units are available before its deadline. ◀

▶ **Lemma A.2.** *There exists a feasible schedule for task set $\mathbf{T}^{ad}$ at any speed no lower than $\frac{1+\varepsilon}{3} + \frac{1+\varepsilon}{3M}$.*

Proof. We apply multiprocessor semi-partitioned scheduling, in which tasks in $\{\tau_m, \tau_{m+M}\}$ are assigned to processor m for $m = 1, 2, \ldots, M$. In our designed semi-partitioned schedule, a job of task τ_{2M+1}, i.e., a part of τ_N, is executed partially on each of the M processors as follows: it runs on processor m for C_N/M amount of time, and then migrates to processor $m+1$ to continue its execution, for $m = 1, 2, \ldots, M-1$. To ensure that the migration can be served immediately, τ_N is given the the highest-priority in this schedule. Therefore, a *subtask* of task τ_N on processor m, denoted as $\tau_{N,m}$, has a relative deadline C_N/M. As long as the speed of the processors is greater than or equal to $\frac{1+\varepsilon}{3}$, task τ_N can meet its deadline. Therefore, in our designed semi-partitioned schedule, each processor m has a task set $\mathbf{T}_m$ that consists of three tasks: τ_m and τ_{m+M} from $\mathbf{T}^{ad}$ and a subtask $\tau_{N,m}$ of task τ_N with execution time C_N/M. We assign the second priority to task τ_{m+M} and the lowest priority to task τ_m on processor m.

We utilize the worst-case response time analysis by Bini et al. [14]. They showed that if $1 - \sum_{\tau_i \in hp(\tau_k, m)} U_i \leq 1$, then the worst-case response time of a task τ_k in a task set $\mathbf{T}_m$ under fixed-priority scheduling on a processor is at most

$$\frac{C_k + \sum_{\tau_i \in hp(\tau_k,m)} C_i - \sum_{\tau_i \in hp(\tau_k,m)} U_i C_i}{1 - \sum_{\tau_i \in hp(\tau_k,m)} U_i}, \tag{37}$$

where $hp(\tau_k, m)$ is the set of the tasks in $\mathbf{T}_m$ that have a higher priorities than task τ_k. Note that the precondition $1 - \sum_{\tau_i \in hp(\tau_k,m)} U_i \leq 1$ for the test in Eq. (37) to be applicable always holds at any arbitrarily speed since we assign τ_m as the lower-priority task on processor m and $U_{m+M} \to 0$, and $U_{N,m} = C_{N,m}/T_N \to 0$.

By Eq. (37), if the speed of processor m is greater than or equal to $\frac{C_N}{M} + C_{m+M} = \frac{1+\varepsilon}{3M} + \frac{1}{3}$, task τ_{m+M} can still meet its deadline in this schedule. By Eq. (37), task τ_m can meet its deadline at speed s in this schedule if

$$1 \geq \frac{C_m/s + \sum_{\tau_i \in hp(\tau_k,m)} C_i/s - \sum_{\tau_i \in hp(\tau_k,m)} \frac{U_i}{s}\frac{C_i}{s}}{1 - \sum_{\tau_i \in hp(\tau_k,m)} U_i/s} = \frac{\varepsilon}{3s} + \frac{1}{3s} + \frac{1+\varepsilon}{3sM} \tag{38}$$

Therefore, as long as $s \geq \frac{1+\varepsilon}{3} + \frac{1+\varepsilon}{3M}$, task τ_m meets its deadline under our designed schedule. ◀

A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling

Mitra Nasri
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
mitra@mpi-sws.org

Geoffrey Nelissen
CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Porto, Portugal
grrpn@isep.ipp.pt

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

An effective way to increase the timing predictability of multicore platforms is to use non-preemptive scheduling. It reduces preemption and job migration overheads, avoids intra-core cache interference, and improves the accuracy of worst-case execution time (WCET) estimates. However, existing schedulability tests for global non-preemptive multiprocessor scheduling are pessimistic, especially when applied to periodic workloads. This paper reduces this pessimism by introducing a new type of sufficient schedulability analysis that is based on an exploration of the space of possible schedules using concise abstractions and state-pruning techniques. Specifically, we analyze the schedulability of non-preemptive job sets (with bounded release jitter and execution time variation) scheduled by a global job-level fixed-priority (JLFP) scheduling algorithm upon an identical multicore platform. The analysis yields a lower bound on the best-case response-time (BCRT) and an upper bound on the worst-case response time (WCRT) of the jobs. In an empirical evaluation with randomly generated workloads, we show that the method scales to 30 tasks, a hundred thousand jobs (per hyperperiod), and up to 9 cores.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Real-time schedulability

Keywords and phrases global multiprocessor scheduling, schedulability analysis, non-preemptive tasks, worst-case response time, best-case response time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.9

Related Version An extended version of this paper is available as a technical report [19], `http://www.mpi-sws.org/tr/2018-003.pdf`.

Acknowledgements The first author is supported by a post-doc fellowship awarded by the Alexander von Humboldt Foundation. The second author was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234). The authors would like to thank the anonymous reviewers for their insightful comments and suggestions.

1 Introduction

While modern multicore platforms offer ample processing power and a compelling price/performance ratio, they also come with no small amount of architectural complexity. Unfortunately, this complexity—such as shared caches, memory controllers, and other shared micro-architectural resources—has proven to be a major source of execution-time unpredictability, and ultimately a fundamental obstacle to deployment in safety-critical systems.

In response, the research community has developed a number of innovative approaches for managing such challenging hardware platforms. One particularly promising approach explored in recent work [1, 15, 24] is to split each job into three distinct phases: **(i)** a dedicated *memory-load* or *prefetching* phase, which transfers all of a job's required memory from the shared main memory to a core-local private cache or scratchpad memory; followed by **(ii)** the actual *execution* phase, in which the job executes *non-preemptively* and in an isolated manner without interference from the memory hierarchy as all memory references are served from a fast, exclusive private memory, which greatly enhances execution-time predictability; and finally **(iii)** a *write-back* phase in which any modified data is flushed to main memory. As a result of the high degree of isolation restored by this approach [20], a more accurate *worst-case execution time* (WCET) analysis becomes possible since the complete mitigation of inter-core interference during the execution phase allows existing uniprocessor techniques [25] to be leveraged. Recent implementations of the idea, such as Tabish et al.'s scratchpad-centric OS [24], have shown the phased-execution approach to indeed hold great promise in practice.

From a scheduling point of view, however, the phased-execution approach poses a number of difficult challenges. As jobs must execute non-preemptively—otherwise prefetching becomes impractical and there would be only little benefit to predictability—the phased-execution approach fundamentally requires a *non-preemptive real-time multiprocessor scheduling problem* to be solved. In particular, Alhammad and Pellizzoni [1] and Maia et al. [15] considered the phase-execution model in the context of non-preemptive *global scheduling*, where pending jobs are allocated simply to the next available core in order of their priorities.

Crucially, to make schedulability guarantees, Alhammad and Pellizzoni [1] and Maia et al. [15] rely on existing state-of-the-art analyses of global non-preemptive scheduling as a foundation for their work. Unfortunately, as we show in Sec. 6, this analytical foundation—i.e., the leading schedulability tests for global non-preemptive scheduling [4, 10, 11, 13]—suffers from substantial pessimism, especially when applied to periodic hard real-time workloads.

To attack this analysis bottleneck, we introduce a new, much more accurate method for the schedulability analysis of *finite sets of non-preemptive jobs* under *global job-level fixed-priority* (JLFP) scheduling policies. Our method, which can be applied to *periodic real-time tasks* (and other recurrent task models with a repeating hyperperiod), is based on a novel state-space exploration approach that can scale to realistic system parameters and workload sizes. In particular, this work introduces a new abstraction for representing the space of possible non-preemptive multiprocessor schedules and explains how to explore this space in a practical amount of time with the help of novel state-pruning techniques.

Related work. Global non-preemptive multiprocessor scheduling has received much less attention to date than its preemptive counterpart. The first sufficient schedulability test for global non-preemptive scheduling was proposed by Baruah [4]. It considered sequential sporadic tasks scheduled with a non-preemptive *earliest-deadline-first* (G-NP-EDF) scheduling algorithm. Later, Guan et al. [10, 11] proposed three new tests; one generic schedulability test for any *work-conserving* global non-preemptive scheduling algorithm, and two response-time bounds for G-NP-EDF and global non-preemptive *fixed-priority* (G-NP-FP) scheduling. Recently, Lee et al. [13, 14] proposed a method to remove unnecessary carry-in workload from the total interference that a task suffers. These tests for sporadic tasks have been used in various contexts such as the schedulability analysis of periodic parallel tasks with non-preemptive sections [21] and systems with shared cache memories [26] or with transactional memories [1, 24]. However, these tests become needlessly pessimistic when applied to periodic tasks as they fail to discount many execution scenarios that are impossible in a periodic setting. Moreover, these tests do not account for any release jitter that may arise due to timer inaccuracy, interrupt latency, or networking delays.

To the best of our knowledge, no exact schedulability analysis for global job-level fixed-priority non-preemptive scheduling algorithms (including G-NP-EDF and G-NP-FP) either for sporadic or for periodic tasks has been proposed to date. The exact schedulability analysis of global *preemptive* scheduling for sporadic tasks has been considered in several works [3, 5, 6, 9, 23]. These analyses are mainly based on exploring all system states that can be possibly reached using model checking, timed automata, or linear-hybrid automata. These works are inherently designed for a preemptive execution model, where no lower-priority task can block a higher-priority one, and hence are not applicable to non-preemptive scheduling. The second limitation of the existing analyses is their limited scalability. They are affected by the number of tasks, processors, and the granularity of timing parameters such as periods. For example, the analysis of Sun et al. [23] can only handle up to 7 tasks and 4 cores, while the solution by Guan et al. [9] is applicable only if task periods lie between 8 and 20.

In our recent work [16], we have introduced an exact schedulability test based on a schedule-abstraction model for uni-processor systems executing non-preemptive job sets with bounded release jitter and execution time variation. By introducing an effective state-merging technique, we were able to scale the test to task sets with more than 30 tasks or about 100000 jobs in their hyperperiod for any job-level fixed-priority scheduling algorithm. The underlying model and the test's exploration rules, however, are designed for, and hence limited to, uniprocessor systems and cannot account for any scenarios that may arise when multiple cores execute jobs in parallel.

Contributions. In this paper, we introduce a sufficient schedulability analysis for global job-level fixed-priority scheduling algorithms considering a set of non-preemptive jobs with bounded release jitter and execution time variation. Our analysis derives a lower bound on the best-case response time (BCRT) and an upper bound on the worst-case response time (WCRT) of each job, taking into account all uncertainties in release and execution times. The proposed analysis is not limited to the analysis of periodic tasks (with or without release jitter), but can also analyze any system with a known job release pattern, e.g., bursty releases, multi-frame tasks, or any other application-specific workload that can be represented as a recurring set of jobs.

The analysis proceeds by exploring a graph, called *schedule-abstraction graph*, that contains all possible schedules that a given set of jobs may experience. To render such an exploration feasible, we aggregate all schedules that result in the same order of start times of the jobs and hence significantly reduce the search space of the analysis and makes it independent from the time granularity of the timing parameters of the systems. Moreover, we provide an efficient path-merging technique to collapse redundant states and avoid non-required state explorations. The paper presents an algorithm to explore the search space, derives merge rules, and establishes the soundness of the solution.

2 System Model and Definitions

We consider the problem of scheduling a finite set of non-preemptive *jobs* $\mathcal{J}$ on a multicore platform with m identical cores. Each job $J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i)$ has an earliest-release time r_i^{min} (a.k.a. *arrival time*), latest-release time r_i^{max}, *absolute* deadline d_i, best-case execution time (BCET) C_i^{min}, WCET C_i^{max}, and priority p_i. The priority of a job can be decided by the system designer at design time or by the system's JLFP scheduling algorithm. We assume that a numerically smaller value of p_i implies higher priority. Any ties in priority are broken by job ID. For ease of notation, we assume that the "$<$" operator implicitly reflects this tie-breaking rule. We use $\mathbb{N}$ to represent the natural numbers including 0. We assume a discrete-time model and all job timing parameters are in $\mathbb{N}$.

At runtime, each job is *released* at an *a priori* unknown time $r_i \in [r_i^{min}, r_i^{max}]$. We say that a job J_i is *possibly released* at time t if $t \geq r_i^{min}$, and *certainly released* if $t \geq r_i^{max}$. Such release jitter may arise due to timer inaccuracy, interrupt latency, or communication delays, e.g., when the task is activated after receiving data from the network. Similarly, each released job has an *a priori* unknown execution time requirement $C_i \in [C_i^{min}, C_i^{max}]$. Execution time variation occurs because of the use of caches, out-of-order-execution, input dependencies, program path diversity, state dependencies, etc. We assume that the absolute deadline of a job, i.e., d_i, is fixed *a priori* and not affected by release jitter. Released jobs remain pending until completed, i.e., there is no job-discarding policy.

Each job must execute sequentially, i.e., it cannot execute on more than one core at a time. Hence, because jobs are non-preemptive, a job J_i that starts its execution on a core at time t occupies that core during the interval $[t, t + C_i)$. In this case, we say that job J_i *finishes by* time $t + C_i$. At time $t + C_i$, the core used by J_i becomes available to start executing other jobs. A job's *response time* is defined as the length of the interval between the *arrival* and completion of the job [2], i.e., $t + C_i - r_i^{min}$. We say that a job is *ready* at time t if it is released and did not yet start its execution prior to time t.

In this paper, we assume that shared resources that must be accessed in mutual exclusion are protected by FIFO spin locks. Since we consider a non-preemptive execution model, it is easy to obtain a bound on the worst-case time that any job spends spinning while waiting to acquire a contested lock; we assume the worst-case spinning delay is included in the WCETs.

Throughout the paper, we use $\{\cdot\}$ to denote a set of items in which the order of elements is irrelevant and $\langle\cdot\rangle$ to denote an enumerated set of items. In the latter case, we assume that items are indexed in the order of their appearance in the sequence. For ease of notation, we use $\max_0\{X\}$ and $\min_\infty\{X\}$ over a set of positive values $X \subseteq \mathbb{N}$ that is completed by 0 and ∞, respectively. That is, if $X = \emptyset$, then $\max_0\{X\} = 0$ and $\min_\infty\{X\} = \infty$, otherwise they return the usual maximum and minimum values in X, respectively.

The schedulability analysis proposed in this paper can be applied to periodic tasks. A thorough discussion of how many jobs must be considered in the analysis for different types of tasks with release offset and constrained or arbitrary deadlines has been presented in [16].

We consider a non-preemptive global JLFP scheduler upon an identical multicore platform. The scheduler is invoked whenever a job is released or completed. In the interest of simplifying the presentation of the proposed analysis, we make the modeling assumption that, without loss of generality, at any invocation of the scheduler, at most one job is picked and assigned to a core. If two or more release or completion events occur at the same time, the scheduler is invoked once for each event. The actual scheduler implementation in the analyzed system need not adhere to this restriction and may process more than one event during a single invocation. Our analysis remains safe if the assumption is relaxed in this manner.

We allow for a non-deterministic core-selection policy when more than one core is available for executing a job, i.e., when a job is scheduled, it may be scheduled on any available core. The reason is that requiring a deterministic tie-breaker for core assignments would impose a large synchronization overhead, e.g., to rule out any race windows when the scheduler is invoked concurrently on different cores at virtually the same time, and hence no such rule is usually implemented in operating systems.

We say that a job set $\mathcal{J}$ is *schedulable* under a given scheduling policy if no execution scenario of $\mathcal{J}$ results in a deadline miss, where an execution scenario is defined as follows.

▶ **Definition 1.** An *execution scenario* $\gamma = \{(r_1, C_1), (r_2, C_2), \ldots, (r_n, C_n)\}$, where $n = |\mathcal{J}|$, is an assignment of execution times and release times to the jobs of $\mathcal{J}$ such that, for each job J_i, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$.

We exclusively focus on work-conserving, and priority-driven scheduling algorithms, i.e., the scheduler dispatches a job only if the job has the highest priority among all ready jobs, and it does not leave a core idle if there exists a ready job. We assume that the WCET of each job is padded to cover the scheduler overhead and to account for any micro-architectural interference (e.g., cache or memory bus interference).

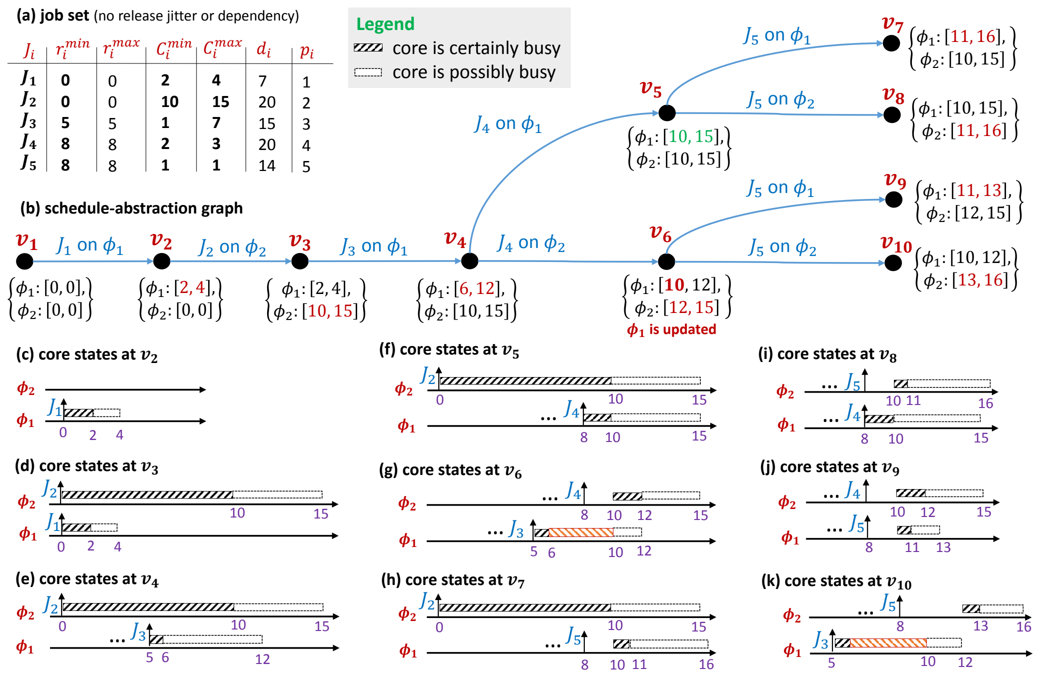
3 Schedule-Abstraction Graph

Our schedulability analysis derives a safe upper bound on the WCRT and a safe lower bound on the BCRT of each job by exploring a superset of all possible schedules. Since the number of schedules depends on the space of possible execution scenarios, which is a combination of release times and execution times of the jobs, it is intractable to naively enumerate all distinct schedules. To solve this problem, we aggregate schedules that lead to the *same ordering* of job start times (a.k.a. dispatch times) on the processing platforms. To this end, in the rest of this section, we introduce an abstraction of job orderings that encodes possible finish times of the jobs.

To represent possible job orderings we use an acyclic graph whose edges are labeled with jobs. Thus, each path in the graph represents a dispatch order of jobs in the system. Fig. 1-(b) shows an example of such a graph. For example, the path from v_1 to v_9 means that the jobs $\langle J_1, J_2, J_3, J_4, J_5 \rangle$ have been scheduled one after another. The length of a path P, denoted by $|P|$, is the number of jobs scheduled on that path.

To account for the uncertainties in the release times and execution times of jobs, which in turn result in different schedules, we use intervals to represent the state of a core. For example, assume that there is only one core in the system and consider a particular job J_i. Assume that the release interval and execution requirement of J_i are $[0, 5]$ and $[10, 15]$, respectively. In a job ordering where J_i is the first job dispatched on the core, the resulting core interval will become $[10, 20]$, where $10 = r_i^{min} + C_i^{min}$ and $20 = r_i^{max} + C_i^{max}$ are the *earliest finish time* (EFT) and *latest finish time* (LFT), respectively, of the job on the core. Here, the interval $[10, 20]$ means that the core will be *possibly available* at time 10 and will be *certainly available* at time 20. Equivalently, any time instant t in a core interval corresponds to an execution scenario in which the core is busy until t and becomes available at t.

Using the notion of core intervals, we define a system state as a set of m core intervals. System states are vertices of the graph and represent the states of the cores after a certain set of jobs has been scheduled in a given order.



(a) job set (no release jitter or dependency)

J_i	r_i^{min}	r_i^{max}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	**0**	0	**2**	**4**	7	1
J_2	**0**	0	**10**	**15**	20	2
J_3	**5**	5	**1**	**7**	15	3
J_4	**8**	8	**2**	**3**	20	4
J_5	**8**	8	**1**	**1**	14	5

Legend
▨ core is certainly busy
□ core is possibly busy
(b) schedule-abstraction graph
v_1 J_1 on ϕ_1 v_2 J_2 on ϕ_2 v_3 J_3 on ϕ_1 v_4 J_4 on ϕ_2 v_6 J_5 on ϕ_2 v_{10}
J_4 on ϕ_1 v_5 J_5 on ϕ_1 v_7
J_5 on ϕ_2 v_8
J_5 on ϕ_1 v_9
v_1: $\{\phi_1: [0,0], \phi_2: [0,0]\}$
v_2: $\{\phi_1: [2,4], \phi_2: [0,0]\}$
v_3: $\{\phi_1: [2,4], \phi_2: [10,15]\}$
v_4: $\{\phi_1: [6,12], \phi_2: [10,15]\}$
v_5: $\{\phi_1: [10,15], \phi_2: [10,15]\}$
v_6: $\{\phi_1: [10,12], \phi_2: [12,15]\}$
ϕ_1 is updated
v_7: $\{\phi_1: [11,16], \phi_2: [10,15]\}$
v_8: $\{\phi_1: [10,15], \phi_2: [11,16]\}$
v_9: $\{\phi_1: [11,13], \phi_2: [12,15]\}$
v_{10}: $\{\phi_1: [10,12], \phi_2: [13,16]\}$
(c) core states at v_2
(d) core states at v_3
(e) core states at v_4
(f) core states at v_5
(g) core states at v_6
(h) core states at v_7
(i) core states at v_8
(j) core states at v_9
(k) core states at v_{10}

Figure 1 A schedule-abstraction graph G for five jobs that are scheduled on two cores: **(a)** shows the job set information (jobs do not have release jitter), **(b)** shows the schedule-abstraction graph, **(c)** to **(k)** show the state of the two cores at system states v_2 to v_{10}, respectively.

3.1 Graph Definition

The schedule-abstraction graph is a directed acyclic graph $G = (V, E)$, where V is a set of system states and E is the set of labeled edges. A system state $v \in V$ is a multiset of m core intervals denoted by $\{\phi_1, \phi_2, \ldots, \phi_m\}$. A core interval $\phi_k = [EFT_k, LFT_k]$ is defined by the EFT and LFT of a job that is scheduled on the core, denoted by EFT_k and LFT_k, respectively. Equivalently, EFT_k is the time at which the core becomes *possibly available* and LFT_k is the time at which the core becomes *certainly available*. Since cores are identical, the schedule-abstraction graph does not distinguish between them and hence does not keep track of the physical core on which a job is executing.

The schedule-abstraction graph contains all possible orderings of job start times in any possible schedule. This ordering is represented by directed edges. Each edge $e = (v_p, v_q)$ from state v_p to state v_q has a label representing the job that is scheduled next after state v_p. The sequence of edges in a path P represents a possible sequence of scheduling decisions (i.e., a possible sequence of job start times) to reach the system state modeled by v_p from the initial state v_1.

3.2 Example

Fig. 1-(b) shows the schedule-abstraction graph that includes all possible start-time orders of the jobs defined in Fig. 1-(a) on a two-core processor. In the initial state v_1, no job is scheduled. At time 0, two jobs J_1 and J_2 are released. Since $p_1 < p_2$, the scheduler first schedules J_1 on one of the available cores. For the sake of clarity, we have numbered the cores in this example, however, they are identical from our model's perspective.

Fig. 1-(c) shows the state of both cores after job J_1 is scheduled. The dashed rectangle that covers the interval $[0, 2)$ shows the time during which the core is certainly not available for other jobs since $C_1^{min} = 2$. In this state, the EFT of ϕ_1 is 2 and its LFT is 4, as shown by the white rectangle, i.e., ϕ_1 may possibly become available at time 2 and will certainly be available at time 4. From the system state v_2, only v_3 is reachable. The transition between these two states indicates that job J_2 is scheduled on the available core ϕ_2 starting at time 0.

As shown in Fig. 1-(d), core ϕ_1 is certainly available from time 4. Thus, when job J_3 is released at time 5, the scheduler has no other choice but to schedule job J_3 on this core. The label of this transition shows that J_3 has been scheduled.

From system state v_4, two other states are reachable depending on the finish times of jobs J_2 and J_3.

State v_5. If core ϕ_1 becomes available before core ϕ_2, then J_4 can start its execution on ϕ_1. This results in state v_5 (Fig. 1-(f)). The core intervals of v_5 are obtained as follows. According to the intervals of v_4, the earliest time at which ϕ_1 becomes available is 6, while the release time of J_4 is 8, thus, the earliest start time of J_4 on core ϕ_1 is 8, which means that its earliest finish time is 10. The latest start time of J_4 such that it is still scheduled on core ϕ_1 is time 12. The reason is that J_4 is released at time 8 and hence is pending from that time onward. However, it cannot be scheduled until a core becomes available. The earliest time a core among ϕ_1 and ϕ_2 becomes available is at time 12 (which is the latest finish time of J_3). Since the scheduling algorithm is work-conserving, it will certainly schedule job J_4 at 12 on the core that has become available. Consequently, the latest finish time of J_4 is $12 + 3 = 15$.

State v_6. In state v_4, if core ϕ_2 becomes available before ϕ_1, then job J_4 can be scheduled on ϕ_2 and create state v_6 (Fig. 1-(g)). In this case, the earliest start time of J_4 is at time 10 because, although it has been released before, it must wait until core ϕ_2 becomes available, which happens only at time 10. As a result, the earliest finish time of J_4 will be time $10 + 2 = 12$. On the other hand, the latest start time of J_4 such that it is scheduled on core ϕ_2 is 12 because at this time, job J_4 is ready and a core (ϕ_1) becomes available. Thus, if J_4 is going to be scheduled on ϕ_2, core ϕ_2 must become available by time 12. Note that since our core-selection policy is non-deterministic, if ϕ_2 becomes available at time 12, J_4 may be dispatched on either core. Consequently, the latest finish time of J_4 when scheduled on ϕ_2 is $12+3 = 15$. Furthermore, system state v_6 may arise only if core ϕ_1 has not become available before time 10, as otherwise job J_4 will be scheduled on ϕ_1 and create state v_5. Thus, state v_6 can be reached only if ϕ_1 does not become available before time 10. To reflect this constraint, the core interval of ϕ_1 must be updated to $[10, 12]$. The red dashed rectangle in Fig. 1-(g) illustrates this update. According to the schedule-abstraction graph in Fig. 1-(b), there exist three scenarios in which J_5 finishes at time 16 and hence misses its deadline. These scenarios are shown in Figs. 1-(h), (i) and (k), and are reflected in states v_7, v_8, and v_{10}, respectively.

4 Schedulability Analysis

This section explains how to build the schedule-abstraction graph. Sec. 4.1 presents the high-level description of our search algorithm, which consists of alternating *expansion*, *fast-forward*, and *merge* phases. These phases will be discussed in details in Sec. 4.2, 4.3, and 4.4, respectively. Sec. 5 provides a proof of correctness of the proposed algorithm.

4.1 Graph-Generation Algorithm

During the expansion phase, (one of) the shortest path(s) P in the graph from the root to a leaf vertex v_p is expanded by considering all jobs that can possibly be chosen by the JLFP scheduler to be executed next in the job execution sequence represented by P. For each such job, the algorithm checks on which core(s) it may execute. Finally, for each core on which the job may execute, a new vertex v'_p is created and added to the graph, and connected via an edge directed from v_p to v'_p.

After generating a new vertex v'_p, the fast-forward phase advances time until the next scheduling event. It accordingly updates the system state represented by v'_p.

The merge phase attempts to moderate the growth of the graph. To this end, the terminal vertices of paths that have the same set of scheduled jobs (but not necessarily in the same order) and core states that will lead to similar future scheduling decisions by the scheduler, are merged into a single state whose future states cover the set of all future states of the merged states. The fast-forward and merge phases are essential to avoid redundant work, i.e., to recognize that two or more states are similar early on before they are expanded. The algorithm terminates when there is no vertex left to expand, that is, when all paths in the graph represent a valid schedule of all jobs in $\mathcal{J}$.

Algorithm 1 presents our iterative method to generate the schedule-abstraction graph in full detail. A set of variables keeping track of a lower bound on the BCRT and an upper bound on the WCRT of each job is initialized at line 1. These bounds are updated whenever a job J_i can possibly be scheduled on any of the cores. The graph is initialized at line 2 with a root vertex v_1. The expansion phase corresponds to lines 6–21; line 13 implements the fast-forward, and lines 14–18 realize the merge phase. These phases repeat until every path in the graph contains $|\mathcal{J}|$ distinct jobs. We next discuss each phase in detail.

4.2 Expansion Phase

Assume that P is a path connecting the initial state v_1 to v_p. The sequence of edges in P represents a sequence of scheduling decisions (i.e., a possible sequence of job executions) to reach the system state modeled by v_p from the initial state v_1. We denote by $\mathcal{J}^P$the set of jobs scheduled in path P. To expand path P, Algorithm 1 evaluates for each job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ that was not scheduled yet whether it may be the next job picked by the scheduler and scheduled on any of the cores. For any job J_i that can possibly be scheduled on a core $\phi_k \in v_p$ *before any other job starts executing*, a new vertex v'_p is added to the graph (see lines 6–12 of Algorithm 1).

To evaluate if job J_i is a potential candidate for being started next in the dispatch sequence represented by P, we need to know:

1. The earliest time at which J_i may start to execute on core ϕ_k when the system is in the state described by vertex v_p. We call that instant the *earliest start time* (EST) of J_i on core ϕ_k, and we denote it by $EST_{i,k}(v_p)$.
2. The time by which J_i must have certainly started executing if it is to be the *next job* to be scheduled by the JLFP scheduler on the processing platform. This second time instant is referred to as the *latest start time* (LST) of J_i and is denoted by $LST_i(v_p)$.

$LST_i(v_p)$ represents the latest time at which a work-conserving JLFP scheduler schedules J_i next after state v_p. Note that $LST_i(v_p)$ is a global value for the platform when it is in state v_p, while $EST_{i,k}(v_p)$ is related to a specific core ϕ_k.

Algorithm 1: Schedule Graph Construction Algorithm.

```
Input   : Job set $\mathcal{J}$
Output : Schedule graph $G = (V, E)$
1  $\forall J_i \in \mathcal{J}, BCRT_i \leftarrow \infty, WCRT_i \leftarrow 0$;
2  Initialize $G$ by adding a root vertex $v_1 = \{[0,0], [0,0], \ldots, [0,0]\}$, where $|v_1| = m$;
3  while $\exists$ a path $P$ from $v_1$ to a leaf vertex $v_p$ s.th. $|P| < |\mathcal{J}|$ do
4      $P \leftarrow$ a path from $v_1$ to a leaf with the least number of edges in the graph;
5      $v_p \leftarrow$ the leaf vertex of $P$;
6      for each job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ do
7          for each core $\phi_k \in v_p$ do
8              if $J_i$ can be dispatched on core $\phi_k$ according to (1) then
9                  Build $v'_p$ using (10);
10                 $BCRT_i \leftarrow \min\{EFT'_k - r_i^{min}, BCRT_i\}$;
11                 $WCRT_i \leftarrow \max\{LFT'_k - r_i^{min}, WCRT_i\}$;
12                 Connect $v_p$ to $v'_p$ by an edge with label $J_i$;
13                 Fast-forward $v'_p$ according to (13);
14                 while $\exists$ path $Q$ that ends to $v_q$ such that the condition defined in
                    Definition 4 is satisfied for $v'_p$ and $v_q$ do
15                     Update $v'_p$ using Algorithm 2;
16                     Redirect all incoming edges of $v_q$ to $v'_p$;
17                     Remove $v_q$ from $V$;
18                 end
19             end
20         end
21     end
22 end
```

A job J_i can be the next job scheduled in the job sequence represented by P if there is a core ϕ_k for which the earliest start time $EST_{i,k}(v_p)$ of J_i on ϕ_k is not later than the latest time at which this job must have started executing, i.e., before $LST_i(v_p)$ (see Lemma 7 in Sec. 5 for a formal proof). That is, J_i may commence execution on ϕ_k only if

$$EST_{i,k}(v_p) \leq LST_i(v_p). \tag{1}$$

For each core ϕ_k that satisfies (1), a new vertex v'_p is created, where v'_p represents the state of the system after dispatching job J_i on core ϕ_k.

Below, we explain how to compute $EST_{i,k}(v_p)$ and $LST_i(v_p)$. Then we describe how to build a new vertex v'_p for each core ϕ_k and job J_i that satisfies (1). Finally, we explain how the BCRT and WCRT of job J_i are updated according to its $EST_{i,k}(v_p)$ and $LST_i(v_p)$, respectively. To ease readability, from here on we will not specify any more that ϕ_k, $EST_{i,k}(v_p)$ and $LST_i(v_p)$ are related to a specific vertex v_p when it is clear from context, and will instead use the short-hand notations $EST_{i,k}$ and LST_i.

Earliest start time. To start executing on a core ϕ_k, a job J_i has to be released and ϕ_k has to be available. Thus, the earliest start time $EST_{i,k}$ of a job J_i on a core ϕ_k is given by

$$EST_{i,k} = \max\{r_i^{min}, EFT_k\}, \tag{2}$$

where r_i^{min} is the earliest time at which J_i may be released and EFT_k is the earliest time at which ϕ_k may become available.

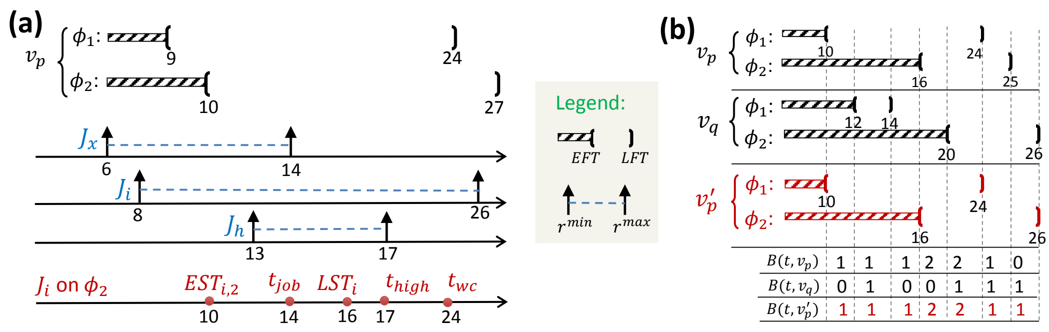

(a)
v_p
ϕ_1: 9 24
ϕ_2: 10 27
J_x 6 14
J_i 8 26
J_h 13 17
J_i on ϕ_2
$EST_{i,2}$ t_{job} LST_i t_{high} t_{wc}
10 14 16 17 24
Legend:
EFT LFT
r^{min} r^{max}
(b)
v_p ϕ_1: 10 24 ϕ_2: 16 25
v_q ϕ_1: 12 14 ϕ_2: 20 26
v'_p ϕ_1: 10 24 ϕ_2: 16 26
$B(t,v_p)$ 1 1 1 2 2 1 0
$B(t,v_q)$ 0 1 0 0 1 1 1
$B(t,v'_p)$ 1 1 1 2 2 1 1

Figure 2 **(a)** Expansion scenario for J_i and ϕ_2, where $p_h < p_i < p_x$. **(b)** An example merge.

Latest start time. Because we assume a work-conserving JLFP scheduling algorithm, two conditions must hold for job J_i be the *next job* scheduled on the processing platform: **(i)** J_i must be the highest-priority ready job (because of the JLFP assumption), and **(ii)** for every job J_j released before J_i, either J_j was already scheduled earlier on path P (i.e., $J_j \in \mathcal{J}^P$), or all cores were busy from the release of J_j until the release of J_i.

If (i) is not satisfied, then a higher-priority ready job is scheduled instead of J_i. Therefore the latest start time LST_i of J_i must be earlier than the earliest time at which a not-yet-scheduled higher-priority job is certainly released, that is, $LST_i < t_{high}$, where

$$t_{high} = \min_{\infty}\{r_x^{max} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P \ \wedge \ p_x < p_i\}. \tag{3}$$

If (ii) is not satisfied, then an earlier released job J_j will start executing on an idle core before J_i is released. Therefore the latest start time LST_i of J_i cannot be later than the earliest time at which both a core is certainly idle and a not-yet-scheduled job is certainly released. Formally, $LST_i \leq t_{wc}$, where

$$t_{wc} \triangleq \max\{t_{core}, t_{job}\}, \tag{4}$$

$$t_{core} \triangleq \min\{LFT_x \mid 1 \leq x \leq m\}, \text{ and} \tag{5}$$

$$t_{job} \triangleq \min_{\infty}\{r_y^{max} \mid J_y \in \mathcal{J} \setminus \mathcal{J}^P\}. \tag{6}$$

In the equations above, t_{core} is the earliest time at which a core is certainly idle and t_{job} is the earliest time at which a not-yet-scheduled job is certainly released.

Combining $LST_i < t_{high}$ and $LST_i \leq t_{wc}$, we observe that J_i must start by time

$$LST_i = \min\{t_{wc},\ t_{high} - 1\}. \tag{7}$$

▶ **Example 2.** Fig. 2-(a) shows how $EST_{i,k}$ and LST_i are calculated when job J_i is scheduled on core ϕ_2. In this example, $t_{job} = 14$ since job J_x becomes certainly available at that time. However, the earliest time at which a core (in this case, core ϕ_1) becomes available is $t_{core} = 24$, thus, $t_{wc} = 24$. On the other hand, the earliest time at which a job with a higher priority than J_i is certainly released is $t_{high} = 17$. Thus, $LST_i = t_{high} - 1 = 16$.

Building a new system state. If Inequality (1) holds, it is possible that job J_i is the next successor of path P and is scheduled on core ϕ_k at any $t \in [EST_{i,k}, LST_i]$ (Lemma 7 in Sec. 5 proves this claim). Our goal is to generate a single new vertex for the schedule-abstraction graph that aggregates all these execution scenarios.

Let v'_p denote the vertex that represents the new system state resulting from the execution of job J_i on core ϕ_k. The earliest and latest times at which ϕ_k may become available after executing job J_i is obtained as follows:

$$EFT'_k = EST_{i,k} + C_i^{min} \quad \text{and} \quad LFT'_k = LST_i + C_i^{max}. \tag{8}$$

Furthermore, because the latest scheduling event in the system state v'_p occurs no earlier than $EST_{i,k}$, no other job in $\mathcal{J} \setminus \mathcal{J}^P$ may possibly be scheduled before $EST_{i,k}$.

▶ **Property 3.** *If job J_i is the next job scheduled on the platform, and if it is scheduled on core ϕ_k, then no job $\in \mathcal{J} \setminus \mathcal{J}^P$ starts executing on any core $\phi_x, 1 \leq x \leq m$ before $EST_{i,k}$.*

Proof. By contradiction. Assume a job $J_j \in \mathcal{J} \setminus \mathcal{J}^P$ starts executing on a core ϕ_x before $EST_{i,k}$. Because J_i cannot start executing on ϕ_k before $EST_{i,k}$, J_j must be different from J_i and hence J_j starts to execute before J_i. That contradicts the assumption that J_i is the first job in $\mathcal{J} \setminus \mathcal{J}^P$ to be scheduled on the platform. ◀

To ensure that Property 3 is correctly enforced in the new system state represented by v'_p, we update the core intervals in state v'_p as follows

$$\phi'_x \triangleq \begin{cases} [EFT'_k, LFT'_k] & \text{if } x = k, \\ [EST_{i,k}, EST_{i,k}] & \text{if } x \neq k \ \wedge \ LFT_x \leq EST_{i,k}, \\ [\max\{EST_{i,k}, EFT_x\}, LFT_x] & \text{otherwise.} \end{cases} \tag{9}$$

The first case of (9) simply repeats (8) for job J_i. The second and third cases ensure that no job in $\mathcal{J} \setminus \mathcal{J}^P$ can be scheduled on those cores before $EST_{i,k}$. This is done by forcing ϕ_x's earliest availability time to be equal to $EST_{i,k}$. Finally, for cores that would certainly be idle after $EST_{i,k}$ (i.e., the second case in (9)), we set LFT_k (i.e., the time at which it becomes certainly available) to $EST_{i,k}$.

Finally, the new vertex v'_p is generated by applying (9) on all cores, i.e.,

$$v'_p = \{\phi'_1, \phi'_2, \ldots, \phi'_m\}. \tag{10}$$

Deriving the BCRT and WCRT of the jobs. Recall that the BCRT and the WCRT of a job are relative to its *arrival time*, i.e., r_i^{min}, and not its actual *release time*, which can be any time between r_i^{min} and r_i^{max}. In other words, release jitter counts towards a job's response time. As stated earlier, the earliest finish time of J_i on core ϕ_k cannot be smaller than EFT'_k and the latest finish time of J_i on core ϕ_k cannot be larger than LFT'_k (obtained from (8)). Using these two values, the BCRT and WCRT of job J_i are updated at lines 10 and 11 of Algorithm 1 as follows.

$$BCRT_i \leftarrow \min\{EFT'_k - r_i^{min}, BCRT_i\} \tag{11}$$
$$WCRT_i \leftarrow \max\{LFT'_k - r_i^{min}, WCRT_i\} \tag{12}$$

If the algorithm terminates, then $WCRT_i$ and $BCRT_i$ contain an upper bound on the WCRT and a lower bound on the BCRT of job J_i, respectively, over all paths. Since the graph considers all possible execution scenarios of $\mathcal{J}$, it considers all possible schedules of J_i. The resulting WCRT and BCRT estimates are therefore safe bounds on the actual WCRT and BCRT of the job, respectively. This property is proven in Corollary 18 in Sec. 5.

The quality of service of many real-time systems depends on both the WCRT and response-time jitter [7] of each task, i.e., the difference between the BCRT and WCRT of that

Algorithm 2: Algorithm that merges v_p and v_q, and creates v'_p.

```
1 Sort and re-index the core intervals $\phi_k(v_p)$ of $v_p$ in a non-decreasing order of their
    EFTs, such that $EFT_1(v_p) \leq EFT_2(v_p) \leq \dots EFT_m(v_p)$;
2 Sort and re-index $v_q$'s core intervals in a non-decreasing order of their EFTs such that
    $EFT_1(v_q) \leq EFT_2(v_q) \leq \dots EFT_m(v_q)$;
3 Pair each two core intervals $\phi_x(v_p)$ and $\phi_x(v_q)$ to create
    $\phi_x(v'_p) \triangleq [\min\{EFT_x(v_p), EFT_x(v_q)\}, \max\{LFT_x(v_p), LFT_x(v_q)\}]$;
```

task. One of the advantages of our schedule-abstraction graph is that it not only provides a way to compute those quantities, but also allows to extract the maximum variation between the response times of successive jobs released by the same task, hence allowing a more accurate analysis of (for instance) sampling jitter in control systems.

4.3 Fast-Forward Phase

As shown in lines 6 and 7, one new state will be added to the graph for each not-yet-scheduled job that can be scheduled next on one of the cores. This situation can lead to an explosion in the search space if the number of states is not reduced. In this work, we merge states to avoid redundant future explorations. To aid the subseqent merge phase, the fast-forward phase advances the time until a job may be released. We denote that instant by $t_{min} \triangleq \min_\infty \left\{ r_x^{min} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P \setminus \{J_i\} \right\}$. The fast-forward phase thus updates each core interval $\phi'_x \in v'_p$ as follows:

$$\phi'_x = \begin{cases} [t_{min}, t_{min}] & LFT_x \leq t_{min}, \\ [\max\{t_{min}, EFT_x\}, LFT_x] & \text{otherwise.} \end{cases} \tag{13}$$

The first case of (13) relies on the fact that from LFT'_x onward (i.e., the time at which a core ϕ'_x becomes certainly available), ϕ'_x remains available until a new job is scheduled on it. Since the earliest time at which a job can be scheduled is t_{min}, this core remains available at least until t_{min}. Thus, it is safe to update its interval to $[t_{min}, t_{min}]$, which denotes that the core is certainly free by t_{min}. Similarly, the second case of (13) is based on the fact that a core ϕ_x that is possibly available at EFT'_x remains possibly available either until reaching LFT'_x (where it certainly becomes free) or until a job may be scheduled on ϕ_x, which does not happen until t_{min} at the earliest. Lemma 9 in Sec. 5 proves that fast-forwarding state v'_p will not change any of the future states that can be reached from v'_p before applying (13).

4.4 Merge Phase

The merge phase seeks to collapse states to avoid redundant future explorations. The goal is to reduce the size of the search space such that the computed BCRT of any job may never become larger, the computed WCRT of any job may never become smaller, and all job scheduling sequences that were possible before merging states are still considered after merging those states. The merge phase is implemented in lines 14–18 of Algorithm 1, where the condition defined below in Definition 4 is evaluated for paths with length $|P| + 1$.

Since each state consists of exactly m core intervals, merging two states requires finding a matching among the two sets of intervals to merge individual intervals. Let states v_p and v_q be the end vertices of two paths P and Q. In order to merge v_p and v_q into a new state v'_p, we apply Algorithm 2. Next, we establish our merging rules, which will be proven to be safe in Corollary 15 in Sec. 5.

▶ **Definition 4.** Two states v_p and v_q can be merged if **(i)** $\mathcal{J}^P = \mathcal{J}^Q$, **(ii)** $\forall\ \phi_i(v_p), \phi_i(v_q)$, $\max\{EFT_i(v_p), EFT_i(v_q)\} \le \min\{LFT_i(v_p), LFT_i(v_q)\}$, and **(iii)** at any time t, the number of possibly-available cores in the merged state must be equal to the number of possibly-available cores in v_p or v_q, i.e.,

$$\forall t \in T, B(t, v'_p) = B(t, v_p) \ \vee \ B(t, v'_p) = B(t, v_q), \tag{14}$$

where $B(t, v_x)$ counts the number of core intervals of a state v_x that contain t, i.e.,

$$B(t, v_x) = \left|\left\{\phi_y(v_x) \mid t \in \big[EFT_y(v_x), LFT_y(v_x)\big]\right\}\right|, \tag{15}$$

and where T is the set of time instants at which the value of $B(\cdot)$ may change, i.e.,

$$T = \{EFT_x(v_p)|\ \forall x\} \ \cup \ \{LFT_x(v_p)|\ \forall x\} \ \cup \ \{EFT_x(v_q)|\ \forall x\} \ \cup \ \{LFT_x(v_q)|\ \forall x\}. \tag{16}$$

▶ **Example 5.** Fig. 2-(b) shows two states v_p and v_q that are merged to create state v'_p. As shown, for any $t \in T$, $B(t, v'_p)$ is equal to $B(t, v_p)$ or $B(t, v_q)$.

Notably, any merge rule that respects condition (i) in Definition 4 is *safe* (see Corollary 1 in Sec. 5.3). The role of conditions (ii) and (iii) is to trade-off between the accuracy and performance of the analysis by evading the inclusion of impossible execution scenarios in the resulting state. We leave the investigation of more accurate (or more eager) merging conditions, as well as the applicability of abstraction-refinement techniques, to future work.

5 Correctness of the Proposed Solution

In this section, we show that the schedule-abstraction graph constructed by Algorithm 1 correctly includes all job schedules that can arise from any possible execution scenario, i.e., for any possible execution scenario, there exists a path in the graph that represents the schedule of those jobs in that execution scenario (Theorem 17). The proof has two main steps: we first assume that the fast-forward and merge phases are not executed and show that the EFT and LFT of a job obtained from Equation (8) are correct lower and upper bounds on the finish time of a job scheduled on a core (Lemma 6) and that for an arbitrary vertex v_p, Inequality (1) is a necessary condition for a job to be scheduled next on core ϕ_k (Lemma 7). From these lemmas, we conclude that without fast-forwarding and merging, for any execution scenario there exists a path in the schedule graph that represents the schedule of the jobs in that execution scenario (Lemma 8).

In the second step, we show that the fast-forward and merge phases are *safe*, i.e., these phases will not remove any potentially reachable state from the original graph (Lemma 9 and Corollary 16). Finally, we establish that Algorithm 1 correctly derives an upper bound on the WCRT and a lower bound on the BCRT of every job (Corollary 18).

5.1 Soundness of the Expansion Phase

In this section, we assume that neither the fast-forward nor the merge phase is executed.

▶ **Lemma 6.** *For any vertex $v_p \in V$ and any successor v'_p of v_p such that job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ is scheduled on core ϕ_k between v_p and v'_p, $EFT_k(v'_p)$ and $LFT_k(v'_p)$ (as computed by* (8)*) are a lower bound and an upper bound, respectively, on the completion time of J_i.*

Proof. If neither the fast-forward nor the merge phases are executed, (9) is the only equation used to build a new state v'_p. In this lemma, we first prove that the EST and LST of the job obtained from (2) and (7) are a lower and an upper bound on the start time of job J_i on ϕ_k after the scheduling sequence represented by P. Then, we conclude that $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finish time of J_i on ϕ_k. The proof is by induction.

Base case. The base case is for any vertex v'_p that succeeds to the root vertex v_1 where all cores are idle. Hence in v'_p, job J_i is scheduled on one of the idle cores, say ϕ_k. Since all cores are idle at time 0, Equation (2) yields $EST_{i,k}(v_1) = r_i^{min}$, which is by definition the earliest time at which job J_i may start. Consequently, the earliest finish time of J_i cannot be smaller than $EFT_k(v'_p) = r_i^{min} + C_i^{min}$.
Similarly, (7) yields $LST_i(v_1) = \min\{t_{high} - 1, t_{job}\}$ (recall that $t_{core} = 0$ since all cores are idle in v_1). J_i cannot start later than $LST_i(v_1) = t_{job}$ if it is the first scheduled job as all cores are idle and hence as soon as a job is certainly released, it will be scheduled right away on one of the idle cores. Similarly, J_i cannot start its execution if it is not the highest-priority job anymore, i.e., at or after time t_{high}. As a result, the latest finish time of J_i cannot be larger than $LFT_k(v'_p) = \min\{t_{job}, t_{high} - 1\} + C_i^{max}$. Therefore, $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finishing time of J_i on ϕ_k after the scheduling sequence $P = \langle v_1, v'_p \rangle$.
For all other cores ϕ_x such that $x \neq k$, (9) enforces that $EFT_x(v'_p) = LFT_x(v'_p) = EST_{i,k}(v_1) = r_i^{min}$ (recall that $EFT_k(v_1) = LFT_k(v_1) = 0$), which is indeed the earliest time at which any job may start on ϕ_x if J_i is the first job executing on the platform and J_i is not released before r_i^{min}.

Induction step. Assume now that each core interval on every vertex from v_1 to v_p along path P provides a lower bound and an upper bound on the time at which that core will possibly and certainly be available, respectively, to start executing a new job. We show that in the new vertex v'_p obtained from scheduling job J_i on core ϕ_k after P, (8) provides a safe lower and upper bound on the finish time of J_i, and for other cores, the new core intervals computed by (9) are safe, i.e., no new job can start its execution on a core ϕ_x before EFT_x and the core cannot remain busy after LFT_x.

EFT. The earliest start time of J_i on core ϕ_k, i.e., $EST_{i,k}(v_p)$, cannot be smaller than $EFT_k(v_p)$ since, by the induction hypothesis, $EFT_k(v_p)$ is the earliest time at which core ϕ_k may start executing a new job. Moreover, a lower bound on $EST_{i,k}(v_p)$ is given by r_i^{min}, because J_i cannot execute before it is released. This proves (2) for ϕ_k. Further, if J_i starts its execution at $EST_{i,k}(v_p)$, it cannot finish before $EST_{i,k}(v_p) + C_i^{min}$ since its minimum execution time is C_i^{min}. Thus, the EFT of job J_i on ϕ_k in system state v'_p cannot be smaller than $EST_{i,k}(v_p) + C_i^{min}$, which proves the correctness of (8) for $EFT_k(v'_p)$.
The EFTs of all other cores ϕ_x in v'_p cannot be smaller than $EFT_x(v_p)$ in state v_p since no new job is scheduled on them. Furthermore, according to Property 3, job J_i can be scheduled on core ϕ_k (instead of any other core) only if no other job in $\mathcal{J} \setminus \mathcal{J}^P$ has started executing on any other core than ϕ_k until $EST_{i,k}(v_p)$. Hence, $\max\{EST_{i,k}(v_p), EFT_x(v_p)\}$ is a safe lower bound on the EST of a job in state v'_p (as computed by (9)).

LFT. Next, we show that $LST_i(v_p)$ cannot exceed $t_{high} - 1$ or t_{wc} as stated by (7). First, consider t_{high} and suppose $t_{high} \neq \infty$ (otherwise the claim is trivial). Since a higher-priority job is certainly released at the latest at time t_{high}, job J_i is no longer the highest-priority job at time t_{high}. Consequently, it cannot commence execution under a JLFP scheduler at or after time t_{high} if it is to be the next job scheduled after P. Hence, job J_i will be a direct successor of path P *only if* its execution starts no later than time $t_{high} - 1$. Now, consider t_{wc}. At time t_{wc}, a not-yet-scheduled job is certainly released and a core is certainly available. Hence a work-conserving scheduler will schedule that job at t_{wc}, thus, job J_i will be a direct successor of path P *only if* its execution starts no later than time t_{wc}. Since $LST_i(v_p)$ is the upper bound on the time at which job J_i can start its execution while being the next job scheduled after path P, the latest finish time of J_i on core ϕ_k cannot be larger than $\min\{t_{high} - 1, t_{wc}\} + C_i^{max}$, which proves the correctness of (8) for $LFT_k(v'_p)$.

Since in state v'_p job J_i is scheduled on core ϕ_k other cores cannot be available before $EST_{i,k}$, otherwise a work-conserving scheduler would schedule J_i on one of those cores instead of on ϕ_k. Equation (9) ensures that if J_i is the next job to be scheduled and if ϕ_k is the core on which J_i is scheduled, no other core will *certainly* be available by $EST_{i,k}(v_p)$, i.e., $EFT_x(v'_p) \geq EST_{i,k}(v_p)$.

By induction on all vertices in V, we have that $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finish time of any job scheduled between any two states v_p and v'_p, including J_i. ◀

▶ **Lemma 7.** *Job J_i can be scheduled next on core ϕ_k after jobs in path P only if* (1) *holds.*

Proof. If job J_i is released at time r_i^{min} and the core ϕ_k becomes available at EFT_k, then it can be dispatched no earlier than at time $EST_{i,k} = \max\{r_i^{min}, EFT_k\}$. If (1) does not hold, then t_{high} or t_{wc} (or both) are smaller than $EST_{i,k}$. This implies that either a higher-priority job other than job J_i is certainly released before $EST_{i,k}$ or a job other than J_i is certainly released before $EST_{i,k}$ and a core is certainly available before $EST_{i,k}$. In both cases, a work-conserving JLFP scheduling algorithm will not schedule job J_i until that other job is scheduled. Consequently, job J_i cannot be the next successor of path P. ◀

▶ **Lemma 8.** *Assuming that neither the fast-forward nor the merge phases are executed in Algorithm 1, for any execution scenario such that a job $J_i \in \mathcal{J}$ completes at some time t on core ϕ_k (under the given scheduler), there exists a path $P = \langle v_1, \ldots, v_p, v'_p \rangle$ in the schedule-abstraction graph such that J_i is the label of the edge from v_p to v'_p and $t \in [EFT_k(v'_p), LFT_k(v'_p)]$, where $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are given by Equation* (8).

Proof. Since Algorithm 1 creates a new state in the graph for every job J_i and every core ϕ_k that respects Condition (1), the combination of Lemmas 6 and 7 proves that all possible system states are generated by the algorithm when the fast-forward and merge phases are not executed. Further, Lemma 6 proves that $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finishing time of J_i, meaning that if J_i finishes at t in the execution scenario represented by path P, then t is within $[EFT_k(v'_p), LFT_k(v'_p)]$. ◀

5.2 Soundness of the Fast-Forward Phase

We prove that fast-forwarding will not affect any of the successor states of an updated state.

▶ **Lemma 9.** *Updating the core intervals of vertex v_p during the fast-forwarding phase does not affect any of the states reachable from v_p.*

Proof. Let v_p be the original state and v_q be the updated state after applying (13). Let path P denote the path from v_1 to v_p. Note that state v_q shares the same path P as v_p. We show that for any arbitrary job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ (i.e., those that are not scheduled in path P) and any arbitrary core $\phi_k(v_p) \in v_p$, the EST and LST of job J_i is the same as for core

$\phi_k(v_q) \in v_q$. From this we conclude that all system states reachable from v_p are reachable from v_q and that those reachable states remain unchanged. More precisely, we show that, $\forall k$, **(i)** $EST_{i,k}(v_p) = EST_{i,k}(v_q)$ and **(ii)** $LST_k(v_p) = LST_k(v_q)$.

Claim (i). From (2), we have $EST_{i,k}(v_p) = \max\{r_i^{min}, EFT_k(v_p)\}$. If the EFT of $\phi_k(v_q)$ has not been updated by (13), i.e., $EFT_k(v_p) > t_{min}$, then we trivially have $EST_{i,k}(v_q) = EST_{i,k}(v_p)$. Otherwise, if $EFT_k(v_q)$ has been updated, it must be true that $EFT_k(v_p) \leq t_{min}$ and $EFT_k(v_q) = t_{min}$. In this case, $EST_{i,k}(v_q) = \max\{r_i^{min}, t_{min}\} = \max\{r_i^{min}, EFT_k(v_p)\} = EST_{i,k}(v_p)$ since $EFT_k(v_p) \leq t_{min} \leq r_i^{min}$ (from the definition of t_{min}). Thus, in both cases, $EST_{i,k}(v_p) = EST_{i,k}(v_q)$.

Claim (ii). From (13) we know that if the LFT of a core $\phi_k(v_p)$ is being updated, $LFT_k(v_p) < t_{min}$ and $LFT_k(v_q) = t_{min}$. By definition, $t_{min} = \min\{r_x^{min} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P\} \leq \min\{r_x^{max} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P\} = t_{job}(v_p)$ (the last equality is due to (6)). Moreover, by (5) we have $t_{core}(v_p) \leq LFT_k(v_p) < LFT_k(v_q) = t_{min} \leq t_{job}(v_p)$ and $t_{core}(v_q) \leq LFT_k(v_q) = t_{min} \leq t_{job}(v_q)$ (because t_{job} only depends on path P and v_p and v_q share the same path). Therefore, by (7), $LST_k(v_p) = \min\{t_{high}(v_p) - 1,\ \max\{t_{job}(v_p),\ t_{core}(v_p)\}\} = \min\{t_{high}(v_p) - 1, t_{job}(v_p)\}$ and $LST_k(v_q) = \min\{t_{high}(v_q) - 1,\ \max\{t_{job}(v_q),\ t_{core}(v_q)\}\} = \min\{t_{high}(v_q) - 1, t_{job}(v_q)\}$. Since t_{job} and t_{high} only depend on path P, and v_p and v_q share the same path, the LST in both states is identical, i.e., $LST_k(v_p) = LST_k(v_q)$. ◀

5.3 Soundness of the Merge Phase

We now establish that merging two states is safe, i.e., it neither removes a possible job sequence from the graph (Corollary 16), nor does it decrease the upper bound on the WCRT (or increase the lower bound on the BCRT) of any job in $\mathcal{J}$ (Corollary 18).

We first define the notion of a "mutated" vertex as follows: v'_p is a *mutated* version of v_p if it has the same set of scheduled jobs as the original state v_p and $\forall x$, $EFT_x(v'_p) \leq EFT_x(v_p)$ and $\forall x$, $LFT_x(v_p) \leq LFT_x(v'_p) \vee LFT_x(v_p) \leq t_{job}(v_p)$. We assume that a mutated state v'_p sits in place of the original state v_p in the schedule-abstraction graph.

Next, for any such mutated vertex, we prove that any job that was a direct successor of the original state is also a direct successor of the mutated vertex (Lemma 10). Moreover, we show that the direct successors of mutated states are also mutated (Lemma 11 and 12). This property is then used to prove the main claim that merging is safe. Due to space limitations, we provide the proofs of Lemmas 10 to 14 in an online technical report [19].

▶ **Lemma 10.** *For a vertex v'_p created by mutating v_p, any job J_i that can be scheduled on core $\phi_k(v_p)$ according to* (1)*, can still be scheduled on core $\phi_k(v'_p)$ according to* (1)*.*

▶ **Lemma 11.** *Let v'_p be created by mutating v_p, and let v_q and v'_q be the vertices resulting from scheduling job J_i on core $\phi_k(v_p)$ and $\phi_k(v'_p)$, respectively. $\forall x$, $LFT_x(v'_q) \geq LFT_x(v_q)$ or $LFT_x(v_q) \leq t_{job}(v_q)$.*

▶ **Lemma 12.** *Let v'_p be created by mutating v_p, and let v_q and v'_q be the vertices resulting from scheduling job J_i on core $\phi_k(v_p)$ and $\phi_k(v'_p)$, respectively. $\forall x$, $EFT_x(v'_q) \leq EFT_x(v_q)$.*

▶ **Lemma 13.** *If v'_p is a vertex created by mutating v_p, then all the system states reachable from v_p are also reachable from v'_p.*

▶ **Lemma 14.** *Let v_q and v_p be two vertices such that $\mathcal{J}^P = \mathcal{J}^Q$ (i.e., the set of jobs scheduled until reaching v_q is equal to the set of jobs scheduled until reaching v_p), then the state v'_p resulting from merging v_p and v_q with Algorithm 2 is a mutated version of both v_p and v_q.*

By successively applying Lemmas 13 and 14, we obtain the following corollary.

▶ **Corollary 15.** *Let v_q and v_p be two vertices such that $\mathcal{J}^P = \mathcal{J}^Q$ (i.e., the set of jobs scheduled until reaching v_q is equal to the set of jobs scheduled until reaching v_p), all system states reachable from v_p and v_q are also reachable from the merged state v'_p.*

▶ **Corollary 16.** *For two states that are merged by Algorithm 1, all system states reachable from either of them are also reachable from the merged state.*

Proof. Since for two states v_p and v_q, Definition 4 enforces that $\mathcal{J}^P = \mathcal{J}^Q$, the resulting merged state satisfies the requirement of Corollary 15 and hence proves the claim. ◀

5.4 Soundness of Algorithm 1

By successively applying Lemmas 8 and 9 and then Corollary 16, we obtain that the analysis is safe, as stated in Theorem 17 and its corollary below.

▶ **Theorem 17.** *For any execution scenario such that a job $J_i \in \mathcal{J}$ completes at some time t on core ϕ_k (under the given scheduler), there exists a path $P = \langle v_1, \ldots, v_p, v'_p \rangle$ in the schedule-abstraction graph such that J_i is the label of the edge from v_p to v'_p and $t \in [EFT_k(v'_p), LFT_k(v'_p)]$, where $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are given by Equation* (8).

▶ **Corollary 18.** *Lines 10 and 11 of Algorithm 1 calculate a lower and an upper bound on the BCRT and WCRT, respectively, of every job in $\mathcal{J}$.*

Proof. Lines 10 and 11 obtain a job's response time directly from (8), which provides correct bounds on the earliest and latest finish times of a job according to Lemma 6. Since according to Theorem 17, for any execution scenario, there is a path in the graph, Algorithm 1 includes all possible schedules of a job and hence the obtained values are correctly lower-bounding and upper-bounding the actual BCRT and WCRT of that job. ◀

5.5 Inexactness of Algorithm 1

The following example shows that the abstraction that we use to represent core states may reflect impossible execution scenarios. Therefore, Algorithm 1 is sufficient but not exact.

Assume that a system state v_p contains two core intervals $\phi_1 = [5, 10]$ and $\phi_2 = [1, 10]$ and that there is an unscheduled job J_1 with $C_1^{min} = C_1^{max} = 5$, $r_1^{min} = r_1^{max} = 1$, and $d_1 = 30$. Further, assume that during the expansion phase of Algorithm 1, J_1 is dispatched to ϕ_1, which results in $\phi_1 = [10, 15]$ and $\phi_2 = [5, 10]$ (after the update phase). According to this new system state, it may happen that core ϕ_2 becomes available at time $5 \in [5, 10]$, and that core ϕ_1 remains busy until time $15 \in [10, 15]$. However, this scenario is actually impossible. If ϕ_1 remains busy until time 15, then J_1 must have started to execute at time 10, implying that both ϕ_1 and ϕ_2 must have been busy until time 10. Otherwise, job J_1 would have been dispatched on ϕ_2 rather than ϕ_1. In other words, ϕ_1 may become available at time 15 only if ϕ_2 becomes available no earlier than time 10. This example shows a dependency between the availability time of the cores, which is ignored in the current system state abstraction to keep the system state encoding simple, and to increase the number of states that can be merged. This design decision, however, makes the analysis inexact since it considers all possible but also some impossible execution scenarios.

6 Empirical Evaluation

We conducted experiments to answer two main questions: **(i)** does our test yield better schedulability; and **(ii)** is the runtime of our analysis practical? To answer the first question, we applied Algorithm 1 to two global non-preemptive scheduling policies: G-NP-FP and G-NP-EDF. As we are unaware of any schedulability analysis for non-preemptive job sets (or periodic tasks) for the aforementioned global scheduling policies, we used the existing tests designed for sporadic non-preemptive task sets as a baseline. These tests include the schedulability test of Baruah [4] for G-NP-EDF (denoted by Baruah-EDF), two tests of Guan et al. [10] for any global non-preemptive work-conserving scheduler (denoted by Guan-Test1-WC), and for G-NP-FP (denoted by Guan-Test2-FP), and the recent schedulability test of Lee (denoted by Lee-FP) [13]. For the sake of comparison, we used simple rate-monotonic priorities for the fixed-priority tests since we did not observe substantial differences when trying out other heuristics such as laxity-monotonic priorities.

To randomly generate a periodic task set with n tasks and a given utilization U, we first randomly generated n period values in the range [10000, 100000] microseconds with log-uniform distribution (and a granularity of $5000\mu s$ as suggested by Emberson et al. [8]. We then used the RandFixSum [22] algorithm to generate n random task-utilization values that sum to U. From the task utilization, we obtained C_i^{max} and set C_i^{min} to be $0.1 \cdot C_i^{max}$. Tasks were assumed to have implicit deadlines. We discarded any task set that had more than 100000 jobs per hyperperiod. Although, in theory, a hyperperiod may contain many more jobs, in industrial settings, e.g., automotive systems [12], periods are usually chosen such that the hyperperiod includes only at most a couple of thousand jobs.

The experiments were performed by varying **(i)** the total system utilization U (for 4 cores and 10 tasks), **(ii)** the number of tasks n (for 4 cores and $U = 2.8$, which is 70% of the capacity of the cores), **(iii)** the number of cores m (for 10 tasks and $U = 2.8$), and **(iv)** the total task utilization U while tasks had 100 microseconds release jitter (10 tasks and 4 cores). This roughly represents jitter magnitudes that can be expected due to interrupt handling delays. For each combination of n, m, and U, 1000 random task sets were generated.

To evaluate schedulability of a task set, we implemented Algorithm 1 as a single-threaded C++ program and performed the analysis on a cluster of hosts having an Intel Xeon E7-8857 v2 processor clocked at 3 GHz and 1.5 TiB RAM. In the experiments, a task set was claimed unschedulable as soon as either an execution scenario with a deadline miss was found or a timeout of four hours was reached. Fig. 3 reports the observed schedulability ratio and runtime of Algorithm 1 for different setups. The schedulability ratio is the ratio of task sets deemed to be schedulable divided by the number of generated task sets.

Schedulability results. Figs. 3-(a) to (c) show a significant gap between the schedulability ratio of our solution and the state-of-the-art tests. For example, while Lee-FP could only identify 8% of schedulable task sets for $U = 2.4$, our test shows that at least 72% of them are schedulable. Similar patterns are seen when for increasing task and core counts. Fig. 3-(b) shows that schedulability improves as the number of tasks increases. This is since, by keeping U constant, increasing n decreases per-task utilization, which in turn reduces WCETs and blocking. Thus, more task sets become schedulable. Of the existing tests, however, only Lee-FP and Guan-Test2-FP benefit from this effect, and only by up to 16% (for $n = 30$).

With the increase in the number of cores, blocking scenarios caused by tasks with large execution times are less likely to occur and hence more task sets are deemed schedulable. However, as shown in Fig. 3-(c), the current tests are quite pessimistic, e.g., Lee-FP could

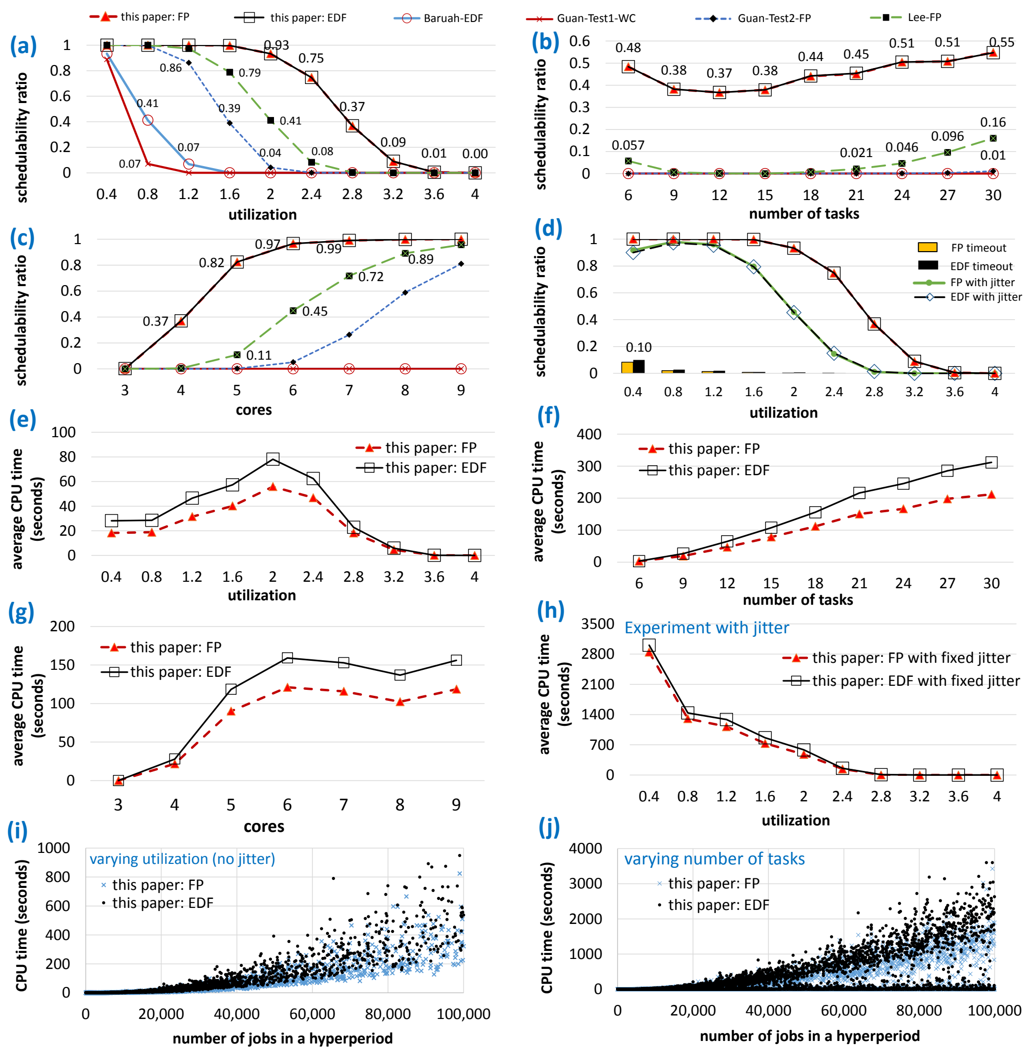


Figure 3 Experimental results for various parameters. **(a, b, c, d)** Schedulability ratio. **(e, f, g, h)** Average analysis runtime. **(i, j)** Analysis runtime vs. the number of jobs in a hyperperiod.

identify only 11% of the task sets as schedulable when (at least) 82% of the task sets are schedulable on 5 cores. From Figs. 3-(a) to (c), we conclude that our analysis is able to reclaim a large portion of pessimism in the baseline analyses (when applied to periodic tasks).

Fig. 3-(d) shows the effect of jitter on schedulability. Since jitter increases the number of possible interleavings between the start time of the tasks, more blocking scenarios become possible and hence tasks with tight deadlines may become unschedulable. This behavior can be observed in the average runtime of the analysis reported in Fig. 3-(h). Yet, our analysis achieves a substantially higher schedulability ratio than the baselines.

It is worth noting that for $U = 0.4$, the counterintuitive drop in schedulability for tasks with jitter is due to the timeout. The bar chart shown at the bottom of Fig. 3-(d) represents the ratio of task sets that could not be analyzed within the four-hour limit. The reason is that for $U = 0.4$, tasks have a small WCET and thus more combinations of job orderings may require analysis before Algorithm 1 is able to merge the branches. In the future, we plan to develop techniques to handle lower (or higher) utilization tasks differently, e.g., by designing more eager merge rules that combine paths with different job sets.

Moreover, we observed that the gap between the schedulability ratio of EDF and FP is small because most of the deadline misses are due to the work-conserving nature of the policy rather than the priority assignment. Namely, since a work-conserving scheduler cannot leave the processor idle, it will schedule any lower-priority job before the next higher-priority job is released. As a result, high-frequency tasks with tight deadlines will miss their deadline before the priority assignment method can play a significant role in improving the order of executions. We conclude that there is a need for a global scheduling algorithm that is able to avoid such blocking scenarios, for instance by being non-work-conserving. While such non-work-conserving non-preemptive scheduling algorithms have recently been proposed for uniprocessor systems [17, 18], currently no such solution exists for multiprocessor platforms.

Runtime of the analysis. Fig. 3-(e) shows that the average analysis runtime increases with increasing task-set utilization, since busy windows become longer. Consequently, paths that have the same set of jobs are merged only at later stages. For larger utilizations such as for $U \geq 2.8$, however, identifying *unschedulable* task sets becomes easy due to the presence of tasks with large WCETs that can block all cores for a long time. Since we stop the analysis as soon as a deadline miss is found, not-schedulable task sets with large utilization can be identified quickly. The analysis runtime hence decreases rapidly for larger utilization values.

Figs. 3-(f) and (g) show that the analysis runtime grows with increasing tasks and core counts because more states are generated in the expansion phase. It is worth noting that unlike the effect pertaining to the number of tasks, increasing the number of cores will not increase the runtime monotonically. The reason is that, as shown in Fig. 3-(c), for a workload with $U = 2.8$ and 10 tasks, almost all task sets are schedulable on 6 cores or more. That is, the number of cores *per se* only has a limited effect on the runtime of the algorithm; however, larger platforms are likely to host large task sets, with a potentially large number of jobs per hyperperiod, and our analysis is sensitive to such increases in workload size.

Figs. 3-(i) and 3-(j) report the runtime of the analysis for each task set w.r.t. the number of jobs in a hyperperiod for two scenarios: varying utilization and varying the number of tasks, respectively. As shown by the figures, the runtime of the analysis grows with the increase in the number of jobs in a hyperperiod. We also observe that with an increase in the number of tasks from 10 (Fig. 3-(i)) to up to 30 (Fig. 3-(j)), the largest observed runtime of the analysis grows linearly, i.e., from 1000 to 4000.

Since a naive analysis without path merging does not scale even for a uniprocessor system, as shown in [16], we did not perform a separate experiment to show the efficiency of the path merging technique. In the future, we plan to further explore the design space for different merge conditions and their efficiency for different task set types and utilizations.

Overall, we conclude that: **(i)** the proposed analysis is practical for realistic workload sizes, **(ii)** it identifies a significantly larger portion of schedulable tasks in comparison with state-of-the-art tests for sporadic tasks, and **(iii)** even when jitter is considered (which allows for more blocking scenarios and uncertainties), our analysis still achieves much higher schedulability than the baseline tests (which, to be clear, are designed for sporadic task sets).

In terms of limitations, we also observed that the runtime of the analysis grows quickly (e.g., more task sets hit the four-hour timeout) for larger systems (e.g., when $n \geq 20$ and $m \geq 16$). This is due to the increase in the number of tasks and the number of ways a task can be assigned to a core in the expansion phase of the algorithm. To scale to such large systems, a more efficient abstraction is needed that allows for more eager merging techniques.

7 Conclusion

The paper provides a sufficient schedulability analysis for global job-level fixed-priority scheduling algorithms and non-preemptive job sets. We have presented a technique for deriving an upper bound on the WCRT and a lower bound on the BCRT by exploring an abstraction of all possible schedules of a job set that reflects the uncertainties in job execution and release times. We developed the notion of a schedule-abstraction graph for global schedulers and introduced two key techniques, namely path merging and fast-forwarding, to slow the state-space growth and proved the analysis to be sound.

Our empirical evaluation using periodic workloads shows significant schedulability improvements w.r.t. the state-of-the-art tests in all experimental setups. The observed runtime of the analysis ranged from a couple of seconds to a couple of hours for realistic system setups, e.g., up to 30 tasks, up to 9 cores, and up to 100000 jobs per hyperperiod, which is an acceptable performance for an offline, design-time analysis.

Furthermore, our current implementation is sequential. We expect that parallelizing the analysis, so that naturally independent scenarios are explored in parallel, would yield a substantial speedup. To this end, we hope to derive rules that allow maximum paralellism between independent exploration frontiers. Moreover, we will investigate different merge rules to reduce the runtime of the analysis. We also plan to extend the solution presented here to analyze systems with more complicated properties such as precedence constraints and preemption points, and to other scheduling problems such as gang scheduling.

References

1 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *ACM International Conference on Embedded Software*, pages 20:1–20:10, 2014.

2 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

3 Theodore P. Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 62–75. Springer, 2007.

4 Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, 2006.

5 Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. *Algorithmica*, 63(4):763–780, 2012.

6 Artem Burmyakov, Enrico Bini, and Eduardo Tovar. An exact schedulability test for global FP using state space pruning. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2015.

7 Anton Cervin, Bo Lincoln, Karl-Erik Arzen, and Giorgio Buttazzo. The Jitter Margin and Its Application in the Design of Real-Time Control Systems. In *International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, pages 1–9, 2004.

8 Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.

9 Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 263–272, 2007.

10 Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011.

11 Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 137–146, 2008.

12 S. Kramer, D Ziegenbein, and A Hamann. Real world automotive benchmark for free. In *International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, 2015.

13 Jinkyu Lee. Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling. *IEEE Transactions on Computers*, 66(10):1816–1823, 2017.

14 Jinkyu Lee and Kang G. Shin. Improvement of real-time multi-coreschedulability with forced non-preemption. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1233–1243, 2014.

15 Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2017.

16 Mitra Nasri and Björn B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2017.

17 Mitra Nasri and Gerhard Fohler. Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 165–175, 2016.

18 Mitra Nasri and Mehdi Kargahi. Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Systems*, 50(4):548–584, 2014.

19 Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. Technical Report MPI-SWS-2018-003, Max Planck Institute for Software Systems, Germany, 2018. URL: `http://www.mpi-sws.org/tr/2018-003.pdf`.

20 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, 2011.

21 Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel real-time scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

22 Roger Stafford. Random vectors with fixed sum. Technical report, University of Oxford, 2006. URL: `http://www.mathworks.com/matlabcentral/fileexchange/9700`.

23 Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling. *Real-Time Syst.*, 52(3):323–355, 2016.

24 Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric OS for multi-core embedded systems. In *IEEE Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016.

25 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The

worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008.

26 Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses

Paolo Pazzaglia
Scuola Superiore Sant'Anna, Pisa, Italy
paolo.pazzaglia@sssup.it

Luigi Pannocchi
Scuola Superiore Sant'Anna, Pisa, Italy
luigi.pannocchi@sssup.it

Alessandro Biondi
Scuola Superiore Sant'Anna, Pisa, Italy
alessandro.biondi@sssup.it

Marco Di Natale
Scuola Superiore Sant'Anna, Pisa, Italy
marco@sssup.it

Abstract

Most works in schedulability analysis theory are based on the assumption that constraints on the performance of the application can be expressed by a very limited set of timing constraints (often simply hard deadlines) on a task model. This model is insufficient to represent a large number of systems in which deadlines can be missed, or in which late task responses affect the performance, but not the correctness of the application. For systems with a possible temporary overload, models like the m-K deadline have been proposed in the past. However, the m-K model has several limitations since it does not consider the state of the system and is largely unaware of the way in which the performance is affected by deadline misses (except for critical failures). In this paper, we present a state-based representation of the evolution of a system with respect to each deadline hit or miss event. Our representation is much more general (while hopefully concise enough) to represent the evolution in time of the performance of time-sensitive systems with possible time overloads. We provide the theoretical foundations for our model and also show an application to a simple system to give examples of the state representations and their use.

2012 ACM Subject Classification Computer systems organization → Embedded software

Keywords and phrases control, real-time, cyber physical systems, weakly hard, deadline miss, performance

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.10

Supplement Material ECRTS Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.4.2.4`

1 Introduction and Motivation

The development of most control applications is based on the assumption that the design, definition and analysis of the controls functionality can be separated from the development and analysis of the software code implementing it. The functional model of the controls and of the program threads implementing them are connected by a suitable set of assumptions on the time properties of the code. In most cases, the assumptions relate to the activation periods, the maximum allowed response times (or deadlines) and possibly the output jitter.

In many instances, if the periodic tasks are schedulable within the deadlines, then the system is assumed to be correct. This assumption corresponds to the hard deadline model.

As highlighted by several authors (a comprehensive discussion of the issue is in [10]), this assumption is simplistic and may lead to overprovisioning of resources on one side (by requesting that all deadlines are met when in reality the system may be tolerant to some deadline misses) and the inability to model the impact of late responses as a performance degradation, given that the only possible outcome is a binary (feasible, infeasible) assessment of the system correctness.

To cope with the possibility of deadline misses, more sophisticated task models have been proposed, including analysis methods that compute the maximum lateness or number of consecutive deadline misses, or the Weakly Hard model, aimed at verifying whether the system can guarantee that at most m deadlines are missed for every set of K consecutive task instances. These models have been developed by the real-time analysis community, often inspired by generic requirements from controls developers, but mostly abstract from considerations on the performance of the controls.

Alternatively, the timing model of the software tasks is included in a general model of the system together with the model of the controls, that is, the two domains are not separated but jointly considered. Examples of these approaches include system models using hybrid or timed automata (such as those used by the Times tool [18]) and models that cosimulate the control functionality and the task timing, to assess the impact of scheduling delays on the performance (examples are the Jitterbug, TrueTime [10] and TRes tools [11]).

Contribution and Paper Structure

In this work, we propose to define a new abstraction for Cyber Physical Systems (CPSs) analysis that represents the performance degradation of the system in correspondence to possible deadline misses. In particular, the focus of the paper is on computing the evolution of control performance for a CPS in which its control tasks can suffer sporadic deadline misses, that can be described by a set of Weakly Hard constraints. We consider a task actuation implemented using the LET paradigm, where the control output is updated at the task deadlines. When a job misses its deadline, the control output is not updated. The LET implementation imposes fixed delays of the control output, thus enabling a precise analysis of the control system. First, the *freshness* of the control output is extracted for each step of the possible sequences of hit and missed deadlines, considering different handling methods for the deadline miss event. Then, the corresponding sequence of update matrices for the state variables is constructed, and a performance value is assigned to the sequence. The *sum of squared errors* is chosen as a representative performance index. The proposed approach allows extracting useful information such as worst-case performance bounds and critical sequences of deadline hits/misses with respect to a target performance.

Our model is more detailed than the Weakly Hard model (summarized in Section 2) since it considers the evolution of the system state in correspondence to miss events and, by taking into account the actual patterns of hits and misses, it includes an estimate of the system performance at each state. The proposed model is still much simpler than hybrid automata since it only considers the impact of a finite number of job completions and abstracts the time behavior by means of deadline miss/hit sequences. The model of the system assumed in this paper is summarized in Section 3, and the proposed approach with the overall objectives are discussed in Section 4.

We show how the size of the state description can be bounded by a set of scheduling assumptions, or as a result of timing analysis, and how the performance annotation can be computed for simple metric functions on Linear Time Invariant (LTI) systems. Furthermore, under the hypothesis that the system evolution is bounded by an exponential function, the number of steps in each analyzed sequence can be effectively limited with a bounded error on the performance index. Section 5 contains the description of how to compute the impact on the control values (update freshness) as a consequence of possible deadline misses, and Section 6 how to compute a state trajectory from sequences of hits and misses. These results are used in Section 7 to assign performance values to the sequences and possibly compute the worst-case values. Finally, in Section 8 we show the application of the proposed method to a case study consisting of a control of a Furuta pendulum.

1.1 State of the Art

The *weakly-hard* real-time schedulability analysis targets the problem of bounding the maximum number of deadline misses over a number of task activations. A dynamic assignment of priorities for streams with m-K requirements is proposed by Hamdaoui et al. [20] to reduce the probability of missing more than m deadlines every K iterations. Weakly hard real-time schedulability analysis can be traced back to the work of Bernat et al. [5] on the m-K model. The analysis in [5] and in other works assumes that there is an explicit initial state of the system, in which the initial offset of each task in the system is known. This limitation is removed in [34].

Recent developments in the study of overloaded systems allow to relax the requirement of knowing the initial system state. The approach proposed by Quinton et al. [23] consists in the *worst-case analysis* of a system model represented as the superposition of a typical behavior (e.g., of periodic task activations) that is assumed feasible, and a sporadic overload (i.e., a rare event). Under such an assumption, other works [16, 30] proposed methods for weakly-hard analysis that consists of two phases: 1) the system is verified to be schedulable under the typical scenario (by the classical hard analysis), and 2) when the system is overloaded, it can be guaranteed that out of K successive activations of a task, at most m of them will miss the deadline.

The analysis of overload conditions is also closely related to the co-design of control and CPU-time scheduling [3]. The influence of response times on the performance of control tasks has been studied in several works such as those by Xu et al [31, 32]. Aminifar et al. [2] proposed an integrated approach for controller synthesis, by selecting the task parameters that meet the expected control performance and guarantee the stability. The concept is later extended [1] to distributed Cyber Physical Systems, while in [15] FlexRay is considered as the communication medium. The m-K model has also been investigated in the co-design of controls (with respect to their performance) and scheduling [12, 24], and is used in [7, 8, 26, 27] to define the maximum number of samples (jobs) that can be dropped over any sequence (density of dropped samples), to guarantee a minimum level of quality to the controls. In [29] the problem of modifying the controller for improving the worst-case performance under m-K constraints is addressed. Moreover, the m-K model has been used for describing stability properties of a controlled system, e.g., to account for the maximum number of deadlines that can be missed in a row without making the system unstable (see [25]). Recent work by Blind and Allgöwer [6] showed that an unstable plant with feedback control that executes in open loop for finite time intervals under m-K constraints, can be subject to stability analysis by means of the Lyapunov method. In a subsequent work, Linsenmayer and Allgöwer [19] faced the problem of finding a controller that stabilizes the plant described in [6] under a given set of weakly hard constraints.

In the controls literature, Yoshimoto and Ushio [33] consider an overloaded real-time platform with multiple controllers. The system described is a LTI plant, with the deadline of the control task equal to the period of the task. They create an arbiter for skipping jobs that maintain the system schedulable while minimizing a performance degradation index based on the number of *consecutively* skipped jobs.

Frehse et al. [12] consider a Weakly Hard system, with a control task that is implemented using the Logical Execution Time (LET) paradigm [17]. The authors create a hybrid automaton that represent the connection between the physical system (described with piecewise affine functions) and the discrete controller, analysing the system with the TWCA approach [23]. They propose then the use of reachability analysis with the model checker SpaceEx to analyze if the trajectories guarantee the required performance.

2 The Weakly-Hard Task Model

The *m-K* model [20] and its generalization in the weakly-hard model [5] are an attempt at describing the impact of deadline misses on the *correctness* of the application. The approach can be used for a real-time system in which a given number of misses can be tolerated without critical consequences. When the number of deadline misses can be bounded in any time interval of a given length, the system is defined as *weakly-hard.*

In this paper, we are interested in Cyber-Physical Systems where control tasks have weakly-hard constraints. The main goal is to extract bounds for the control *performance* as a function of the weakly-hard constraints, and monitor the system at each step. To make the paper self-contained, this section recalls the standard definitions for weakly-hard real-time systems, and provides an overview of the major limitations of state-of-the-art approaches.

2.1 Definitions

In a weakly-hard system, the (m, K) constraint provides a bound on the number m of deadline misses that a task can experience every K instances (i.e., jobs). In the following, the definitions of *satisfaction set* and *hardness* of an (m, K) constraint, taken from the work of Bernat, Burns, and Llamosí [5], are used.

▶ **Definition 1** (Satisfaction set). Given a constraint (m, K), the *satisfaction set* $\mathcal{S}_N$ of (m, K) is the set of all the sequences of hit and missed deadlines of length N that satisfy the constraint.

A sequence $s \in \mathcal{S}_N$ is represented by a string of N letters, using "M" for a deadline miss and "H" for a hit. When necessary, the sequence s will also be denoted as *H/M.* As a particular case, the satisfaction set of the constraint $(0, K)$ (for any K) includes sequences with all hits. This sequence is named hereafter as *H-sequence.* The H-sequence, also denoted as s_H, is the only sequence that satisfies all the possible (m, K) constraints.

When analyzing the weakly-hard properties of a real-time task, it is generally possible to extract a set of constraints under different values for K. A set of p weakly-hard constraints is denoted with Γ, and is defined as: $\Gamma = \{(m_1, K_1), (m_2, K_2), \ldots, (m_p, K_p)\}$. The definition of satisfaction set is then extended to a set Γ:

▶ **Definition 2.** Given a set Γ of weakly-hard constraints, the satisfaction set of Γ is the *intersection* of the satisfaction sets of all the constraints in Γ.

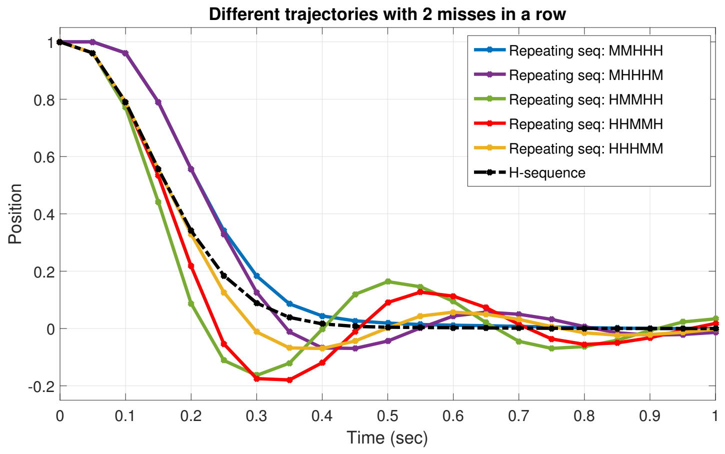


Figure 1 Comparing different state trajectories of a controlled roller moving a sheet of paper, with 2 deadlines missed in a row every 5 instances. Changing the order of the missed deadlines in a H/M sequence leads to different behaviors with different control performance.

2.2 Limitations of the Weakly-Hard Model

In this work, we are interested in finding a model that puts in correspondence the timing properties of a control task (expressed as a sequence of deadline hits and misses), the dynamics of the physical plant (the controlled system), and the control performance. Unfortunately, our objective cannot be achieved with a simple m-K weakly-hard model.

In fact, the weakly-hard model has several limitations when applied to the control domain. First of all, the (m, K) constraint is a model with a "binary" outcome, where either the system satisfies the constraint or not (e.g., for the purpose of stability analysis [25]), but it does not provide any information concerning the control performance that can be guaranteed when the constraint is met.

Another important limitation is that the *order* (i.e., the actual pattern) of deadline misses and hits cannot be fully described by using a combination of (m, K) constraints. This is true even when using the extended model proposed in [5], where the maximum number of *consecutive* deadline misses is considered. Indeed, *different H/M sequences in the same satisfaction set generally lead to different control performance values*, thus making the (m, K) constraint a coarse (and possibly misleading) description of the system when addressing the performance analysis. The explicit consideration of all the valid H/M sequences, together with the evolution of the system state during the sequence, may be important to enable a precise study of the control performance. For instance, if the system is near the steady state, the errors produced by an actuation that uses stale data (e.g., as a result of a deadline miss) are limited. However, if the system is in a transient condition, actuation errors will have a much higher impact. Since the *sensitivity* of the control performance changes with the system state and its evolution, the adoption of the existing weakly-hard models to analyze the control performance requires to always account for the overall worst-case scenario.

For example, consider the model of a *paper roller*, as described in [22] (pag. 40), which is controlled to zero with a periodic task with $T = 50ms$ and $D = 0.7T$. The control task satisfies a weakly-hard constraint with 2 deadline misses in a row every 5 instances. Five H/M sequences that satisfy the considered weakly-hard constraints are applied, and when a deadline is missed the control output is not updated. The results are reported in Figure 1 and show that each H/M sequence leads to a different state trajectory. If the sum of quadratic errors (with respect to a reference) is considered as a control performance metric, then different values will be computed for each trajectory.

These observations motivate the development of a richer model, with an associated analysis technique to study the control performance under weakly-hard constraints.

3 System Model

This section presents the model of the considered Cyber-Physical System in terms of the physical plant, the controller, and its task implementation.

3.1 Plant and Control Description

We consider a physical system modeled as a Linear Time-Invariant (LTI) plant, multi-input-multi-output (MIMO), strictly causal, operating in a well-defined region $\Omega \in \mathbb{R}^n$ of the state space (possibly the entire state space $\mathbb{R}^n$). A continuous-time description of the plant is provided by the following state equation:

$$\dot{x}_c(t) = A_c x_c(t) + B_c u_c(t), \tag{1}$$

where $x_c(t)$ is the state vector, $u_c(t)$ the control output, and A_c and B_c are constant matrices of the dynamics with appropriate dimensions. We assume that a discrete-time controller is implemented as a periodic task τ_i, with period T_i and relative deadline $D_i \leq T_i$. The task is released at the system startup, i.e., time $t = 0$. The period T_i is chosen as a compromise between the stability properties of the system (e.g., the phase margin) and the schedulability constraints given by the available computational resources (speed of the processor, communication rates, etc.). A general heuristic for the choice of the sampling rate consists in guaranteeing that there are 4 to 10 samples during the rise time in response to a step input [28].

The time interval between the k-th and the $(k+1)$-th activation of the control task is defined as $[kT_i, (k+1)T_i)$.

The Logical Execution Time (LET) paradigm is adopted [12] for the control task. An example execution is shown in Figure 2 with three tasks for sensing (τ_S), control (τ_C), and actuation (τ_A). The k-th job of the control task uses the system state sensed at the activation time kT_i (without sensing jitter), and the output is used by the actuator at the deadline (time $kT_i + D_i$). Furthermore, we assume that the actuation value is kept constant until the next update, which occurs at time $(k+1)T_i + D_i$ (as shown by the $u[k-1]$, $u[k] = u[k+1]$ and $u[k+2]$ values in Figure 2). The choice of updating the control value at the task deadline leads to *regularity* of the output timing, with no output jitter, thus simplifying the control synthesis by means of classic techniques based on the assumption of a constant delay D_i (e.g., see [21] for the case with $D_i = T_i$). Moreover, the enforcement of fixed time instants for the control update enhances the *predictability* of the system.

The notation $x[k] = x_c(kT_i)$ is used to denote the discrete-time measure (or an estimate) of the system state. Analogously, $u[k]$ denotes the discrete-time representation of the control output. Note that, due to the delay D_i, $x[k]$ and $u[k]$ are not updated at the same time instant. We assume $x[k] = x[0], \forall k < 0$.

The output $u[k]$ is generated by a control function that is assumed to stabilize the discrete-time plant under consideration. The discrete-time representation of the plant is [4]

$$x[k+1] = A_d x[k] + B_{d1} u[k-1] + B_{d2} u[k], \tag{2}$$

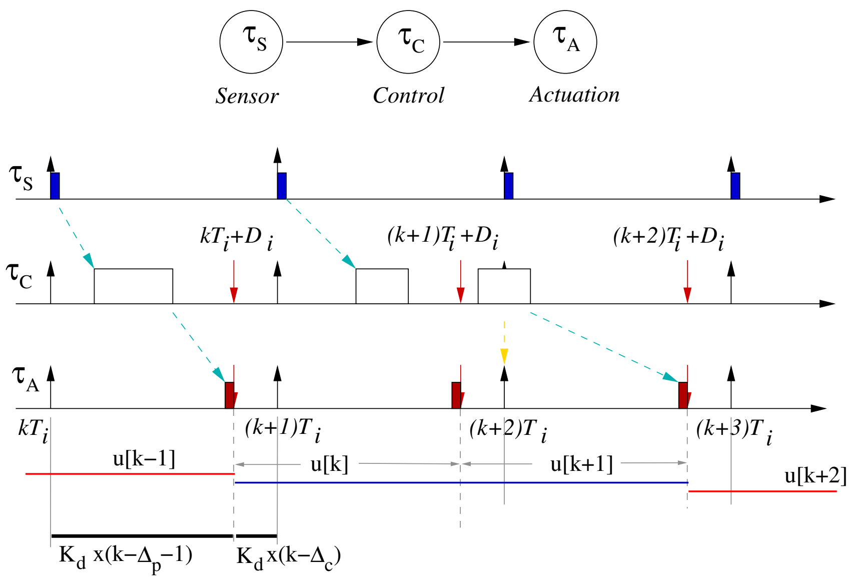


Figure 2 Time execution of sensor, control and actuation tasks.

where the involved matrices are defined as:

$$A_d = e^{A_c T_i}, \tag{3}$$

$$B_{d1} = \int_0^{D_i} e^{A_c s} ds\, B_c, \text{ and } \; B_{d2} = \int_{D_i}^{T_i} e^{A_c s} ds\, B_c. \tag{4}$$

For this work, we consider a standard state-feedback controller of the following form:

$$u[k] = K_d(r - x[k]), \tag{5}$$

where K_d is the stabilizing control matrix, and r is the reference equilibrium state. Without loss of generality, from now on we consider that that the reference is equal to the zero vector, i.e., $u[k] = -K_d x[k]$. Finally, we assume that the initial state $x[0]$ (and thus the input $u[0]$) is known.

In the following, we are interested in reasoning about the output value before and after the deadline for each periodic instance. As shown in Figure 2, in the portion of the control period before the deadline, the value considered is $u[k-1] = K_d x(k - \Delta_p - 1)$, and in the other part is $u[k] = K_d x(k - \Delta_c)$. The delays Δ_p and Δ_c, which will be defined and analyzed in detail in Section 5, depend on possible deadline misses (for example, $\Delta_p = 0$ and $\Delta_c = 0$ for the first cycle in the Figure, and $\Delta_p = 1$ and $\Delta_c = 1$ for the third cycle).

3.2 Task Set Description

The control task τ_i is implemented on a real-time platform, together with N other tasks that can be either periodic or non-periodic. The Worst Case Execution Time (WCET) is assumed to be known for each task and the periodic control task τ_i has WCET $C_i \leq D_i$. We assume a fixed-priority, fully-preemptive scheduler, as in most established commercial standards. Of course, the approach is also applicable to multiple independent control tasks in execution on the same core.

3.3 The Task Model of the Controller

The effect of a deadline miss on the controller output depends on the structure of the controller task and the semantics of information passing. We assume that the task τ_i computes the actuation command using the data sensed at every activation. At its completion, the task copies the output data in a shared memory address, making it accessible to the process that handles the actuator. To match the considered LET paradigm, we also assume that the actuator is activated with the same period of τ_i (equal to T_i) but executed at its deadline (with very high priority and minimum jitter, possibly as a hardware implementation, Figure 2). The actuator reads the data from the shared memory and uses it as the actuation value during the next interval of length T_i. This means that if the output variable of τ_i is not ready at the deadline, the previously-stored value is used for the actuation.

To ensure a *one-to-one correspondence* between each element in an H/M sequence and the corresponding actuation update, we restrict our analysis to the case in which every execution of a control job is guaranteed to complete at least within $(T_i + D_i)$ time units from its activation. This means that the Worst-Case Response Time $WCRT_i$ of the task τ_i, is upper bounded as follows:

$$WCRT_i < T_i + D_i. \tag{6}$$

This condition ensures that there cannot be more than one pending invocation of τ_i at each deadline. The case in which the WCRT exceeds $(T_i + D_i)$ needs a richer description than H/M sequences, hence a considerable additional complexity for the model and the analysis: for this reason it is left as a future work. However, note that only part of the presented analysis relies on this assumption (further details are provided in Section 5.1.1).

4 Approach and Objective

The objective of this work is to propose a new model for relating the performance of control systems to schedulability conditions with possible deadline misses. Our model consists of a (finite) state-based representation, in which a control job belongs at every point in time to one state in the set. The new state is evaluated at each deadline hit or miss event, and represents a specific sequence of hits and misses. This state is annotated with a control performance value (as summarized in Figure 3).

This model can be used as a time contract between the design of the controls and their software (task) implementation. It enables

- the definition of monitors that can not only intercept at run time unforeseen timing faults, but also possible performance degradation that requires a recovery action, and
- the definition of the performance of the system at each deadline hit or miss event.

The approach presented in this paper is divided in steps:

1. A conventional m-K timing analysis (such as in [5] or [34]) is assumed to be performed on the tasks (using the timing model with WCETs). This analysis allows to bound the possible sequences of hits and misses and the number of states that are reachable. Safety monitors can be added to the system to guard against additional misses that can bring the system outside of the set of reachable states and would indicate an error in the estimate of the WCETs or other inaccuracies in the model of the task set.
2. For each state and each hit or miss event, two parameters (Δ_p, Δ_c) are computed (as shown at the bottom of Figure 2), that relate to the *freshness* of the control updates. This step requires only knowledge on the task execution model and the deadline handling

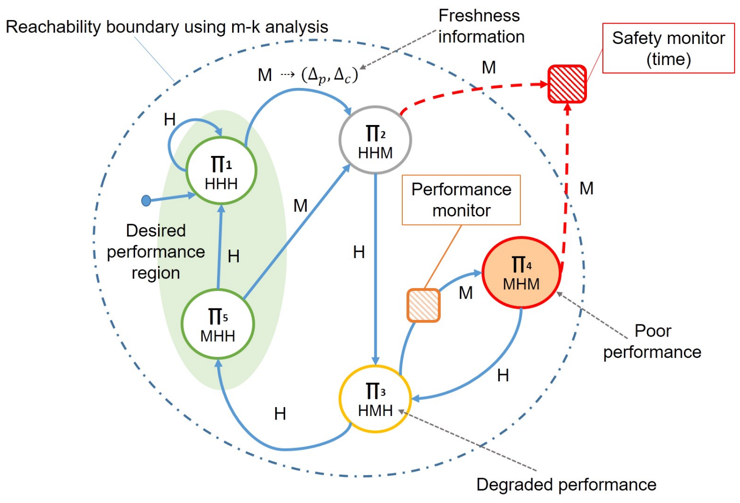


Figure 3 State machine representation of performance indexes for H/M sequences with constraint $(m, K) = (1, 2)$ and window of $N = 3$ steps. The reachability boundary restricts the analysis only to the possible combinations of N steps of hits and misses that satisfy the given (m, K) constraint.

strategy (extracted from the software implementation of the controls). The values Δ_p, Δ_c are computed using a set of state machines as shown in Figure 4 of Section 5 (different from the one of Figure 3).

3. Each state is annotated by the corresponding state update matrix, computed considering the plant model and the control parameters.
4. Performance bounds for the controlled system are extracted, and a description suitable for the on-line monitoring of the system behavior is created, in which each state is annotated by a performance matrix Π_i. Whenever the system transitions to a state with a degraded performance and before it can further progress into a poor performance condition, a monitor may be triggered to try to perform recovery actions.

In the following sections, steps number 2, 3 and 4 are analyzed in detail.

5 From Deadline Misses to Update Freshness

This section studies the impact of deadline misses on the *freshness* of the control updates. The presented approach is general enough to be applied to different deadline miss management policies. This analysis step only requires information related to the software implementation of the controller, i.e., it is independent of the physical system and the control parameters.

5.1 Handling Deadline Misses

Depending on the system implementation, and possibly on the configuration of the operating system, a job that misses its deadline can be handled in different ways. In this work, two common strategies are considered:

1. **Job killed:** the execution of the job that misses its deadline is aborted.
2. **Job continued:** a job that misses its deadline continues to execute until it completes. Multiple pending jobs are served in first-in-first-out order.

Each strategy not only results in a different effect on the timing properties (schedulability and response time) of the task set, but also in a different impact on the *freshness* of the control update and a different performance of the control system.

The update freshness is a parameter used to describe the age (as a number of control steps) of the current actuation value, which can be formally defined as follows.

▶ **Definition 3** (Update freshness Δ_k). Let k' be the control step for which the state $x[k']$ is used to generate the k-th control update $u[k]$, with $k' \leq k$. The *update freshness* $\Delta_k \in \mathbb{N}_{\geq 0}$ is defined as $\Delta_k = k - k'$, that is

$$u[k] = -K_d x[k - \Delta_k].$$

Furthermore, we also introduce the *worst update freshness* Δ_{max}, that is the maximum number of aging steps for the active control value.

The proposed analysis considers each k-th control window (i.e., $[kT_i, (k+1)T_i)$). Due to the delay D_i in the generation of the actuation, two control outputs can exist within such windows, each with a corresponding update freshness. In the sub-window $[kT_i, kT_i + D_i)$ the control output is equal to the one generated in the previous control window, i.e., $u[k-1]$. Conversely, the control output in the window following the actuation update $[kT_i + D_i, (k+1)T_i)$ is equal to $u[k]$. Taking into account the update freshness, the two control outputs within the k-th control window are defined as

$$u[k-1] = -K_d x[k - 1 - \Delta_{k-1}] \tag{7}$$

$$u[k] = -K_d x[k - \Delta_k]. \tag{8}$$

For example, considering the example of Figure 2, the task instance released at time kT_i completes its execution within the deadline, and the actuation value $u[k]$ after the deadline is equal to $-K_d x[k]$ with update freshness 0. As the next deadline is missed, $u[k+1]$ is not updated and remains equal to $-K_d x[k]$, thus the value of the update freshness Δ_{k+1} is now equal to 1.

The next sections will show how to compute the sequences in time of the update freshness values Δ_{k-1} and Δ_k (also denoted as *update freshness pairs*) for all the possible H/M sequences that satisfy an (m, K) constraint. The possible time traces of the freshness pairs are represented using a state machine, where each node is described by a pair (Δ_{k-1}, Δ_k). For the sake of clarity, we will refer to the state machine defining the update freshness as *F-state machine*, and its vertexes as *F-states.* The F-state evolution rules for the job killed and job continued strategies need to be defined to construct the F-state machines.

5.1.1 Job Killed

Under this policy, whenever the job executing in the k-th control window misses its deadline, the actuator uses the previous value for the control output $u[k]$, consequently increasing the update freshness Δ_k. Hence, the definition of the F-state evolution rule for the update freshness follows.

Job killed: F-state evolution rule. Consider the k-th control window, characterized by the pair (Δ_{k-1}, Δ_k). The update freshness pair of the $(k+1)$-th control window is:

- $(\Delta_k, 0)$, if the $(k+1)$-th job of the control task hits its deadline;
- $(\Delta_k, \Delta_k + 1)$, otherwise (deadline miss).

The above rule comes from the following rationale. If a deadline hit occurs in the $(k+1)$-th control window, then the corresponding control output will use a fresh value, i.e., $\Delta_{k+1} = 0$. Otherwise, the control output of the k-th control window will also be used in the next window, with an increase of the update freshness. Therefore, the update freshness of $u[k+1]$ is given by the update freshness of $u[k]$ plus one period, i.e., $\Delta_{k+1} = \Delta_k + 1$.

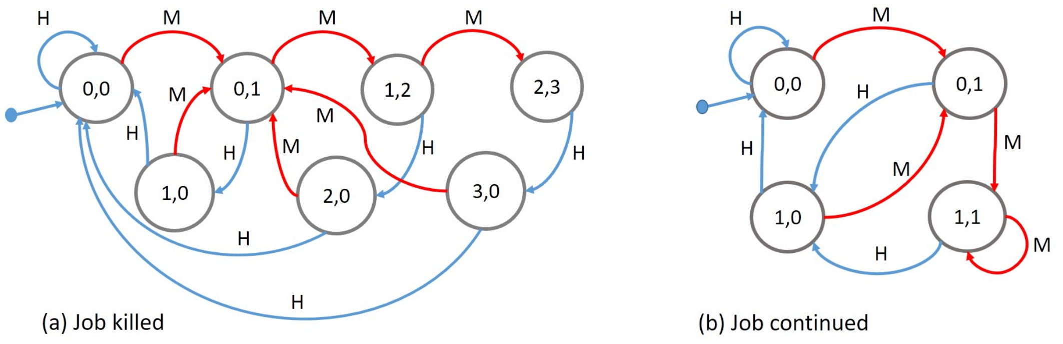


Figure 4 State machines expressing the evolution of the update freshness pair (Δ_p, Δ_c). The one related to the job killed strategy corresponds to the case with $\Delta_{max} = 3$.

Given a bound M^{max} on the number of consecutive deadline misses, it is possible to bound the maximum update freshness experienced in a control window, that is $\Delta_{max} = M^{max}$. Finally, it is important to observe that the above rule is independent of the assumption stated in Equation (6) that bounds the worst-case response time of the control task. In fact, under the job killed strategy, the response-time is always implicitly bounded by the relative deadline D_i.

5.1.2 Job Continued

Under the assumption of Equation (6), the evolution of the update freshness under the job continued strategy is determined by the following rule.

Job continued: F-state evolution rule. Consider the k-th control window, characterized by the pair (Δ_{k-1}, Δ_k). The update freshness pair of the $(k+1)$-th control window is:

- $(\Delta_k, 0)$, if the $(k+1)$-th job of the control task hits its deadline;
- $(\Delta_k, 1)$, otherwise (deadline miss).

Like for the job killed strategy, when the deadline is hit the next control output will dispose of a fresh value, i.e., $\Delta_{k+1} = 0$. On the other hand, if the deadline in the $(k+1)$-th control window is missed, the corresponding control output will be equal to $u[k]$. By Equation (6), a pending job of the control task cannot span more than two consecutive control windows. As a consequence, the state sensed at the beginning of the k-th control window, i.e., time kT_k, will be used at most for producing the $(k+1)$-th control output, which implies that the update freshness is implicitly bounded by one, i.e., $\Delta_{max} = 1$ (independent of M^{max}). Note that, despite this advantage with respect to the job killed strategy, the number of deadline misses under the job continued strategy can significantly increase because of self-pushing [34].

5.2 Constructing the State Machines for the Update Freshness

Given the F-state update rules introduced above, this section shows how to construct a F-state machine that describes the possible evolutions of the update freshness pairs. Each F-state machine is defined by a set of vertexes V, where each vertex is tagged with a freshness pair, and a set of directed edges E connecting the vertexes. An edge $e \in E$ is defined as a triplet consisting in the source vertex, the destination vertex, and a label that states if the vertex is taken when a deadline is hit or miss. For instance, $e = ((0,0),(0,1),M)$ is an edge

Algorithm 1 Construction of state machines for the update freshness.

```
1: global $\mathcal{A} = \{(0,0)\}$
2: global V = $\{(0,0)\}$
3: global E = $\{\}$
4:
5: function Entry_Point( )
6:     explore( $(0,0)$, H )
7:     explore( $(0,0)$, M )
8: end function
9:
10: function explore( $(\Delta_p, \Delta_c)$, x )
11:     $(\Delta'_p, \Delta'_c)$ = State evolution rule($(\Delta_p, \Delta_c)$, x)
12:     V = V $\cup$ { $(\Delta'_p, \Delta'_c)$ }
13:     E = E $\cup$ { ($(\Delta_p, \Delta_c)$, $(\Delta'_p, \Delta'_c)$, x) }
14:     if $(\Delta'_p, \Delta'_c) \notin \mathcal{A}$ then
15:         $\mathcal{A} = \mathcal{A} \cup$ { $(\Delta'_p, \Delta'_c)$ }
16:         explore( $(\Delta'_p, \Delta'_c)$, H )
17:         if $\Delta'_c < M^{max}$ then
18:             explore( $(\Delta'_p, \Delta'_c)$, M )
19:         end if
20:     end if
21: end function
```

that connects an F-state with update freshness defined by the pair $(0, 0)$ to another vertex corresponding to the pair $(0, 1)$ to represent the F-state evolution after a deadline miss.

These F-state machines are characterized by the following two properties: **(i)** the same update freshness pair can be obtained for different values of k (i.e., different control windows may have the same update freshness), and **(ii)** the evolution of the update freshness pair is only dependent on the immediately preceding pair (Δ_{k-1}, Δ_k). For this reason, when needed, the following short notation is adopted to get rid of the index k: Δ_c denotes the *current* update freshness, i.e., $\Delta_c = \Delta_k$ for the k-th control window; and Δ_p denotes the *previous* update freshness, i.e., $\Delta_p = \Delta_{k-1}$.

Algorithm 1 reports the pseudocode to generate the F-state machines. The algorithm exploits the recursive procedure explore that **(i)** computes the next state (Δ'_p, Δ'_c) by means of a state evolution rule given a deadline hit ($x = H$) or miss ($x = M$), **(ii)** adds and connects the new node to the F-state machine, and **(iii)** finally opens two recursive branches related to a deadline hit or miss, respectively. The algorithm termination is guaranteed by keeping track of the previously-visited states in the set $\mathcal{A}$ and by the bound Δ_{max}, both limiting the opening of recursive branches. Two illustrations of the resulting F-state machines are reported in Figure 4. It is worth noting that, given the strategy to handle the deadline misses and the bound Δ_{max}, such F-state machines are fixed and independent of the parameters of the control tasks and the control plant. As a consequence, they provide a very general model to study the evolution of the update freshness.

6 Computing State Trajectories

In the previous section, we defined the relation between H/M sequences and the freshness of the control update. The next step requires combining the update freshness parameters with the information coming from the plant model, which is used to compute the update matrices of the plant state for each step of an H/M sequence. This leads to the definition of a chain of matrices that is then used to extract the state trajectory of the plant (subject to the control).

6.1 State Update Function and Stability Properties

Starting from Equation (2) and combining it with Equations (7) and (8), we derive the state update of the system with an arbitrary freshness pair (Δ_p, Δ_c). The resulting state equation can be rewritten as:

$$x[k+1] = A_d x[k] - B_{d1} K_d x[k-1-\Delta_p] - B_{d2} K_d x[k-\Delta_c] \tag{9}$$

In order to achieve a compact representation of the above equation for different freshness pairs (Δ_p, Δ_c), we introduce the *augmented plant state vector* $\xi[k]$ as

$$\xi[k] = \big[x[k];\, x[k-1];\, \cdots\, x[k-\Delta_{max}-1]\big], \tag{10}$$

which contains the state values of the last $\Delta_{max}+2$ control steps, i.e., from $x[k]$ to $x[k-\Delta_{max}-1]$. Note that $x[k-\Delta_{max}-1]$ is the last possible value of the state considered in Equation (9). Then, by leveraging this augmented state, it is possible to rewrite the state update function of the control system in Equation (9) as follows

$$\xi[k+1] = \mathbf{\Phi}(\Delta_p, \Delta_c)\xi[k]. \tag{11}$$

Here ,$\mathbf{\Phi}(\Delta_p, \Delta_c)$ is the *state update matrix*, which is defined as

$$\mathbf{\Phi}(\Delta_p, \Delta_c) = \begin{bmatrix} A_d & \cdots & -B_{d2}K_d & \cdots & -B_{d1}K_d & \cdots \\ \mathbf{I}_n & \mathbf{0}_n & \cdots & \cdots & \cdots & \cdots \\ \mathbf{0}_n & \mathbf{I}_n & \mathbf{0}_n & \cdots & \cdots & \cdots \\ \vdots & \cdots & \ddots & \ddots & \cdots & \cdots \end{bmatrix}, \tag{12}$$

where $\mathbf{0}_n$ and $\mathbf{I}_n$ are square matrices of zeros and the identity matrix, respectively, with dimension n (n denotes the size of the state space, i.e., $x[k] \in \mathbb{R}^n$). $\mathbf{\Phi}(\Delta_p, \Delta_c)$ is square with dimension $n \cdot (\Delta_{max}+2)$. The definition of $\mathbf{\Phi}(\Delta_p, \Delta_c)$ in Equation (12) is not a direct function of Δ_p and Δ_c: rather, Δ_p and Δ_c determine the position of the blocks $-B_{d1}K_d$ and $-B_{d2}K_d$. The value Δ_{max} determines the matrix size.

Using the state machines for the update freshness defined in the previous section, it is possible to assign each vertex (by means of the corresponding pair (Δ_p, Δ_c)) with a matrix $\mathbf{\Phi}(\Delta_p, \Delta_c)$. For instance, the matrices for the state machine when the job-continue strategy is used (reported in Figure 4(b)) are:

$$\mathbf{\Phi}(0,0) = \begin{bmatrix} A_d - B_{d2}K_d & -B_{d1}K_d & \mathbf{0}_n \\ \mathbf{I}_n & \mathbf{0}_n & \mathbf{0}_n \\ \mathbf{0}_n & \mathbf{I}_n & \mathbf{0}_n \end{bmatrix} \tag{13}$$

$$\mathbf{\Phi}(0,1) = \begin{bmatrix} A_d & -(B_{d1}+B_{d2})K_d & \mathbf{0}_n \\ \mathbf{I}_n & \mathbf{0}_n & \mathbf{0}_n \\ \mathbf{0}_n & \mathbf{I}_n & \mathbf{0}_n \end{bmatrix} \tag{14}$$

$$\mathbf{\Phi}(1,0) = \begin{bmatrix} A_d - B_{d2}K_d & \mathbf{0}_n & -B_{d1}K_d \\ \mathbf{I}_n & \mathbf{0}_n & \mathbf{0}_n \\ \mathbf{0}_n & \mathbf{I}_n & \mathbf{0}_n \end{bmatrix} \tag{15}$$

$$\mathbf{\Phi}(1,1) = \begin{bmatrix} A_d & -B_{d2}K_d & -B_{d1}K_d \\ \mathbf{I}_n & \mathbf{0}_n & \mathbf{0}_n \\ \mathbf{0}_n & \mathbf{I}_n & \mathbf{0}_n \end{bmatrix}. \tag{16}$$

Each such matrix corresponds to a possible mode in which the control system can operate, and such modes can be reached as a function of the pattern defined by an H/M sequence depending on the state machine that defines the update freshness. Furthermore, not all the possible transitions between such modes are actually possible: only the subset of H/M sequences that belong to the satisfaction set of Γ is possible, limiting the possible paths in the state machine of the update freshness. In this view, the control system under analysis can be considered as a particular case of a *constrained switched linear system* [29].

Equation (11) is particularly useful when studying the *stability* of the controlled system. According to the choice of K_d discussed in Section 3, the system is stable when no deadline misses occur (i.e., the dynamic related to matrix $\mathbf{\Phi}(0,0)$ is stable). However, no stability properties are guaranteed for the other operating modes. A performance analysis is meaningless for an unstable system, therefore all possible mode switches must lead to a stable behavior. An interesting approach for defining a controller that is stable for each possible H/M sequences (even if related to a simplified model with respect to the one presented here) is presented in [19]. However, in order to dispose of a bound on the number of steps of the H/M sequences under analysis, we require the stronger condition of *exponentially stability*: given the matrices $\mathbf{\Phi}(\Delta_p, \Delta_c)$, this property can be verified with the technique presented by Yu and Zhang in [35]. More details are provided in Section 7.2.

6.2 Mapping H/M Sequences to State Trajectories

After assigning a matrix $\mathbf{\Phi}(\Delta_p, \Delta_c)$ to each vertex of the state machine for the update freshness, it is possible to obtain a *sequence* of such matrices given an initial vertex and an H/M sequence. The compact notation $\mathbf{\Phi}_k$ denotes the matrix $\mathbf{\Phi}(\Delta_p, \Delta_c)$ obtained at the k-th step of an arbitrary H/M sequence. If the initial state $\xi[0]$ is known, it is possible to recursively compute the $(k+1)$-th state of the system as follows:

$$\begin{aligned}\xi[k+1] &= \mathbf{\Phi}_k \xi[k] \\ \xi[k] &= \mathbf{\Phi}_{k-1} \xi[k-1] \\ &\dots \\ \xi[1] &= \mathbf{\Phi}_0 \xi[0],\end{aligned}$$

so obtaining the following compact form:

$$\xi[k+1] = \mathbf{\Phi}_k \mathbf{\Phi}_{k-1} \cdots \mathbf{\Phi}_0 \xi[0]. \tag{17}$$

Thus, by simply multiplying a set of matrices $\mathbf{\Phi}_k$, it is possible to compute the corresponding *state trajectory* of the system. By considering each possible valid path in the state machine of the update freshness, it is also possible to compute all state trajectories subject to the weakly-hard constraints of the control task. To practically enable such computations, an initial state (in terms of update freshness) and a given length for H/M sequences are required. The former can be selected as the one characterized by the pair $(0,0)$, which matches the condition at the system startup. A bound on the length of H/M sequences is provided in the next section. Finally, it is worth noting that the initial state $\xi[0]$ contributes to the state trajectory with a fixed proportional constant. This means that the initial state can be treated as a *scaling factor*, while the *shape* of the trajectory is only determined by the sequence of hit and missed deadlines. This consideration is fundamental for the performance analysis that will be addressed in the following section, as it allows a significant reduction of its computational complexity.

7 Assigning Performance Values to H/M sequences

The first objective of this section is to assign a control performance index to each state trajectory, computed for a given H/M sequence (as defined in the previous section). As a result, H/M sequences are associated with a performance value, enabling the computation of the worst-case system performance over all the possible H/M sequences. Then, for the purpose of monitoring the evolution of the performance online, we provide a richer state-based performance model.

7.1 Mapping Trajectories to Performance

The proposed approach can be used with different performance metrics: as a representative case, we focus on the *sum of quadratic error* of the augmented plant state vector, which is formally defined for a given H/M sequence s as

$$P(s) = \sum_{i=0}^{N-1} \xi[i]^T \xi[i], \tag{18}$$

where N is the length of the sequence s. This index is indeed the discrete representation of the widely-adopted *integral of squared error* [14]. Note that our approach is also compatible with other performance indexes. Combining Equation (18) with Equation (17), we obtain the following expanded expression for $P(s)$:

$$\begin{aligned} P(s) &= \sum_{i=0}^{N-1} \xi[i]^T \xi[i] \\ &= \xi[0]^T \Big(\mathbf{I} + \mathbf{\Phi}_0^T \mathbf{\Phi}_0 + \mathbf{\Phi}_0^T \mathbf{\Phi}_1^T \mathbf{\Phi}_1 \mathbf{\Phi}_0 + \ldots + \mathbf{\Phi}_0^T \mathbf{\Phi}_1^T \cdots \mathbf{\Phi}_{N-1}^T \mathbf{\Phi}_{N-1} \cdots \mathbf{\Phi}_1 \mathbf{\Phi}_0\Big) \xi[0] \\ &= \xi[0]^T \mathbf{\Psi}(s) \xi[0] \end{aligned} \tag{19}$$

The resulting matrix $\mathbf{\Psi}(s)$ is then a function of the ordered system states during the trajectory determined by the sequence s.

Likewise Equation (17), the initial state $\xi[0]$ can be treated as a proportional constant value also for the performance $P(s)$. This means that an order between the performance of different H/M sequences can be defined independently of the initial state. Motivated by this, we select the norm of $\mathbf{\Psi}(s)$, defined as $\Pi(s) = ||\mathbf{\Psi}(s)||_2$, as a scalar performance index.

7.2 Bounding the Number of Steps

To bound the complexity of the analysis, a *finite horizon* approach is selected, with N steps for the considered H/M sequences. Theoretically speaking, an exact performance analysis should consider an infinite horizon: however, given an arbitrary small error, a limited number of steps is sufficient to obtain a performance measurement.

As a lower bound, the value of N must never be less than the maximum window size K of the weakly-hard constraints, in order to avoid pathological cases where the constraint is not even completely defined on an input sequence. From a control perspective, it is important that the resulting performance analysis is applied to a sufficiently long interval of control steps, such that meaningful information can be extracted. For instance, the number of control steps N should be sufficient to include the settling time of the step response of the system. In general, the interval size must be carefully chosen to include the step response of *all* the possible H/M sequences. As the global switching system is exponentially stable by hypothesis, following the results presented in [35], all the possible response dynamics of the plant state can be upper-bounded by an exponential function. Thus, a safe upper bound for N can be computed as the number of steps for which the exponential envelope converges within a given error of the reference.

7.3 Worst-Case Performance

The worst-case performance can be computed as a function of the weakly-hard constraints of the control task. In this paper, the performance is a quadratic error metric, and is therefore formally a cost; for a true performance (positive) metric, the maximum in (20) (in the following definition) should become a minimum.

▶ **Definition 4** (Worst-Case Performance). Given a set Γ of weakly-hard constraints for the control task and a number of steps N, the worst-case performance for Γ is the *maximum* value of $\Pi(s)$ provided by all sequences in the satisfaction set of Γ with length N, i.e.,

$$WCP(\Gamma, N) = \max_{s \in \mathcal{S}_N} \Pi(s). \tag{20}$$

The normalized worst-case performance $WCPn(\Gamma, N)$ is computed with respect to the sequence s_H of all hits and is defined as

$$WCPn(\Gamma, N) = \frac{WCP(\Gamma, N)}{\Pi(s_H)}. \tag{21}$$

Interestingly, the worst-case performance is not necessarily associated with the sequence that has the largest number of deadline misses in the satisfaction set of Γ. Also, the proposed analysis allows to discern the effects on the performance of the *order* of deadline misses in the H/M sequence. Deadline misses in the early steps of the sequence s typically lead to worse performance with respect to deadline misses that occur late in the sequence. This behavior can intuitively be explained by the results derived by Yu and Zhang in [35]. In Equation (17), the j-th extended state $\xi[j]$ is computed as the result of the multiplication of j matrices $\mathbf{\Phi}_j$. Early misses contribute to the first terms, e.g., with only two or four matrices in the product. Since some matrices related to steps with missed deadlines can determine an unstable steady-state system (i.e., operating modes in which the system can diverge) these terms can result in a system that temporally tends to diverge from the control reference. Otherwise, since a switching control system is stable when the norm of the asymptotic multiplication of such matrices tends to zero, it is expected that longer terms (such as those towards the end of Eq. (17) representing late events in the hit/miss sequence) tend to be inherently more stable and contribute less to the dynamic (the unstable modes are dampened due to the asymptotic stability).

7.4 State-Based Representation of the Performance

The last step of our analysis is creating a *state machine* where every vertex represents the performance value of a H/M sequence of N control steps. For the sake of clarity, the states of this machine will be denoted as *P-states.* The edges between vertexes represent one step in the H/M sequence (hence a hit or missed deadline).

In the resulting P-state machine, it is possible to identify sets of sequences that lead to various degrees of *acceptable* or *poor* performance. Performance monitors can be inserted into the system for controlling (and possibly avoiding) transitions to a poor performance sequence. This can be obtained by increasing the priority of the control task, shedding higher

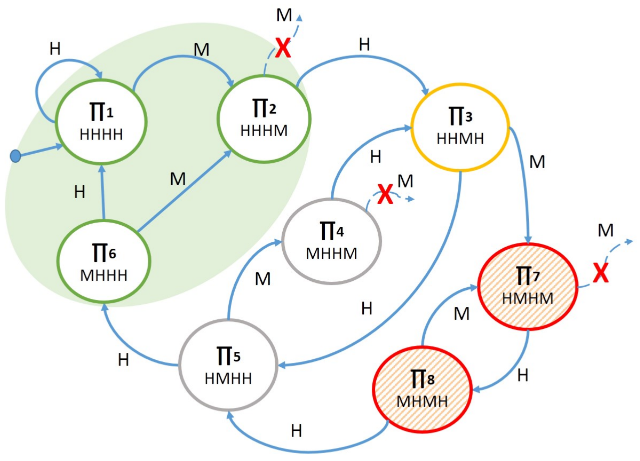


Figure 5 State machine representation of performance indexes for H/M sequences, for the case with $\Gamma = (1, 2)$ and $N = 4$. The vertexes in green are the best values for performance, the red ones are undesirable values and the orange one is a sequence that may lead to an undesirable behavior. The edges marked with a red X are unfeasible transitions.

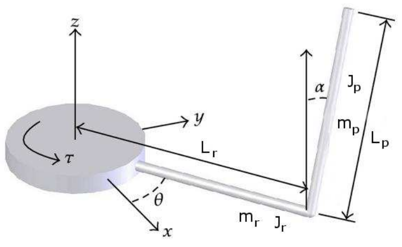


Figure 6 Scheme of the Furuta inverted pendulum, as the plant model used in the case of study.

priority load, or switching to a less computationally expensive implementation. Interesting information can also be extracted by analyzing the number of hits that are needed to recover from sequences with *poor* performance. If an unacceptable sequence is found, the design of the system must be changed in order to compensate this behavior. The solution can be, e.g., changing the control law parameters, or modifying some implementation strategies, such as the task implementation or the deadline miss handling policy.

The resulting P-state machine is a powerful tool for the performance analysis or possibly for the synthesis of runtime monitors controlling the evolution of the system and triggering recovery actions when needed. An illustrative example of the P-state machine for performance analysis is shown in Figure 5. Some transitions are marked with a red X and correspond to transitions that should never occur because of the results of the weakly hard analysis. The vertexes are colored differently following their performance values and critical P-states leading to poor performance can be easily identified.

8 Case Study

In this section, the proposed methodology is applied to a case study, consisting of a small rotary inverted pendulum (Furuta's Pendulum [13]) as illustrated in Figure 6. The objective of the control is to keep the pendulum in the upright position, which is an unstable equilibrium.

Table 1 Numerical values for the parameters describing the Furuta pendulum.

Parameters	**Arm**	**Pendulum**	**Units**	**Motor**		**Units**
m	0.4	0.02	kg	k_t	0.768	$N \cdot m/A$
L	0.1	0.2	m	k_e	0.768	$V \cdot s/rad$
l	0.06	0.1	m	R_m	3.3	Ω
J	0.0018	$2.67 \cdot 10^{-4}$	$kg \cdot m^2$	J_m	0.0015	$kg \cdot m^2$
b	0.0004	0.005	$kg/(m \cdot s)$			

The geometry and mass properties of the system are described by the set of parameters (L_p, l_p, J_p, m_p) and (L_r, l_r, J_r, m_r), which are the length, barycenter position, inertia and mass of pendulum stick (index p) and arm (index r), respectively. The viscous damping on the arm and pendulum joints have coefficients b_r and b_p, respectively. The state vector of the considered plant is defined as $x = [\theta, \alpha, \dot{\theta}, \dot{\alpha}]^T$, where θ is the angular position of the arm and α is the position of the pendulum with respect to the vertical. The controlled input voltage drives an electric motor on the arm joint, producing a torque Λ. The equation relating the voltage and the generated torque is $\Lambda = (k_t(V_m - k_m\dot{\theta}))/R_m$, where k_t is the torque constant of the motor, V_m is the voltage applied, k_e is the back e.m.f constant, and R_m is the resistance of the motor winding. The balancing control problem is considered, using a model linearized in the neighborhood of the upward position of the pendulum arm. Under the defined state vector, the following Linear Time Invariant (LTI) model [9] is obtained:

$$A_c = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{m_p^2 L_p^2 L_r g}{\Upsilon} & \frac{J_p(-b_m - b_r)}{\Upsilon} & \frac{-m_p L_p L_r b_p}{\Upsilon} \\ 0 & \frac{(J_r + m_p L_r^2) g m_p L_p}{\Upsilon} & \frac{m_p L_r L_p(-b_m - b_r)}{\Upsilon} & \frac{-(J_r + m_p L_r^2) b_p}{\Upsilon} \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 0 \\ \frac{J_p k_t}{R_m} \\ \frac{m_p L_r L_p k_t}{R_m} \end{bmatrix}$$

where $b_m = k_e k_t / R_m$ and $\Upsilon = (J_r + m_p L_r^2) J_p - m_p^2 L_r^2 L_p^2$.

Performing the discretization of the system, following Equations (3) and (4), using a sample time $T_i = 0.1s$ and deadline $D_i = 0.2T_i$, and substituting the values with ones provided in Table 1, the model can be expressed in the following numerical form:

$$A_d = \begin{bmatrix} 1.0000 & 0.0036 & 0.0188 & -0.0007 \\ 0 & 1.2282 & -0.0332 & 0.0503 \\ 0 & 0.0266 & 0.0081 & -0.0032 \\ 0 & 3.7230 & -0.2448 & 0.2794 \end{bmatrix}, \quad B_{d1} = \begin{bmatrix} 0.0381 \\ 0.0109 \\ 0.0261 \\ -0.1006 \end{bmatrix}, B_{d2} = \begin{bmatrix} 0.0666 \\ 0.0320 \\ 1.2539 \\ 0.4166 \end{bmatrix}$$

The control law used for this example is a stabilizing static feedback of the state with constants $K_d = [-1.4557, 62.8126, -2.0459, 2.7210]$.

Experimental Setup

The tests have been carried out covering all the possible combinations of hits and misses that satisfy different (m, K) constraints, with an analysis horizon of $N = 20$ steps. For the purposes of this case study, this value is sufficient to capture the main features of the dynamics when different weakly-hard constraints are used. Both the job killed and job continued strategies have been considered. The scheduling parameters (m, K) are computed for $K \in [5, 8]$ and $m \in [1, K - 3]$, respectively. In order to study the evolution of the control performance, for each strategy, the corresponding matrices $\mathbf{\Phi}_k$ have been computed.

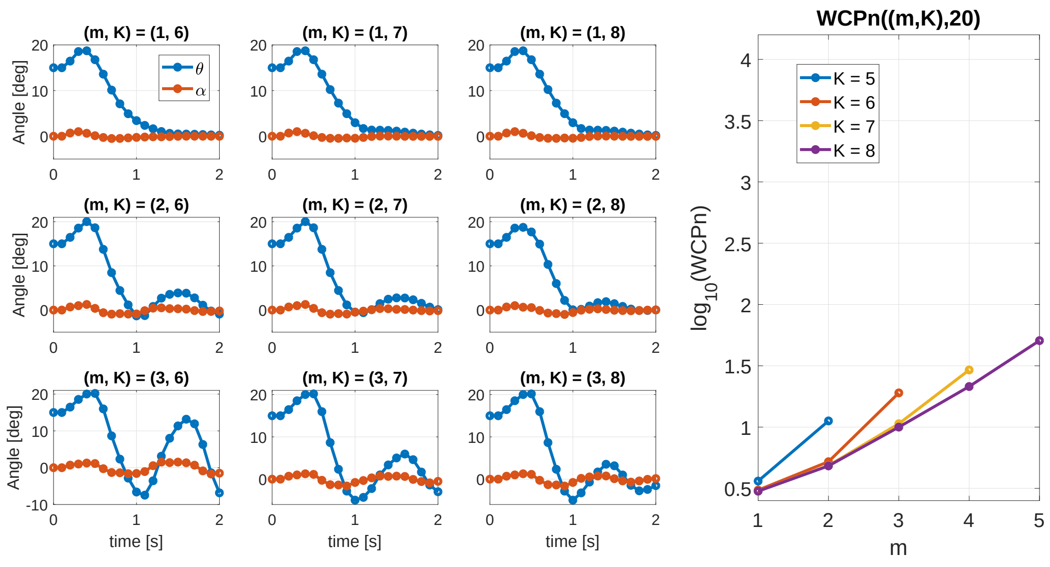


Figure 7 Results related to the **job continued** strategy. *Plots on the left*: state trajectories for the H/M sequence that leads to the worst-case performance under different (m, K) constraints. *Plot on the right*: normalized worst-case performance for the considered configurations.

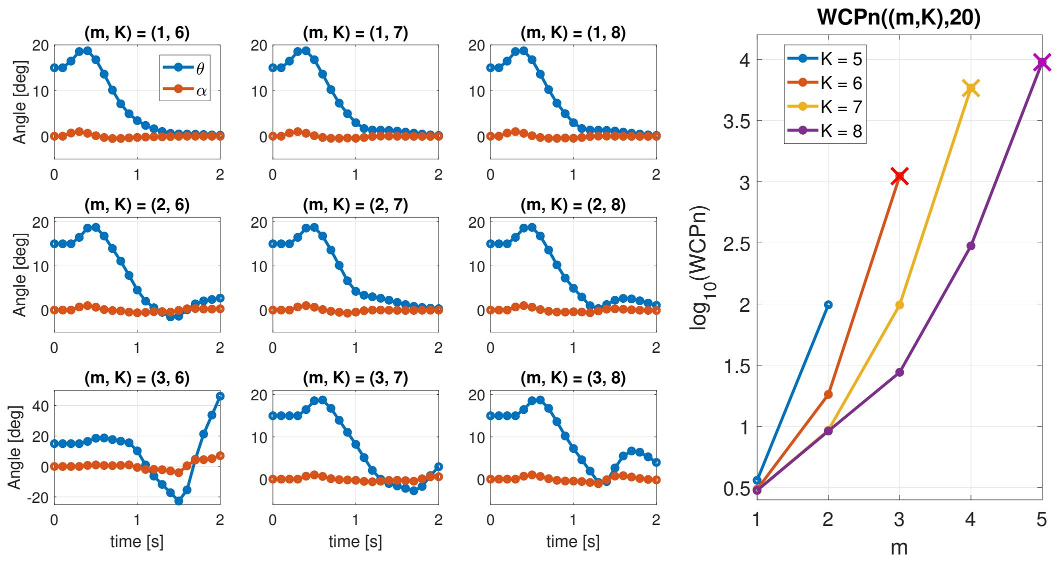
Experimental Results

The experiments produced a big volume of data, thus, for the sake of clarity and compactness, only the most representative results have been represented and discussed here.

By considering different weakly-hard constraints (m, K), Figure 7 reports the trajectories of the system in correspondence to the H/M sequence that originates the worst-case performance. The figure also reports the normalized performance for each of the considered constraints. The plots of the state trajectory (on the left) show that, fixed the window K, the settling time and the width of possible oscillations (of the trajectory) increase as the number of deadline misses m increases. Looking at the plot on the right, it is noticeable that the worst-case performance is monotone with respect to the dimension of the scheduling window K. In particular, note that for the same number of deadline misses, better performance can be obtained for larger values of K. This can be explained by the fact that deadlines are to be placed with a lower density.

Figure 8 shows the experimental results under the same configurations considered in the previous figure but related to the job kill strategy. While it is possible to observe the same monotonic trend of the performance with respect to parameter K, the value of the normalized WCP is worse than the one obtained with the job continued strategy. As a matter of fact, for the same constraint (m, K), the job killed strategy also shows worse trajectories in terms of settling time and oscillations.

Also note that this strategy leads to unstable configurations (e.g., see the case for $(m, K) = (3, 6)$ in the figure), which tend to arise to the larger value of Δ_{max}. The performance value has also been computed for the particular cases of unstable systems only for the purposes of comparison with the job continued strategy of Figure 7. These results pose an interesting observation on the trade-offs between schedulability and performance analysis for the tested system: while the job killed strategy can improve the system schedulability, lowering the computational workload and possibly simplifying schedulability analysis, it tends to provide worse performance. For this reason, we believe that the proposed analysis framework may be valuable when facing with control-scheduling co-design activities.



■ **Figure 8** Results related to the **job killed** strategy. *Plots on the left*: state trajectories for the H/M sequence that leads to the worst-case performance under different (m, K) constraints. *Plot on the right*: normalized worst-case performance for the considered configurations. The elements marked with a "x" refer to unstable configurations.

9 Conclusions

We presented a new methodology to compute a state machine abstraction that allows to relate the performance of a control application to a sequence of deadline hits and misses, subject to weakly-hard constraints, in the execution of a control task updating a control value. We show the possibility of computing the state machine by formal derivation assuming knowledge of the deadline management policy, the LTI system and a simple performance metric. The size of our state representation is constrained by leveraging worst case timing analysis and assuming a finite time horizon.

Future work will include consideration of additional performance metrics and the possible use of simulation techniques to compute the state abstraction when an analytical derivation is not possible.

References

1 Amir Aminifar, Petru Eles, Zebo Peng, and Anton Cervin. Control-quality driven design of cyber-physical systems with robustness guarantees. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1093–1098. EDA Consortium, 2013.

2 Amir Aminifar, Soheil Samii, Petru Eles, Zebo Peng, and Anton Cervin. Designing high-quality embedded control systems with guaranteed stability. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 283–292, 2012.

3 Karl-Erik Årzén, Anton Cervin, Johan Eker, and Lui Sha. An introduction to control and scheduling co-design. In *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, volume 5, pages 4865–4870, 2000.

4 Karl Johan Åström and Björn Wittenmark. *Computer-controlled systems: theory and design.* Prentice-Hall, 1997.

5 Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.

6 Rainer Blind and Frank Allgöwer. Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pages 7510–7515. IEEE, 2015.

7 Tobias Bund and Frank Slomka. Controller/platform co-design of networked control systems based on density functions. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 11–14, 2014.

8 Tobias Bund and Frank Slomka. Worst-case performance validation of safety-critical control systems with dropped samples. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 319–326, 2015.

9 Benjamin Seth Cazzolato and Zebb Prime. On the dynamics of the furuta pendulum. *Journal of Control Science and Engineering*, 2011:3, 2011.

10 A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. E. Arzen. How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime. *IEEE Control Syst. Mag*, 23 (3):16–30, 2003.

11 F. Cremona, M. Morelli, and M. Di Natale. Tres: A modular representation of schedulers, tasks, and messages to control simulations in simulink. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1940–1947. ACM, 2015.

12 Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 53–62. IEEE, 2014.

13 Katsuhisa Furuta, Masaki Yamakita, and Seiichi Kobayashi. Swing up control of inverted pendulum. In *Industrial Electronics, Control and Instrumentation, 1991. Proceedings. IECON'91., 1991 International Conference on*, pages 2193–2198. IEEE, 1991.

14 Madan Gopal. *Digital control engineering*. New Age International, 1988.

15 Dip Goswami, Reinhard Schneider, and Samarjit Chakraborty. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 225–230. IEEE Press, 2011.

16 Zain AH Hammadeh, Sophie Quinton, and Rolf Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *Proceedings of the 14th International Conference on Embedded Software*, page 10. ACM, 2014.

17 Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

18 Pavel Krcal and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *proceedings of 10th International Conference, TACAS'04*. IEEE, 2004.

19 Steffen Linsenmayer and Frank Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *Decision and Control (CDC), 2017 IEEE 56th Annual Conference on*, pages 4765–4770. IEEE, 2017.

20 M. M Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. In *IEEE Transactions on Computers*, 1995.

21 Tsutomu Mita. Optimal digital feedback control systems counting computation time of control laws. In *Decision and Control, 1984. The 23rd IEEE Conference on*, volume 23, pages 520–525. IEEE, 1984.

22 Marieke Posthumus-cloosterman, Cover Design, Gerda Cloosterman, and Paul Verspaget. *Control over communication networks: Modelling, analysis and synthesis*. PhD thesis, University of Eindhoven, 2008.

23 Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 515–520. EDA Consortium, 2012.

24 Parameswaran Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

25 Kang G Shin and Hagbae Kim. Derivation and application of hard deadlines for real-time control systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6):1403–1413, 1992.

26 Damoon Soudbakhsh, Linh TX Phan, Anuradha M Annaswamy, and Oleg Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 2016.

27 Damoon Soudbakhsh, Linh TX Phan, Oleg Sokolsky, Insup Lee, and Anuradha Annaswamy. Co-design of control and platform with dropped signals. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 129–140, 2013.

28 Martin Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. In *Real-Time Systems In Mechatronic Applications*, pages 3–35. Springer, 1998.

29 Eelco P van Horssen, AR Baghban Behrouzian, Dip Goswami, Duarte Antunes, Twan Basten, and WPMH Heemels. Performance analysis and controller improvement for linear systems with (m, k)-firm data losses. In *Control Conference (ECC), 2016 European*, pages 2571–2577. IEEE, 2016.

30 Wenbo Xu, Zain AH Hammadeh, Alexander Kröller, Rolf Ernst, and Sophie Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 247–256. IEEE, 2015.

31 Yang Xu, Karl-Erik Årzén, Enrico Bini, and Anton Cervin. Response time driven design of control systems. *IFAC Proceedings Volumes*, 47(3):6098–6104, 2014.

32 Yang Xu, Karl-Erik Årzén, Anton Cervin, Enrico Bini, and Bogdan Tanasa. Exploiting job response-time information in the co-design of real-time control systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE 21st International Conference on*, pages 247–256, 2015.

33 Tatsuya Yoshimoto and Toshimitsu Ushio. Optimal arbitration of control tasks by job skipping in cyber-physical systems. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, pages 55–64. IEEE Computer Society, 2011.

34 S. Youcheng and M. Di Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. In *EMSOFT Conference, Seoul, Korea.* IEEE, October, 13-17, 2017.

35 Qiang Yu and Xudong Zhao. Stability analysis of discrete-time switched linear systems with unstable subsystems. *Applied Mathematics and Computation*, 273:718–725, 2016.

Intractability Issues in Mixed-Criticality Scheduling

Kunal Agrawal[1]
Washington University in St. Louis,
St. Louis, MO, USA
kunal@wustl.edu
https://orcid.org/0000-0001-5882-6647

Sanjoy Baruah[2]
Washington University in St. Louis
St. Louis, MO, USA
baruah@wustl.edu
https://orcid.org/0000-0002-4541-3445

Abstract

In seeking to develop mixed-criticality scheduling algorithms, one encounters challenges arising from two sources. First, mixed-criticality scheduling is an inherently an on-line problem in that scheduling decisions must be made without access to all the information that is needed to make such decisions optimally – such information is only revealed over time. Second, many fundamental mixed-criticality schedulability analysis problems are computationally intractable – NP-hard in the strong sense – but we desire to solve these problems using algorithms with polynomial or pseudo-polynomial running time. While these two aspects of intractability are traditionally studied separately in the theoretical computer science literature, they have been considered in an integrated fashion in mixed-criticality scheduling theory. In this work we seek to separate out the effects of being inherently on-line, and being computationally intractable, on the overall intractability of mixed-criticality scheduling problems. Speedup factor is widely used as quantitative metric of the effectiveness of mixed-criticality scheduling algorithms; there has recently been a bit of a debate regarding the appropriateness of doing so. We provide here some additional perspective on this matter: we seek to better understand its appropriateness as well as its limitations in this regard by examining separately how the on-line nature of some mixed-criticality problems, and their computational complexity, contribute to the speedup factors of two widely-studied mixed-criticality scheduling algorithms.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases mixed-criticality scheduling, speedup factor, competitive ratio, approximation ratio, NP-completeness results, sporadic tasks

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.11

1 Introduction

In the decade or so since it was first proposed as a formal model for representing mixed-criticality workloads, the Vestal model [15] has been the focus of a large body of scheduling-theoretic research (see [6] for a survey). One reasonable high-level "meta" conclusion that may be drawn from this research is that it is remarkably challenging to come up with general

[1] Supported in part by NSF grants CCF-1337218 and CCF-173387
[2] Supported in part by NSF Grants CNS 1409175, CPS 1446631, and CNS 1563845

algorithms for mixed-criticality scheduling that are efficient in terms of both running time and resource utilization; although impossibility and intractability results abound, relatively few efficient algorithms have been derived.

In this paper we report some of the findings of our ongoing efforts at obtaining a comprehensive understanding of the phenomenon of mixed-criticality scheduling, and seeking to explain why it is so difficult to schedule mixed-criticality systems efficiently. We separate out two distinct sources of intractability in mixed-criticality scheduling: (i) mixed-criticality scheduling is inherently an *on-line* problem, in which information needed to make good scheduling decisions is only revealed gradually during run-time; and (2) even ignoring this on-line nature, some basic mixed-criticality scheduling problems are *computationally intractable*. (For instance, it is known [5, Theorem 1] that determining whether a given collection of independent mixed-criticality jobs is schedulable is NP-hard in the strong sense.)

Now these two sources of intractability – being an on-line problem and being computationally intractable – have traditionally been considered separately in the theoretical computer science community:

1. The *competitive ratio/ factor* metric is used to quantify the sub-optimality of an on-line algorithm vis-à-vis an optimal clairvoyant one that is assumed to have complete knowledge about run-time behavior prior to making any scheduling decisions.
   For example, it is known that given a cache memory of k pages, the Least-Recently Used (LRU) paging algorithm is k*-competitive* [12] – upon some sequences of page requests it may experience up to k times as many page-faults as on optimal clairvoyant algorithm would.
2. In contrast, the *approximation ratio/ factor* metric quantifies the performance degradation resulting from using a polynomial-time algorithm for solving an NP-hard problem approximately.
   It is known, for example, that while the problem of scheduling a directed acyclic graph on a multiprocessor platform to minimize the makespan is NP-hard in the strong sense [14], List Scheduling [8] is a 2-approximation algorithm for this problem that runs in polynomial time: if a given directed acyclic graph can be scheduled optimally upon a specified multiprocessor platform to have makespan M, then List Scheduling will schedule it to have a makespan $< 2M$.

It is widely recognized in the theoretical computer science community that these two sources of intractability are fundamentally different from one another; however in the mixed-criticality scheduling theory literature a single metric – the speedup factor – tends to be used to quantify the effectiveness of mixed-criticality scheduling algorithms in dealing with both sources of intractability. As used in the mixed-criticality scheduling literature, the *speedup factor* metric of an algorithm A is the minimum multiplicative factor by which the speed of a computing platform must be increased in order that A be able to schedule any problem instance that is schedulable by some optimal clairvoyant scheduler. Used in this manner, it is particularly appropriate for quantifying the penalty arising from the on-line nature of mixed-criticality scheduling, but what about the penalty that may arise from computational intractability issues? This fundamental question motivated some lively discussions during a 2017 Dagstuhl Seminar on mixed-criticality systems,[3] regarding the benefits and drawbacks of using speedup factor as a quantitative metric to evaluate mixed-criticality scheduling algorithms. The opinion was expressed that since it compares algorithm A to an optimal *clairvoyant* scheduler,

[3] Dagstuhl Seminar 17131: *Mixed Criticality on Multicore/ Manycore Platforms*, March 26–31, 2017. `http://www.dagstuhl.de/17131`.

speedup factor is not very useful in comparing different on-line (non-clairvoyant) algorithms; a meaningful metric would not empower the adversary against whom each algorithm is being compared with as extreme a power as clairvoyance. The opposing opinion was that speedup factors as used in the mixed-criticality scheduling literature are merely an instantiation of the concept of the competitive factor metric [12], that has long been considered the gold standard for quantifying on-line algorithms – as a specific example, competitive factors continue to be widely used for justifying the choice of paging algorithms, despite the non-existence of clairvoyant paging algorithms. (Some other limitations of speedup factor as a metric for scheduling algorithms have been elaborated upon in [7]; rather than elaborate upon these limitations here we encourage the interested reader to peruse [7] since a discussion of these limitations is somewhat orthogonal to our prime objective here of niggling out the different effects of non-clairvoyance and computational complexity.)

This research. In this research, we attempt to obtain a better understanding of the role that speedup factor plays in characterizing mixed-criticality scheduling algorithms. We focus upon three widely-studied uniprocessor mixed-criticality scheduling problems that have previously been quantified with speedup factors: (i) scheduling of collections of independent dual-criticality *jobs*; (ii) scheduling of collections of independent dual-criticality *implicit-deadline periodic tasks*; and (iii) scheduling of collections of independent dual-criticality *implicit-deadline* ***sporadic*** *tasks*. For each problem, we consider three forms of schedulability:

1. Clairvoyant schedulability: Given a dual-criticality instance I, can I be scheduled correctly[4] by a clairvoyant scheduling algorithm?
2. MC schedulability: Given a dual-criticality instance I, can I be scheduled correctly by an on-line (non-clairvoyant) scheduling algorithm?
3. A-schedulability (for some specified scheduling algorithm A): Given a dual-criticality instance I, can I be scheduled correctly by the scheduling algorithm A?
   In this work, the specific algorithms A that we consider are OCBP [4] for the job-scheduling problem, and EDF-VD [3] for the periodic and sporadic task-scheduling problems.

Computational complexity. The computational (in)tractability status of each of these schedulability problems is known to be as follows

- For each of the three problems that we study it is known (or can be easily deduced from prior results) that clairvoyant schedulability, as well as the respective A-schedulabilities (i.e., OCBP-schedulability for jobs; EDF-VD-schedulability for periodic and sporadic tasks) can be determined efficiently in polynomial time.
- What about MC-schedulability? The state of knowledge here is rather more sparse:
  - It has previously [5] been shown that determining MC-schedulability for a collection of independent dual-criticality jobs is NP-hard in the strong sense.
  - To our knowledge, no non-trivial prior results are known regarding MC-scedulability of dual-criticality periodic or sporadic implicit-deadline task systems. As one of the major contributions of this paper, we show that *determining MC-schedulability for collections of independent dual-criticality periodic and sporadic implicit-deadline tasks is also NP-hard in the strong sense.*
    The significance of this result for our purposes cannot be over-stated: as is the case for OCBP and the scheduling of independent jobs, it follows that EDF-VD, too, is dealing

[4] Precise definitions of what it means to schedule a dual-criticality system correctly are provided in Section 2.

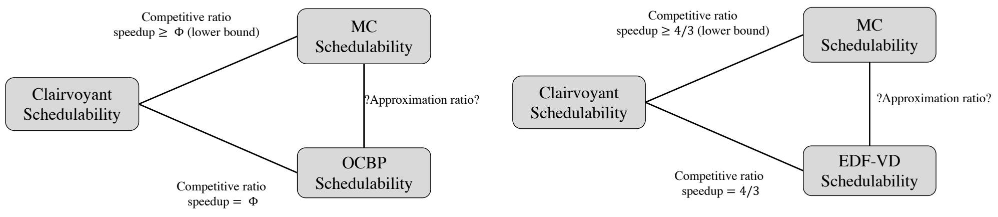


Figure 1 Summarizing the current state of our knowledge regarding speedup bounds for MC-scheduling of dual-criticality independent jobs (left figure) and dual-criticality implicit-deadline periodic & sporadic tasks (right figure). Each figure depicts previously-known lower and upper speedup bounds on the two sources of intractability: the on-line nature of MC-scheduling (given by speedup ratio), and the computational tractability of MC-schedulability analysis (given by approximation ratio).

with both the intractability arising from its non-clairvoyance and the intractability arising from having to solve an NP-complete problem in polynomial time. Hence its sub-optimality (as quantified by its speedup factor), too, can be attributed to two distinct sources.

Non-clairvoyance. The inherent intractability arising from the on-line nature of mixed-criticality scheduling problems has also been studied, and is fairly well understood for our three problems of interest:

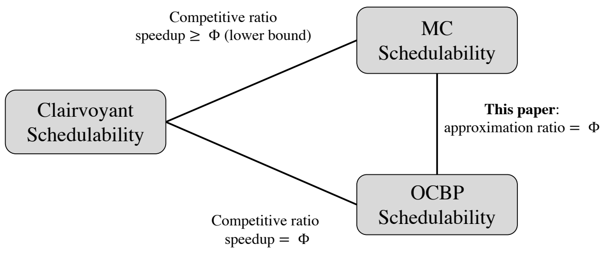
- For collections of independent dual-criticality jobs, it was shown [4] that there are clairvoyant-schedulable instances that are not MC-schedulable without speedup $< \Phi$, where $\Phi = (\sqrt{5}+1)/2 \approx 1.618$ denotes the Golden Ratio:

$$\Phi = \frac{\sqrt{5}+1}{2} \quad \approx \quad 1.618 \tag{1}$$

- For collections of independent dual-criticality implicit-deadline sporadic tasks, it was shown [3] that there are clairvoyant-schedulable instances that are not MC-schedulable without speedup $< 4/3$.
- The proof in [3] is easily adapted from sporadic to periodic tasks, to show that there are clairvoyant-schedulable instances of independent dual-criticality implicit-deadline periodic tasks that are not MC-schedulable without speedup $< 4/3$.

The intractability results described above – computational intractability and loss of performance vis-à-vis clairvoyant schedulability – unfortunately do not fit in neatly with known results concerning specific scheduling algorithms. It is known, for instance, that the polynomial-time algorithm OCBP [4] is able to schedule any clairvoyant-schedulable collection of independent dual-criticality jobs with a speedup Φ. This implies that the speedup ratio Φ that is so widely used to characterize the effectiveness of OCBP as a mixed-criticality scheduling algorithm, is accounted for entirely by the fact that OCBP is solving a problem for which there is a lower bound of Φ on the speedup factor of any non-clairvoyant scheduling algorithm. This result is somewhat paradoxical: the approximation ratio of OCBP vis-à-vis MC-schedulability, arising from the fact that OCBP is only solving this problem approximately (which it inevitably is, since OCBP is a polynomial-time algorithm while determining MC-schedulability for collections of independent dual-criticality jobs is NP-hard in the strong sense) is not accounted for at all – see the left diagram in Figure 1. Similarly, it is known that EDF-VD can schedule any clairvoyant-schedulable collection of independent dual-criticality implicit-deadline periodic or sporadic tasks with a speedup 4/3'rds – this is depicted in the right diagram in Figure 1; this again fails to account for the fact that EDF-VD is solving, in polynomial time, the MC-schedulablity problem for implicit-deadline periodic and sporadic tasks despite our showing, in this paper, that this problem is NP-hard in the strong sense.

One of our major results here is a proof that the *approximation ratio of OCBP* – i.e., its degradation in performance vis-à-vis MC-schedulability – is also quantified by a speedup factor equal to Φ. We show this by synthesizing an independent dual-criticality job instance that is MC-schedulable, but that can only be scheduled by OCBP upon a platform that is Φ times as fast. This immediately yields the interesting conclusion that the speedup factors quantifying (i) the intractability arising from the on-line nature of mixed-criticality scheduling , and (ii) the intractability arising from the computational complexity (NP-hardness) of recognizing MC-schedulability, do **not** "compose" in any meaningful sense: while each is equal to Φ in this case, the speedup factor of OCBP-schedulability vis-à-vis clairvoyant schedulability is also equal to Φ:



Organization. The remainder of this paper is organized as follows. In Section 2 we briefly describe the mixed-criticality workload models we will be using, and define some relevant concepts. In Section 3 we study the problem of scheduling collections of independent dual-criticality jobs. In Section 4 we present our results concerning the scheduling of collections of dual-criticality implicit-deadline recurrent (periodic and sporadic) task systems. We conclude in Section 5 with an enumeration of some interesting open issues and questions.

2 Model and Background

A mixed-criticality (MC) implicit-deadline recurrent (i.e., periodic or sporadic) task system τ consists of a finite specified collection of MC implicit-deadline recurrent tasks, each of which may generate an unbounded number of MC jobs.

MC jobs. As stated in Section 1 above, we will, for the most part, restrict our attention here to *dual*-criticality systems: systems with two distinct criticality levels. A *dual-criticality job* J_i is characterized by a tuple of parameters: $J_i = (\chi_i, a_i, [c_i^L, c_i^H], d_i)$, where

- $\chi_i \in \{L, H\}$ denotes the criticality of the job;
- $a_i \in R^+$ is the release time;
- c_i^L and c_i^H denote low-criticality and high-criticality estimates of the job's worst-case execution time (WCET) parameter; and
- $d_i \in R^+$ is the deadline.

System behavior. The MC job model has the following semantics. Job J_i is released at time a_i, has a deadline at d_i, and needs to execute for some amount of time γ_i. The value of γ_i is not known beforehand, but only becomes revealed by actually executing the job until it *signals* that it has completed execution. These values of γ_i for a given run of the system defines the kind of *behavior* exhibited by the system during that run. If each J_i signals completion without exceeding c_i^L units of execution, we say that the system has exhibited LO-*criticality behavior*; if even one job J_i signals completion after executing for more than c_i^L but no more than c_i^H units of execution, we say that the system has exhibited HI-*criticality behavior*. If any job J_i does not signal completion despite having executed for c_i^H units, we say that the system has exhibited *erroneous behavior*.

Clairvoyance. Before scheduling a collection of jobs, a *clairvoyant scheduling algorithm* knows, for each job J_i in the collection, the precise duration γ_i for which the job will need to execute prior to signaling completion. (Note that clairvoyant scheduling algorithms represent a hypothetical ideal that are not in general implementable in actual systems.). By contrast, an *on-line scheduling algorithm* does not know the values of γ_i beforehand; the value of γ_i is only revealed by executing J_i for a duration γ_i, at which instant it signals completion.

The notions of clairvoyant and on-line scheduling algorithms extend in the obvious manner to the scheduling of recurrent tasks (discussed next).

MC implicit-deadline recurrent tasks. Analogously to traditional (non-MC) implicit-deadline recurrent tasks, an MC implicit-deadline recurrent (*periodic* or *sporadic*) task τ_k is characterized by a four-tuple $(\chi_k, C_k^L, C_k^H, T_k)$, with the following interpretation. Task τ_k generates an unbounded sequence of jobs, with successive jobs being released **exactly** T_k time units apart if the task is periodic, and **at least** T_k time units apart if it is sporadic. Each such job has a deadline that is T_k time units after its release. The criticality of each such job is χ_k, and it has LO-criticality and HI-criticality WCET's of C_k^L and C_k^H respectively; we assume that $C_k^L \leq C_k^H$ for all tasks τ_k.

An MC *implicit-deadline periodic/ sporadic task system* is specified by specifying a finite number of such periodic/ sporadic tasks. As with traditional (non-MC) systems, a MC sporadic task system can potentially generate infinitely many different MC instances (collections of jobs), each instance being obtained by taking the union of one sequence of jobs generated by each task.

Correctness criteria. We define an MC scheduling algorithm to be *correct* if it is able to schedule any system such that

- During all LO-criticality behaviors of the system, all jobs receive enough execution between their release time and deadline to be able to signal completion; and
- During all HI-criticality behaviors of the system, all HI-criticality jobs receive enough execution between their release time and deadline to be able to signal completion.

As defined in Section 1, a dual-criticality instance is said to be *clairvoyant schedulable* if it can be scheduled correctly by some clairvoyant scheduling algorithm, *MC-schedulable* if it can be scheduled correctly by some (non-clairvoyant) on-line algorithm; and A-schedulable for some specified scheduling algorithm A if it can be scheduled correctly by the algorithm A.

3 Scheduling Mixed-Criticality Jobs

In this section we look at a very simple mixed-criticality scheduling problem: that of scheduling instances that are collections of independent dual-criticality jobs upon a single preemptive processor. To our knowledge, this problem was first studied in [4], where an algorithm called OCBP was proposed for solving it; OCBP were further studied in [5, 10]. Recall that an instance is defined to be *clairvoyant-schedulable* if it is scheduled correctly by an optimal clairvoyant algorithm; *MC-schedulable* if it is scheduled correctly by some (non-clairvoyant) on-line algorithm; and *OCBP-schedulable* if it is scheduled correctly by OCBP. The following results were proved in [5, 10] (again, Φ denotes the golden ratio).

R1 There are clairvoyant-schedulable instances that are not MC-schedulable with speedup $< \Phi$.
R2 Determining MC-schedulability is NP-hard in the strong sense (even if all jobs in the instance have equal release dates).
R3 If an instance is clairvoyant schedulable, then it is OCBP-schedulable with speedup Φ.

Results **R1** and **R2** above reveal that the "difficulty" in the problem being solved by OCBP arises from two sources: **R1** tells us that no on-line algorithm, regardless of its run-time computational complexity, can solve it optimally, while from **R2** it appears unlikely that the polynomial-time OCBP is able to even solve the on-line problem exactly. Result **R1** above tells us that no (non-clairvoyant) on-line algorithm can have a competitive factor smaller than Φ. Analogously to the competitive factor metric for quantifying on-line algorithms, the effectiveness of polynomial-time algorithms for solving NP-hard problems approximately is commonly quantified using the *approximation ratio* metric. The result **R3** is therefore somewhat paradoxical, and reveals one of the shortcomings of speedup factor as a metric for mixed-criticality scheduling algorithms: the fact that OCBP is only approximately solving an NP-hard problem is not revealed in its speedup factor (which takes on the optimal value of Φ).

In this section we consider OCBP from the perspectives of comparing its performance to both a clairvoyant algorithm (here we look at its competitive factor) and an optimal non-clairvoyant algorithm (here, we examine its approximation ratio). Both metrics provide different perspectives on its "distance" from optimal behavior – its competitive factor is a measure of its distance from optimality due to its non-clairvoyance (its not knowing the future) while its approximation ratio is due to its computational limitations – it is solving an NP-hard problem in polynomial time.

3.1 Current State of the Art

The OCBP algorithm is known to be speedup-optimal for scheduling dual-criticality collections of independent jobs. Given such a dual-criticality instance I, OCBP aims to derive offline (i.e., prior to run-time) a total priority ordering of the jobs of I such that scheduling the jobs according to this priority ordering guarantees a correct schedule, where *scheduling according to priority* means that at each moment in time the highest-priority available job is executed.

The priority list is constructed recursively using the approach commonly referred to in the real-time scheduling literature as the "Audsley approach" [1, 2]. OCBP first identifies a lowest priority job: Job J_i may be assigned lowest priority if

- it is a low-criticality job (i.e., $\chi_i = L$) and there is at least c_i^L time between its release time and its deadline available if every other job J_j has higher priority and is executed for C_j^L time units; **or**

- it is a high-criticality job (i.e., $\chi_i = H$) and there is at least c_i^H time between its release time and its deadline available if every other job J_j has higher priority and is executed for C_j^H time units.

The above procedure is repeated on the set of jobs excluding the lowest priority job, until all jobs are ordered, or at some iteration no lowest priority job is identified (if this happens OCBP declares failure and exits: it is unable to schedule this instance).

The results **R1**–**R3** listed above permit us to draw the following conclusions about the effectiveness of OCBP:

1. From **R1** and **R3**, we conclude that *OCBP has the optimal* ***competitive ratio*** *from the perspective of speedup-versus-clairvoyance.*
2. But that leaves unanswered the question of how far OCBP is from **on-line optimality** (as indicated by MC-schedulability). In contrast to **R2**, OCBP-schedulability can be determined in polynomial time – what is the **approximation ratio** of OCBP in comparison to an optimal on-line scheduler?

3.2 Additional Insights

We now describe some of our new findings that allow us to better characterize the effectiveness with which the polynomial-time OCBP algorithm approximates solutions to the NP-hard problem of MC-schedulability. We first show, in Section 3.2.1, that OCBP is in fact optimal for scheduling MC-schedulable instances comprising just two jobs. (Although this result may at first seem trivial, we point out that the proof of result **R1** was obtained using 2-job instances; hence the result proving optimality of OCBP vis-à-vis MC-schedulability serves to separate the intractability arising from non-clairvoyance from the intractability arising from computational intractability issues). This positive result contrasts with a negative result in Section 3.2.2, where we show that in general, however, OCBP's approximation ratio is at least Φ: there exist MC-schedulable instances that are not OCBP-schedulable with speedup $< \Phi$.

3.2.1 Instances with at most two jobs

We now show that OCBP is able to optimally schedule any instance comprising at most two jobs. The cases when the instance comprises zero or one jobs is trivial; let us therefore focus on two-job instances. Consider first the case when both jobs have the same release date (without loss of generality, assumed equal to zero). If both jobs have the same criticality, or if the high-criticality job has an earlier deadline, OCBP will find the EDF-schedule for the jobs, and this is the optimal schedule. Consider, therefore, the case when the earlier-deadline job has lower criticality. Consider the following instance:

$$\{J_1 = (L, 0, [c_1^L, -], d_1), J_2 = (H, 0, [c_2^L, c_2^H], d_2)\}$$

with $d_1 < d_2$.

We claim that this instance is MC-schedulable if and only if (in addition to each job being "well-formed" – i.e., $c_i^L \leq c_i^H$ – and individually feasible) the following condition is satisfied:

$$\Big((c_1^L + c_2^L \leq d_1) \bigvee (c_1^L + c_2^H \leq d_2) \Big) \tag{2}$$

The reason why this is a necessary and sufficient condition for MC-schedulability is as follows:

Table 1 Example job instance depicting the non-optimality of OCBP vis-à-vis MC-schedulability. (Here ϵ denotes a positive constant < 1, and $y > 1$.)

	criticality	release date	wcets	deadline
J_1	high	0	$[\epsilon, 1]$	1
J_2	low	0	$[1-\epsilon, 1-\epsilon]$	1
J_3	high	0	$[y-1, y-1]$	y

- If the first disjunct is satisfied, we start out executing J_2.
  This is clearly a correct scheduling strategy, since in any LO-criticality behavior both jobs will complete before the earlier deadline d_1 while in any HI-criticality behavior the HI-criticality job gets to execute to completion and hence (since each job is assumed to be individually feasible) meets its deadline.
- If the second disjunct is satisfied, we start out executing J_1.
  This, too, is clearly correct: the LO-criticality job executes first and therefore (since it is assumed individually feasible) receives up to c_1^L units of execution by its deadline. It does not receive more than c_1^L units of execution regardless of whether it signals completion or not; hence, the disjunct ensures that the HI-criticality job always gets up to $c_@^H$ units of execution prior to its deadline.
- If neither disjunct is satisfied, then consider any schedule over the interval $[0, d_1]$:
  - If job J_1 has executed for $< c_1^L$ units, then the instance reveals low-criticality behavior
  - Else job J_1 has executed for c_1^L units (in which case job J_2 could not have executed for c_2^L units to reveal high-criticality behavior), and the instance reveals high-criticality behavior.

Now observe that if Condition 2 is satisfied, then OCBP would successfully schedule the system – if the first disjunct is true then J_1 can be assigned lower priority while if the second disjunct is true then J_2 could be assigned lower priority, by OCBP.

Now to generalize to the case where the two jobs may not have equal release dates, observe that any work-conserving algorithm would schedule the job that is released throughout the interval between its release and the release date of the second job (or to completion – whichever occurs first). And once the second job arrives, we are back to the case of two jobs with equal release dates, with the WCETs of the job that was released first appropriately modified to reflect the execution that has already occurred.

3.2.2 General Instances

We have seen that OCBP is optimal vis-à-vis MC-schedulability for dual-criticality instances with two or fewer jobs. Unfortunately, this optimality property does not hold for > 2 jobs; we show this by constructing a 3-job instance shown in Table 1. It is easy to show that *this instance is MC-schedulable*: an optimal on-line algorithm would execute J_1 to completion. If this occurs within ϵ time-units, then the optimal algorithm executes J_2 and then J_3; if not, it simply discards J_2 and executes J_3 to completion.

However, *this instance is not OCBP-schedulable*; we prove this by showing that none of the three jobs can be assigned lowest priority by OCBP (and hence OCBP would report failure at the very beginning – it is unable to assign any job lowest priority):

1. For J_1 to be lowest-priority, we would need

$$\Big(1 + (1-\epsilon) + (y-1) \le 1\Big) \Leftrightarrow \Big(1 + y - \epsilon \le 1\Big)$$

   which is impossible since $\epsilon < 1$ and $y > 1$.

2. For J_2 to be lowest-priority, we would need
$$\Big(\epsilon + (1-\epsilon) + (y-1) \le 1\Big) \Leftrightarrow \Big(y \le 1\Big)$$
which is impossible since $y > 1$.
3. For J_3 to be lowest-priority, we would need
$$\Big(1 + (1-\epsilon) + (y-1) \le y\Big) \Leftrightarrow \Big(1-\epsilon+y \le y\Big)$$
which is impossible since $\epsilon < 1$.

We saw above that the 3-job instance depicted in Table 1 is not OCBP-schedulable; we now compute the minimum speedup needed in order that this instance be OCBP-schedulable upon the faster platform. Since the instance is MC-schedulable this speedup value would represent a lower bound on the approximation ratio of OCBP (vis-à-vis MC-schedulability).

Note that OCBP must assign some job lowest priority. We consider each of the three jobs as potential candidate lowest-priority jobs:

1. For J_1 to get lowest priority, the processor would to complete $(1+y-\epsilon)$ units of work by time-instant 1, in order that J_1 complete by its deadline at time-instant 1. The needed speedup is therefore
$$(1+y-\epsilon) \tag{3}$$
2. For J_2 to get lowest priority, the processor would to complete y units of work by time-instant 1, in order that J_2 complete by its deadline, also at time-instant 1. The needed speedup is therefore
$$y \tag{4}$$
3. For J_3 to get lowest priority, the processor would to complete $(1+y-\epsilon)$ units of work by time-instant y, in order that J_3 complete by its deadline at time-instant y. The needed speedup is therefore
$$\frac{1+y-\epsilon}{y} = 1 + \frac{1-\epsilon}{y} \tag{5}$$

Since $y > \epsilon$, Expression (4) < Expression (3) and hence J_2 requires a lower speedup than J_1 to be a viable lowest-priority job. Which of J_2 or J_3 needs a lower speedup to be a viable lowest-priority job depends upon the exact values of ϵ and y. Since the speedup needed for J_2 to be a viable lowest-priority job increases, while the speedup needed for J_3 to be a viable lowest-priority job decreases, with increasing y, the largest speedup needed occurs when the two values are equal:
$$\begin{aligned} & y = 1 + \frac{1-\epsilon}{y} \\ \Leftrightarrow \quad & y^2 - y - (1-\epsilon) = 0 \end{aligned}$$
As $\epsilon \to 0$, the solution to the quadratic equation above approaches the golden ratio $\Phi = (\sqrt{5}+1)/2$, and the speedup needed is equal to this value of y.

We have therefore just shown that this is at least one MC-schedulable instance – the one depicted in Table 1 – for which OCBP needs a speedup of Φ. It has previously been shown that any clairvoyant-schedulable instance is OCBP schedulable with speedup Φ (this is the result **R3** referenced above), from which it follows that any MC-schedulable instance is also OCBP schedulable with speedup at most Φ. We therefore conclude that **the approximation ratio of OCBP is equal to** Φ, yielding the situation depicted in Section 1.

4 Sporadic Task Schedulability

We focus our attention in this section on dual-criticality implicit-deadline recurrent – sporadic and periodic – task systems. Our main contribution here is a proof that the mixed-criticality scheduling problem is NP-hard for such systems; this stands in sharp contrast to the analogous problem for single-criticality systems, where a simple linear-time utilization-based schedulability test is known [11]. This computational complexity result serves to establish that determining MC-schedulability is computationally intractable for dual-criticality recurrent task systems, and hence demonstrates that EDF-VD (like OCBP) is dealing with two sources of intractability: non-clairvoyance and computational complexity. This opens up the need for separating out the effects of the two sources of intractability, and also highlights the need for metrics that are able to separately quantify the approximation-ratio effect (i.e., comparison with MC-optimality) and the competitive-ratio effect (i.e., the sub-optimality arising from the lack of clairvoyance) of EDF-VD in scheduling recurrent sysems..

4.1 Task Set Construction

Recall that, dual criticality mixed criticality job scheduling is NP-complete in the strong sense [5, Theorem 1]. We will use that proof as a starting point for our proof to show how a similar reduction can also be used for both periodic and sporadic tasks. The reduction in [5, Theorem 1] is from an instance I of 3-PARTITION. The 3-PARTITION problem is the following: we are given a set S of $3m$ positive integers $s_0, s_1, s_2, \ldots, s_{3m-1}$ and a positive integer B such that $B/4 < s_i < B$ for each i, and $\sum_{i=0}^{3m-1} s_i = mB$. Instance I is said to have a feasible solution if S can be partitioned into m disjoint sets, $S_0, S_1, \ldots, S_{m-1}$ each of which sum to B.

Hardness construction for mixed-criticality jobs from [5, Theorem 1]. In [5, Theorem 1], the authors showed that dual-criticality job schedulability is solvable in polynomial time iff 3-PARTITION is solvable in polynomial time. Given an instance I of 3-PARTITION, they defined a polynomial-time procedure for generating a set X_I of dual-criticality jobs such that X_I can be scheduled correctly by an online scheduler iff I has a feasible solution.[5]

From I, the dual-criticality jobs set X_I is generated as follows:

- $3m$ HI-criticality jobs ($\chi_i = H$) each with release time 0, deadline $2mB$, $C_i^L = s_i$ and $C_i^H = 2s_i$.
- m LO-criticality jobs ($\chi_i = L$) each with release time iB for $0 \le i \le (m-1)$ and deadline $2B$ after its release and $C_i^L = C_i^H = B$.

The derivation in [5, Theorem 1] shows that this job system X_I is schedulable in an MC-correct manner iff I has a valid 3-partition. We will not recall the details of the proof that shows that if X_i is schedulable, then I is feasible since those details are not important for our proof here. We will briefly recall the direction that shows that if I is feasible then X_I is schedulable. To do so, given a feasible 3-PARTITION for I, we must provide a schedule for X_I such that all jobs are scheduled correctly.

Given a feasible partition $S_0, S_1, S_2, \ldots, S_{m-1}$, the feasible scheduling policy is as follows. Look at m chunks of time, each of size (i.e., duration) $2B$. In the first B time of chunk j, we execute the tasks in S_j – since the LO-criticality execution time of all jobs in S_j sums to B,

[5] To prove NP-completeness, one must also reduce from a set of jobs to an instance of 3-PARTITION; however, we are only concerned with NP-hardness in this paper.

this interval is sufficient to finish these jobs if the system stays in LO-criticality mode. In the second chuck of time, we execute the LO-criticality job. Therefore, all jobs are schedulable in LO-criticality behavior.

We now argue about HI-criticality behavior. Consider a job τ_i in set S_j. Say this job exceeds its LO-criticality execution time. Then the system will discover this by time $(2j+1)B$ since all jobs of set S_j will have completed their LO-criticality execution by then. At this point, the system will transition to HI-criticality mode and all LO-criticality jobs will be discarded. Therefore, the schedule has $(2m-2j-1)B$ time to finish the HI-criticality jobs' remaining computation before their deadline. At this point, all jobs in sets $S_0, S_1, ..., S_{j-1}$ have already signalled completion and the jobs in set S_j have completed B units of work. Therefore, even if all jobs in sets $S_{j+1}, ..., S_{m-1}$ execute for their HI-criticality execution time of $2s_i$, we have total of $(2m-2j-1)B$ total remaining HI-criticality work which can be completed by the deadline.

Translating the construction to dual-criticality tasks. The construction in [5, Theorem 1] described above applies to dual-criticality jobs; now we will show how we can use a similar construction for dual-criticality tasks. Given a feasible instance of 3-PARTITION I, we construct a mixed-criticality job system Y_I as follows:

- $3m$ HI-criticality tasks ($\chi_i = H$) each with period $2mB$, $C_i^L = s_i$ and $C_i^H = 2s_i$.
- 1 LO-criticality task ($\chi_i = L$) with period $2B$ and $C_i^L = C_i^H = B$.

Note that if the system is periodic with the first release of all jobs at time 0, then Y_I generates an instance of dual-criticality jobs X_I in each hyper-period of $2mB$. Therefore, the following lemma about periodic jobs is obvious.

▶ **Lemma 1.** *If tasks are periodic, then this task set Y_I is schedulable iff the job system X_I was schedulable. Therefore, determining schedulability of periodic implicit deadline dual-criticality task systems is NP-hard in the strong sense.*[6]

It is not so straightforward, however, to argue that sporadic schedulability is NP-hard in the strong sense. We will now prove the following theorem in the rest of this section.

▶ **Theorem 2.** *A sporadic task system generated by Y_I is schedulable iff I has a feasible solution, that is, if the corresponding periodic task system is schedulable. Therefore, determining schedulability of sporadic implicit deadline dual-criticality task systems is NP-hard in the strong sense.*

It is clear that if the sporadic task system is schedulable, then the periodic one is also schedulable since periodic releases are just an instance of sporadic releases. The rest of this section will show that if the periodic task system is schedulable, then the sporadic one is also schedulable.

Note that we are not making a general claim about all task systems. It is sufficient to show the following: If an instance I of 3-PARTITION (set S of $3m$ positive integers $s_0, s_1, s_2, ..., s_{3m-1}$ and a positive integer B such that $B/4 < s_i < B$) is feasible (we can find m sets $S_0, S_1, ..., S_{m-1}$ where each set sums to B), then the corresponding dual-criticality sporadic task system Y_I is schedulable. We will do so by designing a particular scheduler for this type of task set.

[6] Again, we are concerned only with NP-hardness; therefore, we needn't show a reduction from periodic task instances to 3-PARTITION instances.

4.2 Scheduling a Sporadic Instance of Y_I

Here, we wish to design a schedule that can correctly schedule the sporadic task set if the periodic task set is schedulable. In other words, given a feasible instance of I, this scheduler will always meet all deadlines if no job exceeds its LO-criticality WCET and will meet all HI-criticality deadlines if some job exceeds its LO-criticality WCET, but does not exceed its HI-criticality WCET. Intuitively, this scheduler tries to mimic the periodic schedule.

LO-criticality mode scheduling algorithm. The run-time scheduling algorithm is an almost-fixed-priority scheduler when in LO-criticality mode. In particular, each high-criticality task is assigned a priority and higher-priority high-criticality tasks take precedence over lower-priority high-criticality tasks. Priority assignment to the $3m$ HI-criticality tasks is determined according to the set they are in for the solution of I. Tasks in set S_0 have priority 0, tasks in S_1 have priority 1 and so on – here, lower numbers denote greater scheduling priority.

However, scheduling decisions for low-criticality jobs are a little different. Each HI-criticality task τ_i in set S_j maintains an auxiliary variable called *slack* $\ell_i \leftarrow jB$ – this is the amount of time the LO-criticality task is allowed to execute after τ_i is released before τ_i becomes "higher priority" than the LO-criticality task.

The schedule is a work-conserving schedule with the following properties. A HI-criticality job always yields to all higher priority HI-criticality jobs. That is, a job generated by task τ_i in set S_j will yield to all jobs generated by tasks in sets $S_0, S_1, .., S_{j-1}$. There are three tasks at each priority level and the scheduler can arbitrarily pick between jobs of these tasks. In addition, each job of HI-criticality task τ_i keeps track of how much LO-criticality work has executed since its release. While this work is smaller than ℓ_i, it yields to LO-criticality jobs. After this work is equal to ℓ_i, it never yields to LO-criticality jobs.

HI-criticality mode scheduling algorithm. The system transitions to HI-criticality mode at time T_r if any HI-criticality job executes for C_i^L time without signalling completion. After T_r, the jobs generated by LO-criticality task are never given any execution time and the HI-criticality jobs just run with earliest deadline first scheduling.

4.3 Proof of Schedulability

We must now show that this task system is always schedulable using the scheduling algorithm described above. The basic idea is that the scheduler tries to mimic the periodic schedule.

Recall that, in the periodic schedule in LO-criticality mode, jobs generated by tasks in S_0 never experience any interference since they are executed as soon as they are released. In general, jobs of tasks of sets $S_0, S_1, ..., S_{j-1}$ execute before jobs of tasks in S_j; in addition jB jobs of the LO-criticality task can execute before tasks in S_j. Therefore, tasks in S_j are guaranteed to complete by time $(2j + 1)B$ in the periodic schedule. The scheduler described above is specifically designed to ensure that a job of τ_i in set S_j never experiences more interference in the sporadic schedule than it experiences in the periodic schedule; therefore, the response time of each HI-criticality task in LO-criticality behavior is at most its response time in the strictly periodic scheduler. This leads to two good properties. First, and more obviously, all HI-criticality tasks meet their deadlines in LO-criticality mode. Second, and less obviously, since the response time of all HI-criticality jobs in LO-criticality mode is bounded by their response time in the periodic schedule, the transition point occurs "early enough." That is, if a HI-criticality job exceeds its LO-criticality WCET and the system transitions into HI-criticality behavior, then all pending jobs still have enough time to complete their

HI-criticality work C_i^L. In addition, we must argue that while the system remains in LO-criticality mode, all LO-criticality tasks also meet their deadlines. This is possibly the most counter-intuitive part of the argument; here we argue that the slack condition ensure that at most a total of B HI-criticality work reaches slack 0 and preempts any single LO-criticality job during its execution.

We will first argue that all jobs (from both HI-criticality and LO-criticality tasks) meet their deadlines while the system remains in LO-criticality behavior and then we will argue that HI-criticality jobs meet their deadlines if the system transitions to HI-criticality behavior.

4.3.1 Correctness Proof for LO-Criticality Behavior

We will first argue that all HI-criticality tasks meet their deadlines in LO-criticality behavior. This is relatively straightforward. We first prove a simple lemma about *idle instant* – that is, an instant such that all pending work has completed. In particular, at the idle instant, all the jobs that were released before this instant has completed.

▶ **Lemma 3.** *While the system remains in* LO*-criticality behavior, in any interval of size* $2mB$*, there is at least one idle instant.*

Proof. We start at an idle instant t and argue that the next idle instant is within $2mB$ time. Look at the interval from t to $t + 2mB$. During this interval, at most mB LO-criticality work is released and at most mB HI-criticality work is released. If this interval stays busy, then all this work is done by the end of this interval; therefore, there is an idle instant at time $t + 2mB$ if there is not one before. ◀

The following corollary follows from Lemma 3 and the fact that the inter-arrival time between consecutive jobs of the same HI-criticality task in our task system is at least $2mB$.

▶ **Corollary 4.** *While the system remains in* LO*-criticality mode, between any two consecutive job releases of the same* HI*-criticality task, there is an idle instant.*

We can now use this corollary to show that all HI-criticality tasks meet their deadlines in LO-criticality mode.

▶ **Lemma 5.** *The response time of a job of task* τ_i *in set* S_j *is bounded by* $(2j + 1)B$ *in* LO*-criticality mode. Therefore, all* HI*-criticality tasks meet their deadlines in* LO*-criticality mode.*

Proof. Due to Corollary 4, no job can suffer interference from two jobs of the same task since there is an idle instant between consecutive releases. A HI-criticality task τ_i in set S_j has priority j and only tasks in sets $S_0, S_1, ..., S_j$ can interfere with it. Therefore, the total LO-criticality WCET of higher priority tasks including its own priority, is exactly $(j + 1)B$. In addition, the total interference it can experience from LO-criticality tasks is at most jB (by enforcement of the slack scheduling policy). Therefore, the job has a response time of at most $(2j + 1)B$. ◀

We now prove the more interesting result that all jobs generated by the LO-criticality task also meet their deadlines in the LO-criticality mode. This argument depends on the slacks. In general, one might worry that if there are many HI-criticality jobs with small remaining slack, then a LO-criticality job may starve and not be able to get enough computation time by its deadline. We will now argue that this cannot happen.

Intuitively, this is easiest to see in a strictly periodic schedule starting at time 0. The jobs of tasks in S_0 have no slack since for all these tasks $\ell_i = 0$ – therefore, they execute first taking up the first B time steps. After this, however, the first LO-criticality job gets to execute and meets its deadline. At this point, jobs in S_1 have exhausted their slack and they execute next. The jobs in S_2 started with slack of $2B$ and have B slack remaining, so the second LO-criticality job executes before them. Therefore, the LO-criticality job gets B execution in the interval from its release time to its deadline every time and meets all its deadlines. We will do an induction argument to see this. In order to do this induction, we need two additional definitions.

We define $w(\tau_i, t)$ as the remaining work of task τ_i at time t. At an idle instant, $w(\tau_i, t) = C_i^L$. If no job of a task τ_i has been released since the last idle instant or a job has been released, but has not yet done any execution, then $w(\tau_i, t) = C_i^L$. As a job of a task executes, its w decreases. In particular, if a job of the task τ_i has done work w since the last idle instant, then $w(\tau_i, t) = C_i^L - w$. If the job (released since last idle period) has completed, then $w(\tau, t) = 0$.

We will define a quantity *running slack* $r\ell_i$ for each task τ_i. At the beginning of the execution and at any idle instant, the running slack $r\ell_i$ is set to the task slack ℓ_i – that is, tasks in set S_j get a slack of jB. This running slack remains unchanged until a job of this task τ_i is released. Once a job J_i of a task τ_i is released, $r\ell_i$ decreases every time step that a job of the LO-criticality task executes – this corresponds to keeping track of how much LO-criticality work has executed since job J_i arrived. By the scheduler definition, once $r\ell_i = 0$ the job J_i stops yielding to LO-criticality jobs and no LO-criticality job can execute until J_i finishes executing. Note that it is sufficient to reset $r\ell_i$ values at each idle instant since by Corollary 4, there is an idle instant between consecutive arrivals of any HI-criticality task; therefore, the running slack $r\ell_i$ is always reset to ℓ_i between consecutive job arrivals of the same task. (Note that since we have a work-conserving scheduler, HI-criticality jobs with slack larger than 0 can run if there is no LO-criticality pending job.)

Recall that we must argue that if a LO-criticality job arrives at time t, at most B HI-criticality work can preempt it – this would ensure that the LO-criticality job gets enough execution time within its release time to deadline scheduling window of $[t, t+2B]$ to complete its own execution requirement of B. We define a quantity that generalizes this notion: we define $Ł(K, t)$ as the total work with slack at most K after time step t. In other words, $Ł(K, t) = \sum_{r\ell_i \leq K} w(\tau_i, t)$. If $Ł(K, t) = W$, then W HI-criticality work has slack smaller than K at time t. At any time instant t, some HI-criticality job can preempt a LO-criticality job only if its work is in $Ł(0, t)$.

At a high level, the function $Ł(K, t)$ can be looked upon as a measure of how much work is urgent (and to what extent). Consider the platform at time r^J when a LO-criticality job J arrives. All the work in $Ł(0, r^J)$ has no slack (its $r\ell_i$ value is 0) and will execute before this LO-criticality job J can start execution. After this the LO-criticality job J can start, but all tasks that have pending jobs in the system will reduce their $r\ell_i$ values – therefore, when J has completed 1 unit of work at time $r^J + t_1$, the pending work that was in $Ł(1, r^J)$ (had running slack $r\ell_i$ smaller than 1 at time t) will now be in $Ł(0, r^J + t_1)$ and will preempt this LO-criticality job. In general, if J has completed b work by time $r^J + t_b$, then the work that was in $Ł(b, r^J)$ may now be in $Ł(0, J_r + t_b)$ and can preempt this LO-criticality job at time $r^J + t_b$. However, note that any work that was not in $Ł(B-1, r^J)$ cannot become urgent before J completes since its slack can only reduce by B and therefore, cannot become 0 before J completes.

We can divide this Ł function into two components: $Ł_f$ and $Ł_p$. $Ł_f$ contains the work of all the tasks where no job of this task has yet arrived into the system since the last idle instant. Therefore,

$$Ł_f(K,t) = \sum_{r\ell_i \leq K, \tau_i \text{not pending}} w(\tau_i, t) \quad = \sum_{r\ell_i \leq K, \tau_i \text{not pending}} C_i^L.$$

$Ł_p$ contains the work of all the jobs that are pending. For all tasks whose jobs have finished executing since the previous idle instant, their corresponding w is 0. Therefore, for all parameters, $Ł_f$ and $Ł_p$ sum up to $Ł$.

We first prove an invariant on $Ł_f$.

▶ **Lemma 6.** *At any time t, $Ł_f(jB-1,t) \leq jB$ for all $1 \leq j \leq m$.*

Proof. At an idle instant, at time 0, before any jobs arrive, all running slacks reset. Therefore, by definitions of ℓ_i, the tasks in set S_j have $r\ell_i = jB$. Therefore, there is B total work with slack 0, B work with slack B, and so on. Therefore, the total work with slack $B-1$ is $Ł_f(B-1,0) = B$, the total work with slack $2B-1$ is $Ł_f(2B-1,0) = 2B$ and, in general, the total work with slack at most $jB-1$ is $Ł_f(jB-1,0) = jB$. Since the total pending HI-criticality work at any time is at most mB and nothing has slack more than $(m-1)B$, we have $Ł_f((m-1)B,t) = Ł_f(mB-1) \leq mB$. $Ł_f$ is maximum at an idle instant, since $Ł_p$ is 0 for all values. After this point, $Ł_f$ only reduces as jobs arrive in the system. ◀

We will now prove an invariant on the function Ł which will let us prove that the LO-criticality task is schedulable.

▶ **Lemma 7.** *When a LO-criticality job J arrives at time r^J, we have $Ł(jB-1,r^J) \leq jB$ for all $1 \leq j \leq m$.*

Proof. We will prove a stronger statement by induction.[7] Without loss of generality, we reset time to 0 at an idle instant. Say an LO-criticality job arrives at time r^J. After the arrival of this LO-criticality job and until its completion or until an idle instant (whichever is sooner), we have to prove the following: Say at time r^J+b+c, the LO-criticality job has executed for b time steps and some HI-criticality jobs have executed for c time steps. Then, we have $Ł(jB-1-b, r^J+b+c) \leq jB-c$. If we prove this statement, then we obviously get the lemma by substituting $b=c=0$.

At an idle instant, at time 0, all running slacks reset and all L is in $Ł_f$. Therefore, Lemma 6 gives us the base case.

We now induct on time steps. Say a LO-criticality job J was released at time step r^J and has a deadline of d^J. Lets say that at in interval $[r^J, r^J+b+c]$, the schedule has done c HI-criticality work and b work on this LO-criticality job. By the inductive hypothesis, we have $Ł(jB-1-b, r^J+b+c) \leq jB-c$.

Case 1: On the next step, say we do 1 unit of LO-criticality work. Then at time step $r^J+b+c+1$, we have done $b+1$ LO-criticality work and c HI-criticality work. We must show that $Ł(jB-1-b-1, r^J+b+c+1) \leq jB-c$. Now the slack of all jobs that have arrived by time r^J+b+c, but not completed reduces by 1. Therefore, we have $Ł_p(iB-1-b-1, r^J+b+c+1) = Ł_p(iB-1-b, r^J+b+c)$. In addition, $Ł_f(iB-1-b-1, r^J+b+c) = Ł_f(iB-1-b, r^J+b+c)$ since $b \leq B$ and $Ł_f$ only changes at discrete intervals of size B. Therefore, we have $Ł(iB-1-b-1, r^J+b+c+1) = Ł_p(iB-1-b-1, r^J+b+c+1) + Ł_f(iB-1-b-1, r^J+b+c+1) =$

[7] Without loss of generality, we are assuming discrete time for ease in induction. One can imagine that each time step takes as long as a machine instruction. One can also do this induction based on events; but it gets more complicated.

$Ł_p(iB-1-b, r^J+b+c) + Ł_f(iB-1-b, r^J+b+c) \leq jB-c$. If a job is released at time step $r^J+b+c+1$, this moves its work from $Ł_f$ to $Ł_p$, but does not change the overall sum in Ł.

Case 2: On the next step, we do one unit of work on some HI-criticality job J_i from task τ_i in set S_j. The only way we can do this while job J is still pending is because $r\ell_i$ has become 0. Therefore, $w(\tau_i)$ is part of $Ł(0, r^J+b+c)$. Since Ł function is cumulative, this work is also part of all $Ł(K, r^J+b+c)$ for all $K>0$. Therefore, when this work is executed, all Ł's reduce by 1 on this time step. Therefore, $Ł(jB-1-b, r^J+b+c+1) = Ł(jb-1-b, r^J+b+c)-1 \leq jB-c-1$.

The deadline of the LO-criticality job is $2B$ time steps after its release time. Say we did $W \leq B$ LO-criticality work by the deadline and $2B-W$ HI-criticality work. Therefore, we have $Ł(iB-1-W, J_d) \leq iB-2B+W$. Since $W \leq B$, we have $Ł(iB-1-B, J_d) \leq iB-2B+W \leq iB-B$. Since this invariant is true for all $0 \leq i \leq m-1$, we have, $Ł(iB-1, J_d) \leq iB$. If the next job is immediately released, we have satisfied the invariant. If the next job is not immediately released, then some HI-criticality job executes and the $Ł(iB-1, t)$ for current time instant t only decreases, therefore, the invariant remains true when the next LO-criticality job is released. ◀

We can now argue that all LO-criticality jobs meet their deadlines if the system remains in LO-criticality mode. Recall that we argued that any work that was not in $Ł(B-1, t)$ when a LO-criticality job J arrives at time t cannot become urgent before J completes since its slack cannot become 0 before J completes.

▶ **Lemma 8.** *LO-criticality jobs finish by their deadlines in LO-criticality mode.*

Proof. From Lemma 7, we know that when a LO-criticality job arrives at time t, $Ł(B-1, t) \leq B$. Therefore, at most B work has slack smaller than B when this job arrives. As this job executes, slack of all jobs reduces, but only the work that already has slack smaller than B can get to 0 before this job finishes. Therefore, only this B work can interfere with this LO-criticality job. Therefore, the LO-criticality job gets to execute for B time units within its scheduling window of $2B$ and hence does not miss its deadline. ◀

4.3.2 Correctness for HI-Criticality Behavior

We now have to show that all jobs can meet their deadlines in HI-criticality mode. In particular, we will say that a transition occurs at T_r when some HI-criticality job executes for C_i^L time steps, but does not signal completion. The LO-criticality task does not get any further execution after time T_r. For all jobs that are pending at time T_r or arrive after T_r, we must argue that they can complete by their deadlines even if they execute for HI-criticality WCET C_i^H.

The intuition behind this proof relies on the periodic schedule. Recall that in the periodic schedule's schedulability relies on the following fact: if a job J from task τ_i in set S_j is released at time r^J (with deadline $d^J = r^J+2m$), then it is guaranteed to finish its LO-criticality work by time $r^J+(2j+1)B$. Therefore, if this job were going to cause a mode-transition, then it would happen before this time, implying $T_r = r^J+(2j+1)B$ (or earlier). In addition, by this time there are no pending jobs from any set in $S_1, S_2, ..., S_{j-1}$. Therefore, we can only have pending jobs from tasks in sets $S_j, S_{j+1}, ..., S_{m-1}$. The total HI-criticality WCET for all jobs in this set is $2(m-j)B$ and we have already done B work from set S_j. Therefore, the total pending work is $(2m-2j-1)B$. Since all HI-criticality jobs have the same release time and deadline in the periodic schedule, all this work can be completed by the deadline. Staring from the next hyper-period, we have an implicit deadline task system with on HI-criticality tasks which have total utilization of 1; therefore, they are schedulable.

In the sporadic schedule, we no longer have the nice property that all pending HI-criticality jobs have the same deadline. However, we can still argue the crucial point that the total pending work at the transition time T_r is bounded. In particular, just like the periodic schedule, if a job from task τ_i from set S_j causes the transition, then no job from sets $S_0, S_1, ..., S_{j-1}$ can be pending. Therefore, the total amount of pending work is bounded.

We will say that there is carry-over work for task τ_i if a job J of task τ_i has been released by the transition time T_r, but has not completed. We denote the set of carryover jobs by C. We say that each task τ_i has completed d_i work by transition time T_r if the latest release of this job has done d_i work by time T_r. This quantity d_i is defined for both carryover jobs and non-carryover jobs.

We now prove a generalization of Lemma 5.

▶ **Lemma 9.** *Consider a job J_i of task τ_i in set S_j with release time $r^{J_i} < T_r$ which has not completed by time T_r. We have $T_r \leq r^{J_i} + 2jB + \sum_{\tau_k \in S_j} d_k$. Therefore, the deadline for job J_i is $d^{J_i} \geq T_r + 2mB - 2jB - \sum_{\tau_k \in S_j} d_k$.*

Proof. After r^{J_i}, only tasks of higher priority than τ_i (those belonging to S_p where $p < j$) tasks in S_j and LO-criticality tasks can execute before. Since the system remains in LO-criticality mode until time T_r, no job has previously exceeded its LO-criticality WCET. There is only jB work with higher priority and due to the slack condition, only jB LO-criticality work can execute. Finally, $\sum_{\tau_k \in S_j} d_k$ work was done for tasks in set S_j. Since the system stays busy between r^{J_i} and T_r, we get the condition on release time. The condition on d^{J_i} is obtained by adding $2mB$ (the relative deadlines of all HI-criticality tasks) to r^{J_i}. ◀

Lemma 9 provides us a crucial condition that no pending job's deadline is too close to the transition point. Even more crucially, the higher the priority of the pending job (the smaller the j value), the farther in the future its deadline is guaranteed to be. This is due to the following. Either this job just completed its LO-criticality WCET C_i^L at time T_r and did not signal completion (causing the mode transition) or this job has not yet completed its LO-criticality WCET C_i^L. (If it had completed its C_i^L earlier, it cannot still be pending since it would either complete or cause a transition earlier.) But we know that higher priority jobs have a smaller response time in LO-criticality mode. Therefore, if higher-priority jobs were going to exceed their LO-criticality WCET, causing a transition, then they cause this transition sooner after their release compared to lower priority jobs.

This condition helps us prove completion of carryover jobs using a *reverse-priority* scheduler. This scheduler only looks at carry-over jobs (it ignores all jobs that arrive after T_r) and schedules jobs in reverse order of priorities. That is, carryover jobs from tasks in set S_p where $p > j$ are scheduled before carryover jobs from set S_j for all j.

▶ **Lemma 10.** *All jobs with carry-over work complete by their deadlines using the reverse-priority scheduler.*

Proof. As we mentioned earlier, Lemma 9 implies that if we look at all the pending jobs at transition time T_r, the higher the priority of the job, the farther in the future the deadline of the job is guaranteed to be. This does not mean that all deadlines are ordered by priorities – it only means that the higher priority jobs cannot have deadlines that are too soon after T_r.

It is easy to see that with the reverse priority scheduler, that carry-over jobs with priority j will finish by time $T_r + \sum_{\tau_k \in S_j, S_{j+1}, ..., S_{m-1}} (C_k^H - d_k) \leq T_r + 2(m-j)B - \sum_{\tau_k \in S_j, S_{j+1}, ..., S_{m-1}} d_k$ time using the reverse priority scheduler. Since the deadline of any job J_i with priority j is $d^{J_i} \geq T_r + 2(m-j)B - \sum_{\tau_k \in S_j} d_k$ (Lemma 9), the job meets its deadline. ◀

▶ **Corollary 11.** *All jobs with carry-over work complete by their deadlines using EDF.*

Proof. Since all carry-over jobs were released before T_r, EDF will always schedule them before scheduling any jobs released after T_r. Therefore, just like the reverse-priority scheduler, EDF ignores all non-carryover jobs until the carryover work is done. We know that EDF is an optimal scheduler and if any scheduler can schedule a set of jobs, EDF can. Therefore, since reverse-carryover scheduler and EDF consider the same set of carryover jobs, and reverse-carry over scheduler meets all deadlines (Lemma 10), then EDF also meets all deadlines. ◀

We must now prove that all jobs that are released after transition time T_r can meet their deadlines. These jobs are easier to reason about since after T_r, we do not have to worry about mixed-criticality execution any more – we simply have a simple implicit deadline task system where all tasks have the same period and deadlines.

▶ **Lemma 12.** *All* HI*-criticality jobs meet their deadlines in* HI*-criticality mode when scheduled with EDF.*

Proof. Corollary 11 indicates that all carryover jobs meet their deadlines. So we need only worry about jobs that are released after time T_r. Since all jobs have the same relative deadline and we schedule using EDF, no job can interfere with a job that is released after itself. Therefore, no job can suffer interference from 2 jobs of the same task. Since the total HI-criticality WCET of all HI-criticality tasks collectively is $2mB$ and the relative deadline is $2mB$, there is enough time to execute one job of all tasks after a job's release time and still meet its deadline. ◀

Lemmas 5, 8 and 12, and Theorem 2 collectively allow us to conclude that given an instance I of 3-PARTITION, we can generate a sporadic task system that is schedulable if and only if I has a solution. Therefore, a dual-criticality schedulability for sporadic task systems is NP-hard in the strong sense – this completes the proof of the main result in this section.

5 Conclusions and Discussion

Designing efficient mixed-criticality scheduling algorithms is a very challenging problem. In this paper we have described our efforts at approach this problem by breaking it out into two constituent components, seeking to separate the difficulties that arise from the on-line nature of mixed-criticality scheduling – the fact that much important information is simply not known prior to run-time – and those arising from computational complexity issues. As a major step to doing so, we first needed to establish that determining MC-schedulability for recurrent (periodic or sporadic) implicit-deadline task systems is NP-hard in the strong sense; in this manner, we showed that all three mixed-criticality scheduling problems considered – scheduling collections of jobs, of periodic tasks, and sporadic tasks – have to deal with both sources of intractability.

With regards to the scheduling of collections of independent jobs, we established, for the first time, an approximation ratio for OCBP vis-à-vis MC-schedulability, thereby quantifying OCBP's deviation from optimal behavior due to computational complexity issues (as opposed to its sub-optimality due to non-clairvoyance). This result gave rise to several interesting issues and questions:

1. A somewhat odd aspect of this result is that OCBP's approximation ratio and its competitive ratio (ie., its performance hit vis-à-vis a clairvoyant scheduler) are both equal to the same constant ($\Phi, \approx 1.618$); we do not have an understanding as to why this should be so.

2. We observed that speedup factor, when used as a metric for both competitive ratio and approximation ratio, does not appear to compose in any meaningful manner: while the competitive ratio of any MC-scheduling job algorithm is $\geq \Phi$ and OCBP's approximation ratio for solving the MC-scheduling problem is Φ, the speedup factor of OCBP vis-à-vis clairvoyant schedulability is also Φ (rather than, say, 2Φ or Φ^2).
3. It would be interesting to seek to characterize other algorithms that have been proposed for scheduling dual-criticality job instances (such as MC-EDF [13] and LE-EDF [9, page 29]) by approximation ratios. These algorithms have been experimentally observed to perform better than OCBP on randomly-generated data; perhaps their superiority can be quantified by showing that they have a smaller approximation ratio than Φ?

With regards to EDF-VD and the scheduling of recurrent task systems, the non-optimality of EDF-VD vis-à-vis MC-schedulability had not, to our knowledge, been explored previously. By showing that MC-schedulability for dual-criticality recurrent task systems is NP-hard in the strong sense, we have provided some justification for the use of these non-optimal algorithms. There are several open issues concerning the analysis of EDF-VD – we would, in essence, like to eventually have as complete an understanding of EDF-VD's effectiveness as we have currently been able to obtain for OCBP. This includes determining whether EDF-VD is optimal (vis-à-vis MC-schedulability – it is known to not be optimal vis-à-vis clairvoyant schedulability) for certain classes of task systems, determining approximation ratios, etc.

References

1 N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, The University of York, England, 1991.

2 N. C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, 1993.

3 S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

4 Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

5 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

6 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, 2017. doi:10.1145/3131347.

7 Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I. Davis. On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:25, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2017.9.

8 R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

9 Zhishan Guo. *Real-Time Scheduling Of Mixed-Critical Workloads Upon Platforms With Uncertainties*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 2016.

10 Haohan Li. *Scheduling Mixed-Criticality Real-Time Systems.* PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 2013.

11 C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

12 D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

13 Dario Socci, Petro Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, Paris (France), 2013. IEEE Computer Society Press.

14 J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

15 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

Improving the Schedulability and Quality of Service for Federated Scheduling of Parallel Mixed-Criticality Tasks on Multiprocessors

Risat Mahmud Pathan
Chalmers University of Technology, Sweden
risat@chalmers.se

Abstract

This paper presents federated scheduling algorithm, called MCFQ, for a set of parallel mixed-criticality tasks on multiprocessors. The main feature of MCFQ algorithm is that *different* alternatives to assign each high-utilization, high-critical task to the processors are computed. Given the different alternatives, we carefully select one alternative for each such task so that *all* the other tasks can be successfully assigned on the remaining processors. Such flexibility in choosing the right alternative has two benefits. First, it has higher likelihood to satisfy the total resource requirement of all the tasks while ensuring schedulability. Second, computational slack becomes available by intelligently selecting the alternative such that the total resource requirement of all the tasks is minimized. Such slack then can be used to improve the QoS of the system (i.e., never discard some low-critical tasks). Our experimental results using randomly-generated parallel mixed-critical tasksets show that MCFQ can schedule much higher number of tasksets and can improve the QoS of the system significantly in comparison to the state of the art.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases mixed-criticality systems, real-time systems, multiprocessor scheduling, federated scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.12

Funding This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

1 Introduction

Multicore processors offer high computing power to meet the increasing demand of more advanced functions in many real-time systems like automotive and avionics. Multicores also provide the opportunity to integrate multiple functions having different levels of criticality on the same platform. The real-time tasks of such mixed-critical (MC) systems require different levels of assurance in meeting their deadlines. A relatively high-critical task requires a higher level of assurance in meeting its deadline because such a task is often safety-critical and its correctness under very pessimistic assumptions needs to be approved by the certification authority (CA). On the other hand, the system designers' objective is to ensure the correctness of both high- and low-critical tasks but under relatively less pessimistic assumptions. The different concerns and pessimism between the CA and system designer makes it challenging to develop a real-time multiprocessor scheduling strategy for MC system.

Parallel programming paradigm allows both inter- and intra-task parallelism to effectively exploit the processing capacity of a parallel multicore architecture: each real-time task can be implemented using a task-based parallel programming model such as OpenMP4.0 [33], where the dependencies between sequential chunks of computation (called, subtasks) are specified by programmers. Thus, each parallel task can be viewed as a direct acyclic graph (DAG), where the nodes are subtasks and edges are dependencies (called, precedence constraints) between the subtasks. This paper presents a scheduling algorithm and its analysis for a collection of dual-criticality sporadic DAG tasks on multiprocessors where each task is either a high-critical (HI) task or a low-critical (LO) task.

Several works on scheduling (non-MC) sporadic DAG tasks on multiprocessors [25, 31] abstract the complex internal structure of each DAG task using only two parameters: *total work* and *critical-path length*. The total work of a task τ_i is the sum of the worst-case execution times (WCETs) of all the subtasks of task τ_i. The critical-path length of task τ_i is the maximum sum of the WCETs of the subtasks that belong to any source-to-sink path of task τ_i. Li et al. [27] proposed mixed-critical DAG task model by associating a *nominal* and an *overload* value for the total work and critical-path length for each DAG task. The nominal and overload total work of a DAG task τ_i are respectively denoted by C_i^N and C_i^O such that $C_i^N \leq C_i^O$. Similarly, the nominal and overload critical-path length are respectively denoted by L_i^N and L_i^O where $L_i^N \leq L_i^O$. Li et al. [26] recently (September, 2017) proposed federated scheduling of implicit-deadline MC sporadic DAG tasks, called MCFS-Improve, which is an improvement of their original work in [27].

The basic idea of federated scheduling is the following. Each MC task τ_i with overload utilization larger than 1 is assigned to a set of dedicated processors and all the low-utilization tasks are assigned on the remaining processors. Each task is also assigned a virtual deadline [4] such that the task meets its deadline if it does not overrun its nominal total work and critical-path length. The runtime system has two states: *typical* and *critical*. Each task initially starts in typical state. The state of the system is switched from typical to the critical state when some job does not signal completion by its virtual deadline. During the critical state, the LO-critical tasks may need to be discarded to allocate additional computing resource to the HI-critical tasks so that each HI-critical task meets its deadlines during the critical state. Li et al. [27, 26] proposed a very interesting algorithm to assign a collection of MC sporadic DAG tasks to a given number of processors and apply a schedulability test to determine whether the assignment guarantees the MC-correctness of the system or not (the formal definition of MC-correctness will be presented shortly).

By carefully analyzing the task-assignment algorithm in [26], we observed that the number of dedicated processors required for individual high-utilization task does not take into account how many processors are required for the other tasks. Consequently, the task assignment in MCFS-Improve [26] may declare failure due to not having enough number of processors for all the tasks even if there exists another way of allocating dedicated processors to individual high-utilization task. Our second observation is that the task assignment algorithm in MCFS-Improve does not explicitly consider to maximize the number of LO-critical tasks that do not need to be discarded in the critical state. Maximizing the number of LO-critical tasks that are never discarded is important to improve the QoS of the system.

The task assignment algorithm is very crucial for guaranteeing the MC-correctness for federated scheduling on multiprocessors. Since the problem of assigning tasks to the processors (even for sequential tasks) is NP-hard in the strong sense, designing an effective task assignment algorithm for federated scheduling is not only important but also more challenging for parallel tasks in comparison to sequential tasks. To this end, we propose a new task assignment algorithm for federated scheduling, called Mixed-Criticality Federated Scheduling with QoS (MCFQ), and empirically show that the performance is significantly better in terms of both schedulability and improving the QoS of the system in comparison to MCFS-Improve.

The main feature of MCFQ algorithm is that it finds *different* alternative ways to assign individual high-utilization task to different number of dedicated processors based on a new schedulability test. After all the different alternatives to assign each high-utilization task are computed, we carefully select one particular alternative for each high-utilization task such that *all* the tasks can be successfully assigned to the available number of processors. In contrast to the task-assignment algorithm in [27, 26] that makes "local" decision about task assignment when analyzing each individual high-utilization task separately, we make a "global" decision by taking into account how processors can be intelligently allocated to the tasks so that there are enough processors for all the tasks. The main contributions of this paper are the following:

- A new federated scheduling algorithm MCFQ for a set of implicit-deadline MC sporadic DAG tasks on M processors is proposed. A new schedulability analysis for the high-utilization and HI-critical tasks is proposed. The main outcome of the analysis is a polynomial-time schedulability test that can be used to determine different alternatives for allocating such tasks to dedicated processors. Based on the different alternatives for assigning the high-utilization and HI-critical tasks, we ultimately find different alternatives to assign *all* the tasks to the processors such that MC-correctness for each such alternative is guaranteed.
- We select the alternative to assign all the tasks that minimizes the total number of processors required during the critical state, which maximizes the number of *unused* processor during the critical state. The unused processors during the critical state are used to meet the demand of additional computing capacity of the HI-critical tasks rather than discarding some or all the LO-critical tasks. We apply Integer Linear Programming (ILP) to maximize the number of such non-discarding LO-critical tasks.
- Empirical investigation using randomly-generated tasksets shows that both the number of schedulable tasksets and the QoS of the system using MCFQ algorithm are significantly higher than the state-of-the-art MCFS-Improve algorithm.

The remainder of this paper is organized as follows. Section 2 presents the system model and useful definitions that are used in this paper. An overview of the MCFQ algorithm is presented in Section 3. The detailed schedulability analysis of the MCFQ algorithm is presented in Section 4. Empirical investigation is presented in Section 5. Finally, related works are presented in Section 6 before concluding in Section 7.

2 System Model and Useful Definitions

We consider scheduling a set $\Gamma = \{\tau_1, \ldots \tau_n\}$ of n implicit-deadline MC sporadic DAG tasks on M identical processors such that each processor has a (normalized) speed of one. Each task τ_i is characterized by the tuple $(Z_i, T_i, D_i, C_i^N, C_i^O, L_i^N, L_i^O)$ where

- $Z_i \in \{\texttt{HI}, \texttt{LO}\}$ is the criticality of the task: LO and HI specifies that task τ_i is a low-critical task and a high-critical task, respectively;
- $T_i \in \mathbb{R}^+$ is the minimum inter-arrival time of the jobs (i.e., called the period) of the task;
- $D_i \in \mathbb{R}^+$ is the relative deadline the task such that $D_i = T_i$;
- C_i^N and C_i^O are the maximum nominal and maximum overload total work for any job of task τ_i where $C_i^N \leq C_i^O$ for $Z_i = \texttt{HI}$ and $C_i^O = C_i^N$ for $Z_i = \texttt{LO}$; and
- L_i^N and L_i^O are the maximum nominal and maximum overload critical-path length for any job of task τ_i where $L_i^N \leq L_i^O$ for $Z_i = \texttt{HI}$ and $L_i^O = L_i^N$ for $Z_i = \texttt{LO}$.

If a job of task τ_i is released at time r, then it must complete its execution by time $(r + D_i)$. The nominal and overload utilizations of task τ_i are respectively denoted by u_i^N and u_i^O such that $u_i^N = C_i^N/D_i$ and $u_i^O = C_i^O/D_i$. If $u_i^O > 1$, then task τ_i is a *high-utilization* task; otherwise, it is a *low-utilization* task. Based on the overload utilization and the criticality, the tasks in set Γ are categorized in four *disjoint* subsets $\Gamma_{\mathtt{HH}}$, $\Gamma_{\mathtt{HL}}$, $\Gamma_{\mathtt{LH}}$, and $\Gamma_{\mathtt{LL}}$ as follows:

$$\Gamma_{\mathtt{HH}} = \{\tau_i \mid u_i^O > 1 \text{ and } Z_i = \mathtt{HI}\} \qquad \Gamma_{\mathtt{LH}} = \{\tau_i \mid u_i^O > 1 \text{ and } Z_i = \mathtt{LO}\}$$
$$\Gamma_{\mathtt{HL}} = \{\tau_i \mid u_i^O \le 1 \text{ and } Z_i = \mathtt{HI}\} \qquad \Gamma_{\mathtt{LL}} = \{\tau_i \mid u_i^O \le 1 \text{ and } Z_i = \mathtt{LO}\}$$

Note that $\Gamma = \Gamma_{\mathtt{HH}} \cup \Gamma_{\mathtt{LH}} \cup \Gamma_{\mathtt{HL}} \cup \Gamma_{\mathtt{LL}}$. We will use the following lemmas later in this paper.

▶ **Lemma 1.** *Consider a* `MC` *DAG task τ_i. The following property is satisfied:*

$$(C_i^O - C_i^N) \ge (L_i^O - L_i^N) \tag{1}$$

Proof. By the definition of total work and critical-path length, it is evident that the total work includes the work on the critical path. Therefore, the difference between the overload and nominal total work is larger than or equal to the difference between the overload critical-path length and nominal critical-path length. Therefore, $(C_i^O - C_i^N) \ge (L_i^O - L_i^N)$. ◀

▶ **Lemma 2.** *Consider a job J of a DAG task τ that is released at time r and executes on m dedicated processors using a work-conserving algorithm*[1] *where $m \ge 1$. If the remaining total work and the remaining length of the critical path at time $(r+t)$ are respectively C and L where $t \ge 0$, then job J completes its execution no later than at time $(r+R)$ such that*

$$R \le t + L + \frac{C-L}{m} \tag{2}$$

Proof. Since $(r+R)$ is the time at which the job completes its execution, there is at least one processor busy executing the nodes of job J in the interval $[r+t, r+R]$. Let ℓ is the cumulative length of intervals in $[r+t, r+R]$ during which at least one processor is idle where $\ell \ge 0$. Therefore, all the m processors are simultaneously busy for a cumulative length of intervals equal to $(R-t-\ell)$ in the interval $[r+t, r+R]$.

Since the remaining length of the critical path decreases when there is at least one processor idle, we have $0 \le \ell \le L$. Therefore, the total work completed during the interval $[r+t, r+R]$ is at least $\ell + m \cdot (R-t-\ell)$. Since the the maximum remaining total work at time $(r+t)$ is C, we have

$$\begin{aligned} & \ell + m \cdot (R-t-\ell) \le C \\ & (\text{since } m \ge 1 \text{ and } \ell \le L) \\ \Rightarrow\ & L + m \cdot (R-t-L) \le C \\ \Leftrightarrow\ & R \le t + L + \frac{C-L}{m} \end{aligned}$$ ◀

Our proposed `MCFQ` scheduling algorithm assigns a virtual deadline, denoted by D_i^v, to each task τ_i. As will be evident later, the virtual deadline for each task is assigned such that each job of task τ_i is guaranteed to meet its deadline by it virtual deadline if the total work and critical-path length does not exceed their nominal values C_i^N and L_i^N, respectively.

[1] A work conserving algorithm is any scheduling algorithm that never idles a processor if there is a node waiting for execution.

States. The system operates either in typical or critical state. The system starts in typical state. If each job of each task τ_i signals completion by its virtual deadline D_i^v, then the system remains in the *typical state*. If any job does not complete by its virtual deadline (i.e., either the total work or critical-path length exceeds the nominal value), then the system is said to *switch* from typical to critical state. Once the system switches to the critical state, jobs of the LO-critical tasks may be discarded. The system remains in the critical state if each job of the HI-critical task signals completion without overrunning its overload total work C_i^O and overload critical-path length L_i^O. All other states are erroneous.

Correctness. We define an algorithm for scheduling a set of MC tasks to be correct if the following properties are satisfied:

- During the typical state, all the jobs of each task meet their deadlines.
- During the critical state, all the jobs of each HI-critical task meets their deadlines.

It is evident from the definition of correctness that if the state of the system is changed from typical to critical, then the runtime scheduler can discard the execution of such LO-critical tasks during its critical state in order to provide additional computing resource to ensure the correctness of the HI-critical tasks. The system can switch back from critical to typical state based on the approach proposed by Li et al. [26, p. 794].

3 An Overview of the MCFQ Algorithm

The MCFQ scheduling works in two phases: an offline task assignment phase and an online runtime scheduling phase. In this section, we present an overview of the task-assignment phase and the runtime scheduler of MCFQ. In Section 4, we present the details of the task-assignment phase, present the schedulability analysis, and prove the correctness of MCFQ.

Task Assignment Phase. This phase determines the mapping of the tasks to the processors and also computes a virtual deadline D_i^v for each task τ_i. The idea of virtual deadline, originally proposed for EDV-VD scheduling of sequential tasks [4], is also used by Li et al. [27, 26] for MCFS-Improve algorithm. The virtual deadline of each task τ_i is used by the runtime scheduler to determine whether the system needs to switch from typical to critical state or not. The method to compute the virtual deadline will be presented shortly.

The MCFQ scheduling algorithm assigns each task to the processors based on whether it is a high- or low-utilization task. It assigns each high-utilization (i.e., $u_i^O > 1$) task $\tau_i \in (\Gamma_{\texttt{HH}} \cup \Gamma_{\texttt{LH}})$ to a set of dedicated processors. We denote π_i^N and π_i^O the number of dedicated processors assigned to a high-utilization task τ_i for the typical and critical states, respectively. The number of dedicated processors assigned to each HH task $\tau_i \in \Gamma_{\texttt{HH}}$ for the typical and critical states satisfies $\pi_i^N \leq \pi_i^O$. A HH task τ_i is assigned additional $(\pi_i^O - \pi_i^N)$ processors to guarantee its correctness only if τ_i does not complete by its virtual deadline.

For each LH task τ_i, the task-assignment only determines the number of dedicated processors π_i^N for the typical state. A LH task τ_i may need to provide its π_i^N processors (by discarding τ_i) to some HI-critical task τ_k during the critical state. Therefore, π_i^O=0 for each LH task τ_i if such a task is dropped; otherwise, π_i^O=π_i^N to specify that τ_i is never dropped.

We assign the low-utilization tasks in set $\Gamma_{\texttt{HL}} \cup \Gamma_{\texttt{LL}}$. Since $u_i^O \leq 1$ for each task $\tau_i \in (\Gamma_{\texttt{HL}} \cup \Gamma_{\texttt{LL}})$, we have $C_i^N \leq C_i^O \leq D_i$. Therefore, such a low-utilization task τ_i can execute sequentially and does not necessarily require parallelism to meet its deadline. Similar to [27, 26], the MCFQ algorithm also assigns all the low-utilization tasks using the MC-Partition-0.75 algorithm proposed in [6]. By applying MC-Partition-0.75 algorithm [6] on all the low-utilization tasks, we determine the *minimum* number of processors required to ensure the correctness of these low-utilization tasks. Note that MC-Partition-0.75 algorithm allocates for all the low-utilization tasks the same number of processors for both the typical and critical states. Let $\Pi_{\texttt{LU}}$ is the minimum number of processors (computed by applying MC-Partition-0.75 algorithm) required for the correctness of all the low-utilization tasks during the typical and critical states. The minimum can be found by applying a bisection search.

After the number of processors determined for each high-utilization task and for all the low-utilization tasks for the typical and critical states is determined, we apply the **capacity constraint**: *if the total number of processors required by all the tasks during each individual state is not more than M (i.e., number of available processors), then the task-assignment phase declares success; otherwise, it declares failure.*

Before the task assignment phase starts, we assume that all the `LH` tasks may need to be dropped (i.e., we assume $\pi_i^O = 0$ for each $\tau_i \in \Gamma_{\texttt{LH}}$). After the `MCFQ` algorithm finds a successful assignment for all the tasks by assuming that all `LH` tasks are dropped, it may be the case that the total number of processors required during the critical state for all the tasks is smaller than M. In other words, there may be (unused) processors that have no task assigned during the critical state. If the number of such unused processors during the critical state is more than π_i^N for some `LH` task τ_i, then we set $\pi_i^O{=}\pi_i^N$ to specify that such a `LO`-critical task is never dropped. Such adjustment will not compromise the schedulability of the `HH` task τ_k because the additional $(\pi_k^O - \pi_k^N)$ processors to the `HH` task τ_k during the critical state can be assigned from the set of idle processors rather than discarding the `LH` task τ_i during the critical state.

Run-Time Scheduler. The runtime scheduler of `MCFQ` algorithm works as follows:

- The system starts in typical state. During the typical state,
  - the nodes of each high-utilization task τ_i are scheduled using any work conserving scheduling algorithm on π_i^N number of dedicated processors; and
  - the nodes of all the low-utilization tasks are scheduled on $\Pi_{\texttt{LU}}$ processors on which they are assigned by the MC-Partition-0.75 algorithm.
- If any `HI`-critical task τ_i does not signal completion by its virtual deadline D_i^v, then the system switches from typical to the critical state, and
  - If $u_i^O > 1$, then one by one active (i.e., not dropped yet) `LH` task τ_k for which $\pi_k^O = 0$ is dropped until additional $(\pi_i^O - \pi_i^N)$ processors to the `HH` task τ_i are assigned. The nodes of the `HH` task τ_i are now scheduled using a work conserving scheduling algorithm on π_i^O dedicated processors.
  - If $u_i^O \leq 1$, the all the `LL` tasks are dropped and the `HL` tasks are scheduled on the $\Pi_{\texttt{LU}}$ processors on which they are assigned by the MC-Partition-0.75 algorithm.

Note that if some `HL` task τ_i (i.e., $u_i^O \leq 1$) triggers the switching of system's state from typical to critical, then all the `LL` tasks are dropped (no `LH` task is dropped) since all the `HL` tasks (according to [6]) still meets their deadline on $\Pi_{\texttt{LU}}$ processors during the critical state. If some `HH` task τ_i (i.e., $u_i^O > 1$) triggers the switching of system's state from typical to critical, then adequate number of `LH` tasks are dropped to assign the `HH` task τ_i additional $(\pi_i^O - \pi_i^N)$ processors. The remaining (not yet dropped) `LH` tasks may continue execution until some other `HH` task does not complete by its virtual deadline. Therefore, the system may degrade gracefully as is pointed by Li et al.in [27, 26].

Practicality of Federated Scheduling. The practical consideration of federated scheduling of parallel DAG tasks is discussed in [25] by pointing out that there is no preemption on any high-utilization task since each such task has a dedicated number of processors. Note that a low priority parallel task in global scheduling (all processors are shared) or partitioned scheduling (more than one task may share a dedicated subset of the processors) may suffer from preemption. Li et al. in [26] also developed a reference system written in OpenMP by implementing the `MCFS-Improve` scheduling in Linux using the RT_PREEMPT patch as the underlying RTOS. It has been experimentally shown in [26] that the overhead of real implementation of federated scheduling for parallel `MC` tasks is low. Since the runtime scheduling of `MCFS-Improve` and our proposed `MCFQ` algorithms are fundamentally the same, the `MCFQ` algorithm can also be implemented the same way as in [26] and is also expected to have very low implementation overhead.

4 Schedulability Analysis and Task Assignment of MCFQ Algorithm

This section presents the task assignment strategy of `MCFQ` algorithm. The schedulability analysis of the tasks in sets $\Gamma_{\texttt{LH}}$ and $\Gamma_{\texttt{HH}}$ are presented in subsections 4.1 and 4.2, respectively. Recall that the MC-Partition-0.75 is used to determine the minimum number of processors $\Pi_{\texttt{LU}}$ to correctly schedule the tasks in set $(\texttt{HL} \cup \texttt{LL})$. The total number of processors required to guarantee the correctness for all the tasks in Γ is determined in subsection 4.3.

4.1 Task Assignment: LH tasks

In this section, the number of processors π_i^N required to ensure the correctness of `LH` task τ_i is determined. The virtual deadline for each `LH` task τ_i is $D_i^v = D_i$. For the time being, we assume π_i^O=0 for each `LH` task τ_i (such a task is dropped during the critical state). In subsection 4.4, we will determine which `LH` tasks do not need to be dropped and we reset $\pi_i^O = \pi_i^N$ for such `LH` tasks.

▶ **Lemma 3.** *The execution of each LH task $\tau_i \in \Gamma_{LH}$ is correct using the runtime scheduler of MCFQ algorithm if task τ_i is assigned π_i^N dedicated processors during the typical state where*

$$\pi_i^N = \lceil (C_i^N - L_i^N)/(D_i - L_i^N) \rceil \tag{3}$$

Proof. The proof is same as the proof in [26, (Lemma 2, p. 771)]. ◀

4.2 Task Assignment: HH Tasks

In this subsection, the schedulability analysis of each `HH` task τ_i in order to determine the number of processors required to ensure its correctness during the typical and critical states is presented. Each `HH` task τ_i is also assigned a virtual deadline D_i^v.

The outcome of the analysis is a schedulability test, denoted by $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$, where μ_i^N and μ_i^O are respectively the number of dedicated processors assigned to task τ_i during the typical and critical states such that $1 \le \mu_i^N \le \mu_i^O$. If the schedulability test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ is satisfied, then it is guaranteed that task τ_i meets its deadline where $\mu_i^N$and $\mu_i^O$are the number of dedicated processors for τ_i during the typical and critical state, respectively.

Since there are M processors on the multiprocessor platform, we apply $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ for all possible pairs of (μ_i^N, μ_i^O) where $\mu_i^N = 1, 2, \ldots M$ and $\mu_i^O = \mu_i^N, \mu_i^N + 1, \ldots M$ to determine the valid pairs of (μ_i^N, μ_i^O) for which `HH` task τ_i meets its deadline during the typical and critical states. From all the valid pairs (μ_i^N, μ_i^O) for each `HH` task $\tau_i \in \Gamma_{\texttt{HH}}$, we select one pair for each `HH` task τ_i as the final values of π_i^N and π_i^O. The opportunity to select the values of π_i^N and π_i^O from the different possible pairs has higher likelihood of satisfying the capacity constraints of the platform, which is demonstrated using the following example.

▶ **Example 4.** Consider a multiprocessor platform $M = 16$ and a taskset with three high-utilization `MC` tasks. There is one `LH` task τ_a and two `HH` tasks τ_b and τ_c. The specific values of the total work and critical-path length of these tasks are not needed to understand this example. Assume that $\pi_a^N = 5$ for the `LH` task τ_a and $\pi_a^O = 0$. Also assume that the `SCHH`$(\tau_b, \mu_b^N, \mu_b^O)$ test is satisfied for only one pair $(\mu_b^N, \mu_b^O) = (4, 9)$ for task τ_b. Since there is only one pair $(\mu_b^N, \mu_b^O) = (4, 9)$ for `HH` task τ_b, we have only one option for selecting the final values of π_b^N and π_b^O such that $\pi_b^N = \mu_b^N = 4$ and $\pi_b^O = \mu_b^O = 9$.

Finally, consider that `SCHH`$(\tau_c, \mu_c^N, \mu_c^O)$ is satisfied for two different pairs $(\mu_c^N, \mu_c^O) = (5, 8)$ and $(\mu_c^N, \mu_c^O) = (6, 7)$ for task τ_c. Since there are two possible pairs of (μ_c^N, μ_c^O) for task τ_c, there are two possible ways to select the final values of π_c^N and π_c^O for task τ_c.

If we select $(\pi_c^N, \pi_c^O) = (\mu_c^N, \mu_c^O) = (5, 8)$ for task τ_c, the total number of processors for the three tasks τ_a, τ_b and τ_c during the typical state is $(5 + 4 + 5) = 14$, which is not larger than $M = 16$. The total number of processors for the three tasks τ_a, τ_b and $\tau_c$during the critical state is $(0 + 9 + 8) = 17$, which is *larger* than $M = 16$. Consequently, the capacity constraint is *not* satisfied and the overall task allocation phase declares failure.

If we select $(\pi_c^N, \pi_c^O) = (\mu_c^N, \mu_c^O) = (6, 7)$ for task τ_c, the total number of processors for the three tasks τ_a, τ_b and τ_c during the typical state is $(5 + 4 + 6) = 15$, which is not larger than $M = 16$. The total number of processors for all the three tasks during the critical state is $(0 + 9 + 7) = 16$, which is not larger than $M = 16$. Consequently, the capacity constraint is satisfied for both states and the overall task allocation phase declares success. ◀

Example 4 demonstrates that the selection of the final values of π_i^N and π_i^O for each of the `HH` tasks τ_i from the different alternative pairs of (μ_i^N, μ_i^O) is crucial to the overall success of the task assignment algorithm of `MCFQ`. Before we present the schedulability test `SCHH`$(\tau_i, \mu_i^N, \mu_i^O)$ in Lemma 5, we present how virtual deadline D_i^v is assigned to τ_i.

Virtual Deadline Assignment. Consider that the number of dedicated processors for `HH` task τ_i during the typical and critical states are μ_i^N and μ_i^O, respectively. The virtual deadline D_i^v for `HH` task τ_i is assigned as follows:

$$D_i^v = L_i^N + (C_i^N - L_i^N)/\mu_i^N \tag{4}$$

▶ **Lemma 5** (Schedulability test `SCHH`$(\tau_i, \mu_i^N, \mu_i^O)$). *Consider a pair (μ_i^N, μ_i^O) such that the `HH` task τ_i is assigned μ_i^N and μ_i^O dedicated processors respectively for the typical and critical states where $1 \le \mu_i^N \le \mu_i^O$. Each job of task τ_i meets its deadline in all the correct states if the following equation is satisfied:*

$$D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O + min\{L_i^N, \frac{\omega_i}{\mu_i^N}\} \cdot (1 - \frac{\mu_i^N}{\mu_i^O}) \tag{5}$$

where $\omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)$.

Proof. Since $(C_i^O - C_i^N) - (L_i^O - L_i^N) \ge 0$ from Eq. (1), we have $\omega_i \ge 0$. Moreover, $L_i^N \ge 0$ and $\mu_i^N \ge 1$. It follows that $min\{L_i^N, \omega_i/\mu_i^N\} \ge 0$. Because $\mu_i^O \ge \mu_i^N$, we also have $(1 - \mu_i^N/\mu_i^O) \ge 0$. Therefore, $min\{L_i^N, \omega_i/\mu_i^N\} \cdot (1 - \mu_i^N/\mu_i^O) \ge 0$ and from Eq (5) we have

$$D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O \tag{6}$$

Consider a generic job J_i of task τ_i. Without loss of generality assume that the job is released at time 0. The entire execution of job J_i happens in any of the three possible scenarios: (i) stable typical state, (ii) stable critical state, and (iii) during the transition from typical to critical state. A stable state refers to the situation when there is no switching of states during the execution of job J_i. This lemma is proved by showing that job J_i meets its deadline for all these three scenarios if Eq (5) is satisfied. Since we are considering implicit-deadline tasks, if the generic job J_i meets its deadline by time D_i, then each other job of τ_i will also meet its deadline.

Stable typical state. During the stable typical state, the subtasks of task τ_i are executed using any work-conserving scheduling algorithm on μ_i^N dedicated processors. Since job J_i executes entirely in stable typical state, it signals completion at or before its virtual deadline D_i^v. We will show that $D_i^v \le D_i$, which shows job J_i meets its deadline during the stable typical state. Since $L_i^O \ge L_i^N$, from Eq. (6) we have $D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^N$. Since $\frac{\omega_i}{\mu_i^O} \ge 0$ and $D_i^v = \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N$ from Eq. (4), it follows that $D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N = D_i^v$.

Stable critical state. During the stable critical state, the subtasks of task τ_i are executed using any work-conserving scheduling algorithm on μ_i^O dedicated processors. The total work and the critical-path length of any job of task τ_i during the critical state is at most C_i^O and L_i^O, respectively. Based on Lemma 2, the maximum time job J_i takes to finish its execution starting from its release at time 0 is $L_i^O + (C_i^O - L_i^O)/\mu_i^O$. We will show that $L_i^O + (C_i^O - L_i^O)/\mu_i^O \le D_i$, which implies that job J_i meets its deadline during the stable critical state. Since $\omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)$, from Eq. (6) we have

$$D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^O} + L_i^O$$

$$\Leftrightarrow \quad D_i \ge \frac{C_i^O - L_i^O}{\mu_i^O} + (C_i^N - L_i^N) \cdot (\frac{1}{\mu_i^N} - \frac{1}{\mu_i^O}) + L_i^O$$

(Since $C_i^N \ge L_i^N$ because total work includes the work on the critical path and $\mu_i^O \ge \mu_i^N$, we have $(C_i^N - L_i^N) \cdot (1/\mu_i^N - 1/\mu_i^O) \ge 0$)

$$\Rightarrow \quad D_i \ge \frac{C_i^O - L_i^O}{\mu_i^O} + L_i^O$$

State Switching. For this case, the job J_i does not complete execution by its virtual deadline D_i^v and it switches from typical to critical state at time D_i^v. The subtasks of job J_i execute on μ_i^N processors during the interval $[0, D_i^v)$ and on μ_i^O processors after time D_i^v.

Let ℓ be the cumulative length of intervals in $[0, D_i^v)$ during which at least one of the μ_i^N dedicated processors assigned to job J_i of task τ_i is idle such that $0 \le \ell \le D_i^v$. Since J_i is not finished by time D_i^v and because at least one processor is idle for a duration of ℓ time units in $[0, D_i^v)$, the length of the critical path by time D_i^v is decreased by at least ℓ time units. The remaining length of the critical path at time D_i^v, denoted by L_{remain}, is at most

$$L_{remain} = (L_i^O - \ell) \tag{7}$$

where L_i^O is the overload critical-path length of τ_i. The cumulative length of intervals in $[0, D_i^v)$ during which all the μ_i^N processors are simultaneously busy is $(D_i^v - \ell)$. Therefore, the amount of work done before the task τ_i switches its state at time D_i^v is at least $[\ell + \mu_i^N \cdot (D_i^v - \ell)]$. The remaining amount of total work at time D_i^v, denoted by C_{remain}, is at most

$$C_{remain} = C_i^O - [\ell + \mu_i^N \cdot (D_i^v - \ell)] = C_i^O - \ell - \mu_i^N \cdot (D_i^v - \ell) \tag{8}$$

where C_i^O is the overload total work of task τ_i. Since the total remaining work includes the remaining work of the critical path, we have

$$
\begin{aligned}
& L_{remain} \leq C_{remain} \\
& \big(\text{From Eq. (7) and Eq. (8)}\big) \\
\Leftrightarrow\ & L_i^O - \ell \leq C_i^O - \ell - \mu_i^N \cdot (D_i^v - \ell) \\
& \Big(\text{Since } D_i^v = \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N \text{ from Eq. (4)}\Big) \\
\Leftrightarrow\ & L_i^O - \ell \leq C_i^O - \ell - \mu_i^N \cdot (L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N} - \ell) \\
\Leftrightarrow\ & \mu_i^N \cdot L_i^N + L_i^O - C_i^O + C_i^N - L_i^N \leq \mu_i^N \cdot \ell \\
\Leftrightarrow\ & L_i^N - \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N} \leq \ell \\
& (\text{Since } 0 \leq \ell) \\
\Rightarrow\ & max\left\{0, L_i^N - \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N}\right\} \leq \ell \\
\Leftrightarrow\ & L_i^N - max\left\{0, L_i^N - \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N}\right\} \geq L_i^N - \ell \\
\Leftrightarrow\ & min\left\{L_i^N, \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N}\right\} \geq L_i^N - \ell \\
& \big(\text{Since } \omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)\big) \\
\Leftrightarrow\ & min\left\{L_i^N, \frac{\omega_i}{\mu_i^N}\right\} \geq L_i^N - \ell
\end{aligned} \tag{9}
$$

Since μ_i^O processors are assigned to job J_i from time D_i^v, the job J_i completes its execution no later than time $D_i^v + L_{remain} + \frac{C_{remain} - L_{remain}}{\mu_i^O}$ according to Lemma 2. We will show that $D_i^v + L_{remain} + \frac{C_{remain} - L_{remain}}{\mu_i^O} \leq D_i$, which implies that J_i completes at or before its deadline. We have to prove that the following holds:

$$
\begin{aligned}
& D_i^v + L_{remain} + \frac{C_{remain} - L_{remain}}{\mu_i^O} \leq D_i \\
& (\text{From Eq. (7) and Eq. (8)}) \\
\Leftrightarrow\ & D_i^v + L_i^O - \ell + \frac{[C_i^O - \ell - \mu_i^N \cdot (D_i^v - \ell)] - (L_i^O - \ell)}{\mu_i^O} \leq D_i \\
\Leftrightarrow\ & D_i^v + L_i^O - \ell + \frac{C_i^O - \mu_i^N \cdot (D_i^v - \ell) - L_i^O}{\mu_i^O} \leq D_i \\
& (\text{Since } D_i^v = \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N \text{ from Eq. (4)} \\
\Leftrightarrow\ & L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^O - \ell + \frac{C_i^O - \mu_i^N \cdot \left(L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N}\right) + \mu_i^N \cdot \ell - L_i^O}{\mu_i^O} \leq D_i \\
\Leftrightarrow\ & L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^O - \ell + \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^O} - \frac{\mu_i^N \cdot (L_i^N - \ell)}{\mu_i^O} \leq D_i \\
& (\text{Since } \omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)) \\
\Leftrightarrow\ & \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O + (L_i^N - \ell) \cdot (1 - \frac{\mu_i^N}{\mu_i^O}) \leq D_i
\end{aligned}
$$

$$\begin{aligned}
&\text{(From Eq. (9), } min\left\{L_i^N, \frac{\omega_i}{\mu_i^N}\right\} \geq (L_i^N - \ell))\\
\Leftarrow\quad & \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O + min\left\{L_i^N, \frac{\omega_i}{\mu_i^N}\right\} \cdot (1 - \frac{\mu_i^N}{\mu_i^O}) \leq D_i\\
\Leftrightarrow\quad & Eq.\ (5)
\end{aligned}$$

Therefore, the generic job J_i of HH task τ_i meets its deadline in all the three scenarios. ◀

For each HH task τ_i, we can apply the schedulability test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ in Eq. (5) to determine whether the HH task τ_i meets its deadline in all correct states if μ_i^N and μ_i^O number of dedicated processors are assigned during the typical and critical states, respectively. We say that $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$=TRUE if Eq. (5) is satisfied; otherwise $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$=FALSE. The salient feature of the schedulability test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ is that the set of all possible pairs (μ_i^N, μ_i^O) where $1 \leq \mu_i^N \leq \mu_i^O \leq M$ for which HH task τ_i is deemed schedulable in all the correct states can be determined. The elements in each such pair are potential final values of π_i^N and π_i^O for task τ_i. To this end, we define $\overline{\Omega}(\tau_i)$ the set of all such valid pairs (μ_i^N, μ_i^O) for which the HH task τ_i is schedulable in any state as follows:

$$\overline{\Omega}(\tau_i) = \left\{(\mu_i^N, \mu_i^O) \mid \texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O) = \texttt{TRUE};\ \mu_i^N = 1, 2 \ldots M;\ \mu_i^O = \mu_i^N \ldots M\right\} \qquad (10)$$

We now filter some of the unnecessary elements from set $\overline{\Omega}(\tau_i)$ to limit the number of valid pairs. Consider that $\texttt{SCHH}(\tau_i, 1, 1)$=FALSE, $\texttt{SCHH}(\tau_i, 1, 2)$=TRUE and $\texttt{SCHH}(\tau_i, 1, 3)$=TRUE for some HH task τ_i. Based on Eq. (10), we have $(1, 1) \notin \overline{\Omega}(\tau_i)$ and $\{(1, 2), (1, 3)\} \subseteq \overline{\Omega}(\tau_i)$. However, we may discard the element $(1, 3)$ from set $\overline{\Omega}(\tau_i)$ since when μ_i^N=1 it is unnecessary (wastage of resource) to consider μ_i^O=3 because μ_i^O=2 processors are enough to guarantee the correctness of task τ_i during the critical state. Therefore, we only need to consider such pair $(\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i)$ where $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1)$=FALSE. To this end, we define $\Omega(\tau_i)$ the set of pairs (μ_i^N, μ_i^O) from set $\overline{\Omega}(\tau_i)$ for which $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1)$=FALSE as follows:

$$\Omega(\tau_i) = \left\{(\mu_i^N, \mu_i^O) \mid (\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i); \texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1) = \texttt{FALSE}\right\} \qquad (11)$$

Note that Eq. (5) can be tested in constant time for the given values of C_i^N, L_i^N, C_i^O, L_i^O, μ_i^N and μ_i^O for HH task τ_i. The set $\overline{\Omega}(\tau_i)$ in Eq. (10) can be computed for task τ_i by applying test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ at most $M(M + 1)/2$ times since $1 \leq \mu_i^N \leq \mu_i^O \leq M$. Therefore, the time complexity to compute the set $\overline{\Omega}(\tau_i)$ for one HH task τ_i is $O(M^2)$. Since $1 \leq \mu_i^N \leq \mu_i^O \leq M$, the number of elements in $\overline{\Omega}(\tau_i)$ is $O(M^2)$. For all the $O(M^2)$ elements in set $\overline{\Omega}(\tau_i)$, we can test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1) = \texttt{FALSE}$ is Eq. (11) is time $O(M^2)$. Therefore, set $\Omega(\tau_i)$ can be computed in time $O(M^2)$. Since for each element $(\mu_i^N, \mu_i^O) \in \Omega(\tau_i)$ we have $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1)$=FALSE, it follows that the number of elements in set $\Omega(\tau_i)$ is $O(M)$ and the set $\Omega(\tau_i)$ can be computed in time $O(M^2)$.

▶ **Lemma 6.** *If $(\mu_i^N, \mu_i^O) \in \Omega(\tau_i)$, then the HH task τ_i meets all its deadlines if the number of dedicated processors during the typical and critical states are μ_i^N and μ_i^O, respectively.*

Proof. Since $(\mu_i^N, \mu_i^O) \in \Omega(\tau_i)$ only if $(\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i)$ based on Eq. (11). Moreover, if $(\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i)$, then $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O) = \texttt{TRUE}$ based on Eq. (10). Therefore, task τ_i meets all its deadlines based on Lemma 5 if the number of dedicated processors during the typical and critical states are μ_i^N and μ_i^O, respectively. ◀

▶ **Example 7.** Consider the two HH tasks in Table 1 and M=8. We only list few elements of set $\Omega(\tau_i)$ in Table 1 for simplicity of presentation. The values in Table 1 will be used later.

Table 1 Example of two HH tasks and some elements (μ_i^N,μ_i^O) in $\Omega(\tau_i)$ computed using Eq. (11)

Task	C_i^N	L_i^N	C_i^O	L_i^O	D_i	Some elements in $\Omega(\tau_i)$
τ_1	9	4	52	20	45	{(1,2), (2,2), (3,3)}
τ_2	11	4	80	42	54	{(2,6),(3,4)}

Given the set $\Omega(\tau_i)$ for each task $\tau_i \in \Gamma_{\texttt{HH}}$, we now determine the total number of processors required for correctly scheduling *all* the HH tasks in each state. Our objective is to find different alternatives to assign ***all*** the HH tasks to the processors using the different alternatives in $\Omega(\tau_i)$ for each HH task τ_i.

Without loss of generality assume that there are Q number of HH tasks in set Γ such that $Q = |\Gamma_{\texttt{HH}}|$ and the indices of the HH tasks in set $\Gamma_{\texttt{HH}}$ ranges from 1 to Q such that $\Gamma_{\texttt{HH}} = \{\tau_1, \tau_2, \ldots \tau_Q\}$. We also define sequence $\mathcal{S}^p_{\texttt{HH}} =< \tau_1, \tau_2, \ldots \tau_p >$ that includes the HH tasks with indices from 1 to p, for $p = 1, 2, \ldots Q$. Note that the sequence $\mathcal{S}^Q_{\texttt{HH}}$ includes all the tasks in $\Gamma_{\texttt{HH}}$. Given the sequence of p tasks in $\mathcal{S}^p_{\texttt{HH}}$, we denote $\xi(\mathcal{S}^p_{\texttt{HH}})$ as the set where

- each element in set $\xi(\mathcal{S}^p_{\texttt{HH}})$ is a *pair of sequences* such that for each such pair of sequences
  - each sequence has p numbers;
  - the i^{th} element in the first sequence is the number of processors required to meet the deadline of the i^{th} HH task in sequence $\mathcal{S}^p_{\texttt{HH}}$ during the typical state, and
  - the i^{th} element in the second sequence is the number of processors required to meet the deadline of the i^{th} HH task in sequence $\mathcal{S}^p_{\texttt{HH}}$ during the critical state.

For example, consider $\xi(\mathcal{S}^3_{\texttt{HH}})$ ={(<1,2,3>,<4,5,6>), (<2,2,4>, <3,5,5>)} for the three tasks in sequence $\mathcal{S}^3_{\texttt{HH}} =< \tau_1, \tau_2, \tau_3 >$. The interpretation of set $\xi(\mathcal{S}^3_{\texttt{HH}})$ ={(<1,2,3>,<4,5,6>), (<2,2,4>, <3,5,5>)} is the following:

- There are two elements in set $\xi(\mathcal{S}^3_{\texttt{HH}})$. Each of the two elements $(< 1, 2, 3 >, < 4, 5, 6 >)$ and $(< 2, 2, 4 >, < 3, 5, 4 >)$ is a pair of sequences, where each sequence in a pair has $p = 3$ numbers.
- The pair $(< 1, 2, 3 >, < 4, 5, 6 >)$ specifies that the number of dedicated processors required for task τ_1 (which is the 1st task in sequence $\mathcal{S}^3_{\texttt{HH}}$) during the typical and critical states are 1 and 4, respectively. Similarly, the number of dedicated processors required for task τ_3 in sequence $\mathcal{S}^3_{\texttt{HH}}$ for the typical and critical states are 3 and 6, respectively. The total number of processors for all the three tasks in set $\mathcal{S}^3_{\texttt{HH}}$ for the typical and critical state are $(1 + 2 + 3) = 6$ and $(4 + 5 + 6) = 15$, respectively.

Each element in set $\mathcal{S}^Q_{\texttt{HH}}$ specifies a particular alternative to assign *all* the HH tasks from sequence $\mathcal{S}^Q_{\texttt{HH}}$ to the processors so that the deadlines for all the HH tasks during the typical and critical states are met. After the set $\xi(\mathcal{S}^Q_{\texttt{HH}}\ )$ is computed, we select one alternative from set $\xi(\mathcal{S}^Q_{\texttt{HH}}\ )$ so that the capacity constraint for *all* the tasks in Γ is satisfied. Next we present how to compute set $\xi(\mathcal{S}^Q_{\texttt{HH}}\ )$.

4.2.1 Computing $\xi(\mathcal{S}^Q_{\texttt{HH}}\ )$

We apply dynamic programming to find set $\xi(\mathcal{S}^Q_{\texttt{HH}})$. The sum of the p numbers in the first sequence and the sum of the p numbers in the second sequence for any element in $\xi(\mathcal{S}^p_{\texttt{HH}})$ are respectively the total number of processors required during the typical and critical states for the HH tasks in $\mathcal{S}^p_{\texttt{HH}}$. Since the number of processors of the platform is M, the total number of processors required for any state must not be larger than M in order to satisfy the capacity constraint. Based on this observation, the set $\xi(\mathcal{S}^Q_{\texttt{HH}})$ is recursively computed by considering one-by-one HH task from the sequence $\mathcal{S}_{\text{HH}}^{Q}$. In other words, we first compute $\xi(\mathcal{S}_{\text{HH}}^{1})$, then we compute $\xi(\mathcal{S}_{\text{HH}}^{2})$, and continuing in this fashion, we finally compute $\xi(\mathcal{S}_{\text{HH}}^{Q})$.

The set $\xi(\mathcal{S}_{\text{HH}}^{p})$ for $p = 1$ is computed as follows:

$$\xi(\mathcal{S}_{\text{HH}}^{1}) = \xi(<\tau_1>) = \{(<a>,<b>) \mid (a,b) \in \Omega(\tau_1)\ \} \tag{12}$$

where $\Omega(\tau_1)$ is given in Eq (11). By assuming that the set $\xi(\mathcal{S}_{\text{HH}}^{p-1})$ is already computed, the set $\xi(\mathcal{S}_{\text{HH}}^{p})$ is recursively computed for $p = 2, 3, \ldots Q$ as follows:

$$\xi(\mathcal{S}_{\text{HH}}^{p}) = \Big\{\Big(<a_1, \ldots a_{p-1}, a_p>, <b_1, \ldots b_{p-1}, b_p>\Big) \mid \texttt{COND1} \wedge \texttt{COND2} \wedge \texttt{COND3} \wedge \texttt{COND4}\Big\} \tag{13}$$

where

COND1: $(<a_1, \ldots a_{p-1}>, <b_1, \ldots b_{p-1}>) \in \xi(\mathcal{S}_{\text{HH}}^{p-1})$

COND2: $(a_p, b_p) \in \Omega(\tau_p)$

COND3: $(a_1 + \ldots + a_{p-1} + a_p) \leq M$ and $(b_1 + \ldots + b_{p-1} + b_p) \leq M$

COND4: If $(a_1 + \ldots + a_p) \neq (c_1 + \ldots + c_p)$ for some $(<c_1, \ldots c_p>, <d_1, \ldots d_p>) \in \xi(\mathcal{S}_{\text{HH}}^{p})$, then add $(<a_1, \ldots a_p>, <b_1, \ldots b_p>)$ in set $\xi(\mathcal{S}_{\text{HH}}^{p})$; otherwise, if $(a_1 + \ldots + a_p) = (c_1 + \ldots + c_p)$ and $(b_1 + \ldots + b_p) < (d_1 + \ldots + d_p)$ for some $(<c_1, \ldots c_p>, <d_1, \ldots d_p>) \in \xi(\mathcal{S}_{\text{HH}}^{p})$, then add $(<a_1, \ldots a_{p-1}, a_p>, <b_1, \ldots b_{p-1}, b_p>)$ in set $\xi(\mathcal{S}_{\text{HH}}^{p})$ and remove $(<c_1, \ldots c_p>, <d_1, \ldots d_p>)$ from set $\xi(\mathcal{S}_{\text{HH}}^{p})$.

Discussion. The set $\xi(\mathcal{S}_{\text{HH}}^{p})$ in Eq. (13) is computed by selecting each element $(<a_1, \ldots, a_{p-1}>, <b_1, \ldots b_{p-1}>)$ from $\xi(\mathcal{S}_{\text{HH}}^{p-1})$ due to COND1 and each element (a_p, b_p) from $\Omega(\tau_p)$ due to COND2 such that $(a_1 + \ldots + a_{p-1} + a_p) \leq M$ and $(b_1 + \ldots + b_{p-1} + b_p) \leq M$ due to COND3. A new element $(<a_1, \ldots a_{p-1}, a_p>, <b_1, \ldots b_{p-1}, b_p>)$ is added to set $\xi(\mathcal{S}_{\text{HH}}^{p})$ only if COND4 is true, i.e., there is no other element $(<c_1, \ldots c_{p-1}, c_p>, <d_1, \ldots d_{p-1}, d_p>)$ that is already in set $\xi(\mathcal{S}_{\text{HH}}^{p})$ such that $(a_1 + \ldots + a_{p-1} + a_p) = (c_1 + \ldots + c_{p-1} + c_p)$ and $(b_1 + \ldots + b_{p-1} + b_p) \geq (d_1 + \ldots + d_{p-1} + d_p)$.

The COND4 ensures that for any two elements $(<a_1, \ldots a_{p-1}>, <b_1, \ldots b_{p-1}>)$ and $(<c_1, \ldots c_{p-1}, c_p>, <d_1, \ldots d_{p-1}, d_p>)$ where $(a_1 + \ldots + a_{p-1} + a_p) = (c_1 + \ldots + c_{p-1} + c_p)$, the element with smaller total number of processors for the critical state is included in set $\xi(\mathcal{S}_{\text{HH}}^{p})$ while the other element is not included in set $\xi(\mathcal{S}_{\text{HH}}^{p})$ (i.e., removed if included previously). Consequently, for a given total number of processors required for the tasks in sequence $\mathcal{S}_{\text{HH}}^{p}$ for the typical state, there is at most one element in set $\xi(\mathcal{S}_{\text{HH}}^{p})$. Since COND3 is satisfied, there are at most M different possibilities for the total number of processors required for the tasks for the typical state. Therefore, the number of elements in set $\xi(\mathcal{S}_{\text{HH}}^{p})$ is at most $O(M)$. We have the following Lemma 8.

▶ **Lemma 8.** *If $(<a_1, a_2, \ldots a_Q>, <b_1, b_2, \ldots b_Q>) \in \xi(\mathcal{S}_{HH}^{Q})$, then the p^{th} HH task τ_p in sequence $\mathcal{S}_{HH}^{Q}$ meets the deadline in typical and critical states if a_p and b_p dedicated processors are assigned to τ_p respectively during the typical and critical states for $p = 1, 2, \ldots Q$.*

Proof. If $(<a_1, a_2, \ldots a_Q>, <b_1, b_2, \ldots b_Q>) \in \xi(\mathcal{S}_{\text{HH}}^{Q})$, then the pair $(a_p, b_p) \in \Omega(\tau_p)$ due to COND2 for $p = 1, 2, \ldots Q$. Based on Lemma 6, it holds for each $(a_p, b_p) \in \Omega(\tau_p)$ that task τ_p meets its deadline during the typical and critical state if a_p and b_p dedicated processors are allocated to τ_p during the typical and critical state, respectively. ◀

Time Complexity to find $\xi(\mathcal{S}_{\text{HH}}^{Q})$. The time complexity to compute $\xi(\mathcal{S}_{\text{HH}}^{Q})$ is $O(n \cdot (n + M) \cdot M^2)$. Recall that there are $O(M)$ elements in $\Omega(\tau_i)$ for each τ_i in $\mathcal{S}_{\text{HH}}^{Q}$ (discussed after Eq. (11)). Therefore, the base in Eq. (12) can be computed for task τ_1 in time $O(M)$ since each element $(a_1, b_1) \in \Omega(\tau_i)$ is stored in set $\xi(\mathcal{S}_{\text{HH}}^{1})$=$\xi(<\tau_1>)$ as $(<a>,<b>)$.

The COND4 guarantees that there are at most $O(M)$ elements in set $\xi(\mathcal{S}_{\texttt{HH}}^{k})$ for $k = 1, \ldots Q$. During each step of the recursion the set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ is computed by considering one element from $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and one element from $\Omega(\tau_p)$. Since there are at most $O(M)$ elements in each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$, the time-complexity to select all the possible ways to select one element from each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$ (i.e., applying COND1 and COND2) is $O(M^2)$. And, there are $O(M^2)$ possible choices to select one element from each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$.

For each of these $O(M^2)$ selections, we apply COND3 and COND4. Given a selection $(< a_1, \ldots a_{p-1} >, < b_1, \ldots b_{p-1} >) \in \xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $(a_p, b_p) \in \Omega(\tau_p)$, we can apply COND3 in time $O(n)$ since there are $2(p-1) = O(n)$ additions to evaluate COND3. We then apply COND4 in time $O(M)$ since there can be at most $O(M)$ elements already included in $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ and the sums in COND4 are already computed during this step of the recursion. Consequently, for all the $O(M^2)$ ways to select one element from each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$, the set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ is computed in time $O((n+M) \cdot M^2)$ during the p^{th} recursive step. Since there are at most Q=O(n) tasks in sequence $\mathcal{S}_{\texttt{HH}}^{Q}$, the set $\xi(\mathcal{S}_{\texttt{HH}}^{Q})$ can be computed in time $O(n \cdot (n+M) \cdot M^2)$.

▶ **Example 9.** Consider two HH tasks in Table 1 and M=8 where $\Omega(\tau_1)$ ={(1,2), (2,2), (3,3)} and $\Omega(\tau_2)$ ={(2,6), (3,4)} for $\mathcal{S}_{\texttt{HH}}^{2} =< \tau_1, \tau_2 >$.

Based on Eq (12), we have $\xi(< \tau_1 >)$={(<1>,<2>), (<2>,<2>), (<3>,<3>)}. We will now show how to find $\xi(< \tau_1, \tau_2 >)$ based on Eq. (13). There are total $3 \times 2 = 6$ ways to select one element from each set $\xi(< \tau_1 >)$ and $\Omega(\tau_2)$ by applying COND1 and COND2. Therefore, set $\xi(< \tau_1, \tau_2 >)$ *without* applying COND3 and COND4 is

$$\begin{aligned}\xi(< \tau_1, \tau_2 >) = \{&(< 1, 2 >, < 2, 6 >), (< 1, 3 >, < 2, 4 >), (< 2, 2 >, < 2, 6 >),\\ &(< 2, 3 >, < 2, 4 >), (< 3, 2 >, < 3, 6 >), (< 3, 3 >, < 3, 4 >)\}\end{aligned}$$

After applying COND3, the element $(< 3, 2 >, < 3, 6 >)$ is not included in set $\xi(< \tau_1, \tau_2 >)$ since $(3 + 6) > M = 8$. After applying COND4, the element $(< 2, 2 >, < 2, 6 >)$ is not included in set $\xi(< \tau_1, \tau_2 >)$ since there is another element $(< 1, 3 >, < 2, 4 >)$ such that $(2 + 2) = (1 + 3)$ and $(2 + 6) > (2 + 4)$. Therefore, we have

$$\begin{aligned}\xi(< \tau_1, \tau_2 >) = \{&(< 1, 2 >, < 2, 6 >), (< 1, 3 >, < 2, 4 >),\\ &(< 2, 3 >, < 2, 4 >), (< 3, 3 >, < 3, 4 >)\}\end{aligned}$$

4.3 Overall Task Assignment: Capacity Constraint

In this subsection, we determine whether there is an assignment of all the tasks to the processors such that the total number of processors required during each of the two states is not larger than M. We will now determine a set, denoted by Π, which is a subset of $\xi(\mathcal{S}_{\texttt{HH}}^{Q})$ using which it can be verified whether the capacity constraint at each state for all the tasks is satisfied or not. The set Π is defined as follows:

$$\Pi = \begin{cases} \emptyset & \text{if } Q > 0 \text{ and } \xi(\mathcal{S}_{\texttt{HH}}^{Q}) = \emptyset \\ \{(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \mid \texttt{COND5} \wedge \texttt{COND6}\} & \text{otherwise} \end{cases} \tag{14}$$

where

COND5: $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \xi(\mathcal{S}_{\texttt{HH}}^{Q})$

COND6: $(a_1 + \ldots + a_Q + \sum_{\tau_i \in \Gamma_{\texttt{LH}}} \pi_i^N + \Pi_{\texttt{LU}}) \leq M$ and $(b_1 + \ldots + b_Q + \Pi_{\texttt{LU}}) \leq M$.

▶ **Theorem 10.** *The MCFQ scheduling algorithm correctly schedules all the tasks in set* Γ *if* $\Pi \neq \emptyset$.

Proof. Since $\Pi \neq \emptyset$, we have at least one pair $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \Pi$ such that `COND5` and `COND6` are satisfied. Since $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \xi(\Gamma_{\texttt{HH}})$ according to `COND5`, each `HH` task τ_i meets its deadline in both typical and critical state if it is assigned a_i and b_i processors according to Lemma 8.

Each `LH` task τ_i requires π_i^N dedicated processors to ensure its correctness according to Eq. (3) of Lemma 3. Therefore, the total number of dedicated processors for all the `LH` tasks to ensure their correctness is $\sum_{\tau_i \in \Gamma_{\texttt{LH}}} \pi_i^N$. The total number of processors required for scheduling all the low-utilization tasks during typical and critical state is $\Pi_{\texttt{LU}}$, where $\Pi_{\texttt{LU}}$ is the minimum number of processors required by the MC-Partition-0.75 to schedule all the low-utilization tasks in set $(\Gamma_{\texttt{HL}} \cup \Gamma_{\texttt{LL}})$.

Therefore, the total number of processors for all the tasks is $(a_1 + \ldots + a_Q + \sum_{\tau_i \in \Gamma_{\texttt{LH}}} \pi_i^N + \Pi_{\texttt{LU}})$ and $(b_1 + \ldots + b_Q + \Pi_{\texttt{LU}})$ respectively for the typical and critical state. Since $(a_1 + \ldots + a_Q + \sum_{\tau_i \in \Gamma_{\texttt{LH}}} \pi_i^N + \Pi_{\texttt{LU}}) \leq M$ and $(b_1 + \ldots + b_Q + \Pi_{\texttt{LU}}) \leq M$ based on `COND6`, the capacity constraint at each state is met, the task assignment declares success, and the system correctly schedules all the tasks based on `MCFQ` algorithm. ◀

4.4 Improving the QoS of LH Tasks

The set Π in Eq. (14) provides different alternatives to assign all the tasks to the processors by assuming that all the `LH` tasks are dropped during critical state. However, if there are unused processors during the critical state, then such unused processors may be allocated to the `HH` tasks rather than dropping the `LH` tasks during critical state. Based on this observation, we propose a scheme to maximize the number of `LH` tasks that are never dropped.

We select the alternative from set Π that minimizes the total number of processors required during the critical state for all the `HH` tasks. Let $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \Pi$ is the alternative that minimizes the total number of processors required during the critical state for all the `HH` tasks. The number of unused processors during the critical state, denoted by Π_{idle}, is computed as $\Pi_{idle} = M - (\sum_{i=1}^{Q} b_i + \Pi_{\texttt{LU}})$.

The unused processors can be allocated to the `HH` task τ_i when it does not complete by its virtual deadline and requires additional $(\pi_i^O - \pi_i^N)$ processors to ensure its correctness. By allocating the unused processors to the `HH` tasks, we may not need to drop some or any of the `LH` tasks. Given that there are Π_{idle} unused processors during the critical state, we formulate an ILP to maximum the number of `LH` tasks that are never dropped.

Let $x_i \in \{0, 1\}$ denote a decision variable whether the `LH` task τ_i may need to be dropped or not. If $x_i = 1$, then the `LH` task τ_i is never dropped and will be assigned $\pi_i^O = \pi_i^N$ dedicated processors also during the critical state. If $x_i = 0$, then the `LH` task τ_i may need to be dropped and we set $\pi_i^O = 0$. The value of decision variable x_i for $\tau_i \in \Gamma_{\texttt{LH}}$ is determined using the following ILP to maximum the number of `LH` tasks that are never dropped:

$$\begin{aligned} \underset{x_i}{\text{maximize}} \quad & \sum_{\tau_i \in \Gamma_{\texttt{LH}}} x_i \\ \text{subject to} \quad & \sum_{\tau_i \in \Gamma_{\texttt{LH}}} \pi_i^N \cdot x_i \leq \Pi_{idle} \qquad \textbf{and} \quad (x_i = 0 \textbf{ or } x_i = 1) \end{aligned} \tag{15}$$

Given the values of x_i for all the `LH` tasks, the fraction of the total number of `LH` tasks that are never dropped is $(\sum_{\tau_i \in \Gamma_{\texttt{LH}}} x_i)/|\Gamma_{\texttt{LH}}|$ and is the measure of the QoS for a given taskset under `MCFQ` algorithm. We can also improve the QoS of the `LL` tasks by allocating them to such idle processors based on partitioned EDF scheduling for sequential tasks (not addressed in this paper).

5 Empirical Investigation

The recent work by Li et al. [26] proposed the `MCFS-Improve` schedulability test for federated scheduling of `MC` parallel tasks. In this section, we present the effectiveness of our proposed schedulability test in Theorem 10 (denoted by `Our-MCFQ`) in guaranteeing the schedulability and improving the QoS of randomly generated `MC` parallel tasks in comparison to the state-of-the-art `MCFS-Improve` test in [26]. Before we present our results, we present the taskset generation algorithm.

5.1 Taskset Generation Algorithm

Since both `Our-MCFQ` and `MCFS-Improve` tests depend only on the total work and the critical-path length of each parallel task, we will directly generate these two parameters for each parallel task. We denote U^N and U^O respectively the total nominal utilization of all the tasks and total overload utilization of all the `HI`-critical tasks in a randomly generated taskset Γ such that $U^N = \sum_{\tau_i \in \Gamma} u_i^N$ and $U^O = \sum_{\tau_i \in (\Gamma_{\texttt{HH}} \cup \Gamma_{\texttt{HL}})} u_i^O$. Let $U_B = max\{U^N/M, U^O/M\}$ denotes the upper bound on normalized total system utilization. Note that $U_B \leq 1$ is a necessary condition for schedulability of taskset Γ on M processors.

The following experimental parameters are used for generating a random `MC` sporadic DAG taskset with normalized total system utilization U_B for M processors:

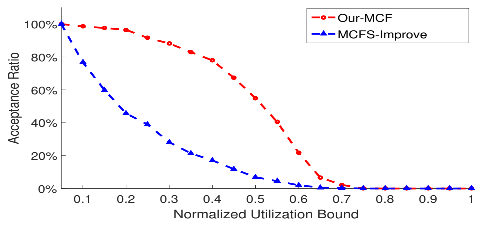
- The proportion of high-utilization tasks in a taskset is controlled using probability p^{hu}.
- The overload utilization of each high-utilization task is controlled using u_{max}.
- The ratio of the period and overload critical-path length of task τ_i is controlled using a parameter P_{max} such that $1 \leq T_i/L_i^O \leq P_{max}$.
- The proportion of `HI`-critical tasks is controlled using probability p^{hc}.
- The ratio of overload and nominal utilizations of task τ_i is controlled using a parameter R_{max} such that $1 \leq u_i^O/u_i^N \leq R_{max}$.

The following values of the experimental parameters are used:

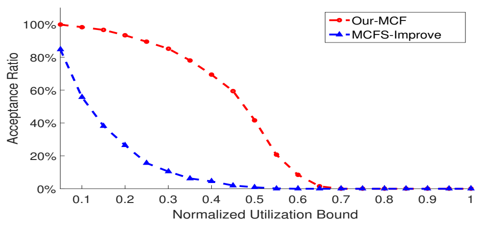
- Number of processors: $M \in \{16, 32, 48, 64, 80, 96, 112, 128, 144, 160\}$.
- Normalized utilization bound: $U_B \in \{0.05, 0.1, \ldots 1.0\}$.
- Probability of a task to be a high-utilization task: $p^{hu} \in \{0.1, 0.2, \ldots 1.0\}$.
- Upper bound on overload utilization of a high-utilization task: $u_{max} \in \{2.0, 4.0, \ldots 16.0\}$.
- The maximum ratio of period and overload critical-path length: $P_{max} \in \{2.0, 2.25 \ldots 4.0\}$.
- Probability of a task to be a `HI`-critical task: $p^{hc} \in \{0.1, 0.2, \ldots 1.0\}$.
- The maximum ratio of overload and nominal utilizations: $R_{max} \in \{2.0, 2.25 \ldots 4.0\}$.

We consider a total of 12,960,000 different combinations of the above parameters to generate the tasksets. For each combination, we generate 1000 parallel `MC` tasksets where each taskset is generated as follows (each parameter is selected from an uniform distribution):

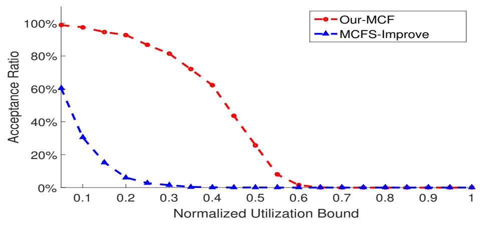
- Task period $D_i = T_i$ is drawn from the range $[10, 1000]$.
- A real number p_i^u is drawn from the range $[0, 1]$. If $p_i^u \leq p^{hu}$, then τ_i is a high-utilization task and its overload utilization u_i^O is drawn in the range $[1.02, u_{max}]$; otherwise, τ_i is a low-utilization task and its overload utilization u_i^O is drawn in the range $[0.02, 1]$. The overload total work of τ_i is $C_i^O = u_i^O \times T_i$.
- A real number P_i is drawn from the range $[1, P_{max}]$ and the overload critical-path length is $L_i^O = T_i/P_i$.
- A real number p_i^c is drawn from the range $[0, 1]$. If $p_i^c \leq p^{hc}$, then $Z_i = \texttt{HI}$; otherwise $Z_i = \texttt{LO}$.
- If $Z_i = \texttt{HI}$, then a real number R_i is drawn from the range $[1, R_{max}]$; otherwise $R_i = 1$.



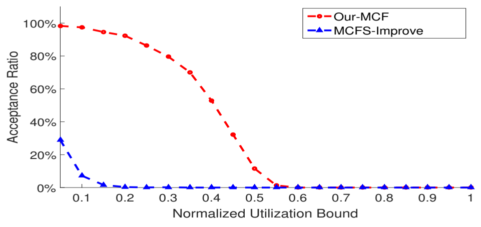
(a) $M = 16$



(b) $M = 32$



(c) $M = 64$



(d) $M = 128$

Figure 1 Comparison of acceptance ratios for different number of processors for $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$.

- The nominal total work and critical-path length are $C_i^N = C_i^O/R_i$ and $L_i^N = L_i^O/R_i$, respectively.
- Repeat the above steps as long as $max\{U^O/m, U^N/m\} \leq U_B$. Once the condition is violated, discard the task that was generated the last.
- If the resulting taskset satisfies the condition $max\{U^O/m, U^N/m\} > U_B - 0.05$, then accept the taskset and stop the procedure. Otherwise, discard the taskset and the repeat the above steps.

The above taskset generation procedure ensures that each taskset has a total normalized utilization within the range $U_B - 0.05$ and U_B. This is reasonable because in our experiments we consider values of U_B that are incremented in step of 0.05.

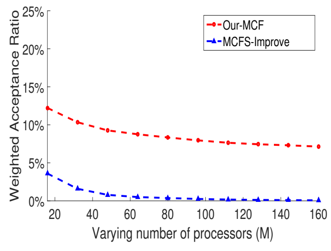
5.2 Results: Schedulability Tests

We compare the effectiveness of `Our-MCFQ` test in terms of guaranteeing the schedulability of randomly generated parallel `MC` tasksets in comparison to the `MCFS-Improve` test in [26].

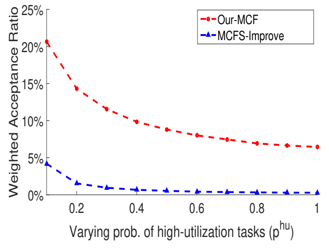
For a given schedulability test and values of M, U_B, p^{hu}, u_{max}, P_{max}, p^{hc} and R_{max}, let the *acceptance ratio* denotes the fraction of tasksets out of 1000 tasksets that are deemed schedulable by the test at normalized utilization bound U_B. The acceptance ratios for $M = 16, 32, 64, 128$ are presented in Figure 1 for $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$ where the x-axis is the normalized utilization bound U_B and the y-axis is the acceptance ratio.

The acceptance ratios of both tests decreases as the normalized utilization bound U_B increases. Such decreasing trend in acceptance ratio for larger U_B is expected because tasksets with a relatively larger utilization are generally difficult to schedule.

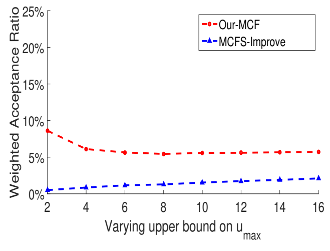
The acceptance ratio of `Our-MCFQ` test is significantly better than the acceptance ratio of `MCFS-Improve` test for $M = 16, 32, 64, 128$. For example, the acceptance ratio in Figure 1b at $U_B = 0.4$ for $M = 32$ is around 70% for `Our-MCFQ` test and less than 10% for `MCFS-Improve` test. For $M = 128$ in Figure 1d, the acceptance ratio at $U_B = 0.2$ is



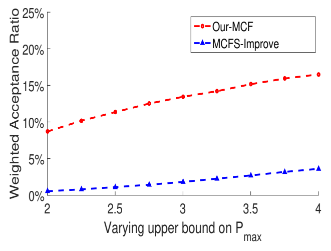
(a) Varying M



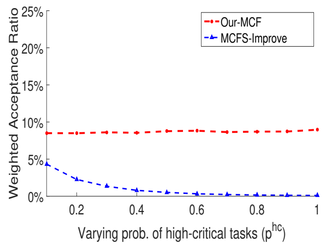
(b) varying p^{hu}



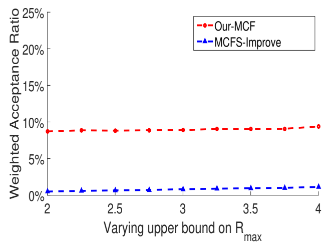
(c) varying u_{max}



(d) Varying P_{max}



(e) Varying p^{hc}



(f) Varying R_{max}

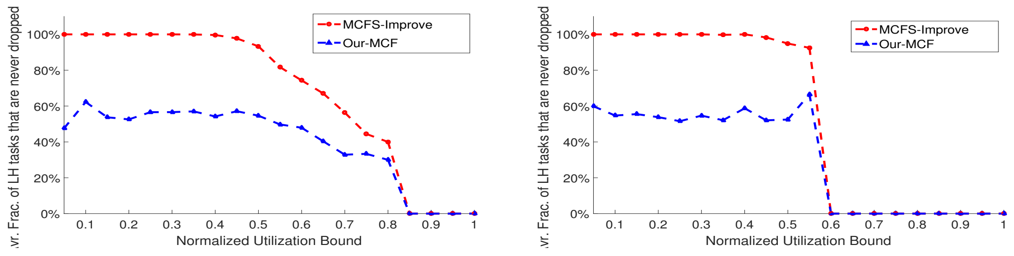
Figure 2 Weighted acceptance ratios for varying values of M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}.

around 90% for `Our-MCFQ` test and 0% for `MCFS-Improve` test. The acceptance ratio of the `MCFS-Improve` test decreases to zero very rapidly with increasing U_B for higher number of processors in comparison to the `Our-MCFQ` test.

The relatively higher acceptance ratio of the `Our-MCFQ` test is due to our proposed task assignment algorithm for the `HH` tasks. The `MCFQ` algorithm determines an assignment of the `HH` tasks to the processors by choosing from different alternatives by taking in to account the number of processors required for other tasks during the typical and critical states. On the other hand, the task assignment of the `MCFS-Improve` test is restrictive in terms of the number of different alternatives for assigning the `HH` tasks to the processors. It can be analytically shown that if we plugin the alternative for assigning processors to the `HH` tasks computed based on the `MCFS-Improve` test into the proposed schedulability test in Eq. (5), then the test in Eq. (5) is also satisfied, which implies that the capacity augmentation bound of the `MCFS-Improve` test also applies to our proposed test. However, such an analysis is omitted in this paper due to space constraint.

The results presented in [26] show quite high acceptance ratio in comparison to the results presented in this paper for the `MCFS-Improve` test. The reason is that we do not use the task set generation algorithm from [26] because some of the assumptions were not explicitly described in [26]. For example, it is not described in [26] how random numbers with log normal distribution with mean $(1 + \sqrt{m}/3)$ was generated without knowing the mean (μ) and standard deviation (σ) of the associated normal distribution.

For comparison of the acceptance ratios of `Our-MCFQ` test and `MCFS-Improve` test for varying values of M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}, we also computed the *weighted acceptance ratios* and presented in Figure 2. The weighted acceptance ratio denotes the fraction of schedulable tasksets weighted by the normalized utilization bound U_B. If $AR(U_B)$ denotes the acceptance ratio of a schedulability test for normalized utilization bound U_B for some given values of M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}, then the weighted acceptance ratio for a set S of U_B values is given as follows: $W(S) = \left(\sum_{U_B \in S}(AR(U_B) \times U_B)\right) / \sum_{U_B \in S} U_B$.



(a) $M = 16$ **(b)** $M = 32$

Figure 3 Average fraction of LH tasks that are never dropped for $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$.

When computing the weighted acceptance ratio by varying one parameter, the other five parameters are kept fixed. The fixed values of the parameters are $M = 64$, $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$ and $R_{max} = 2.0$. The significantly higher weighted acceptance ratio of `Our-MCFQ` test in comparison to `MCFS-Improve` test is evident in Figure 2a-2f respectively for the variation of the parameters M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}. The acceptance ratio of `Our-MCFQ` is much higher because the task assignment algorithm is successful in finding an allocation of the tasks to the processors such that the system is correct while the task assignment of the `MCFS-Improve` test fails in many cases to find such an assignment.

5.3 Results: Quality of Service

In this subsection, we compare the effectiveness of `Our-MCFQ` test with `MCFS-Improve` test in improving the QoS of the system in terms of average fraction of the number of `LH` tasks that are not dropped regardless of the state of the system. Note that `Our-MCFQ` test can significantly schedule more tasksets than the `MCFS-Improve` test (Figure 1). For fairness, we compare the QoS for only those tasksets that are deemed schedulable by both tests.

For each taskset that is deemed schedulable using both the `Our-MCFQ` test and the `MCFS-Improve` test, (i) we apply the ILP in Eq. (15) to determine the fraction of the number of `LH` tasks that are never dropped under the `MCFQ` algorithm, and (ii) we also determine the fraction of the number of `LH` tasks that are never dropped based on the implementation in [26]. The average fraction of the number of `LH` tasks that are never dropped (over all the tasksets that are schedulable by both test) at each normalized utilization bound U_B is computed for each test and presented for $M = 32$ and $M = 64$ in Figure 3 where $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$.

It is evident that `Our-MCFQ` test is able to schedule all the `LH` tasks for normalized utilization $U_B \leq 0.4$ while `MCFS-Improve` is never successful in allocating all the `LH` tasks for any U_B. For $U_B > 0.4$, the `Our-MCFQ` test can also schedule large fraction of the `LH` tasks without ever dropping them in comparison to `MCFS-Improve`. Therefore, the QoS of the system using `Our-MCFQ` test is much higher than that of under `MCFS-Improve`.

6 Related Work

There have been several works on real-time scheduling of parallel non-MC tasks on multiprocessors based on fork-join model [22, 1], synchronous parallel task model [35, 32, 15], and the dag task model [10, 12, 28, 3, 31]. Many of these works proposed resource-augmentation bounds and schedulability tests for global scheduling where the nodes of the tasks are allowed to migrate from one processor to another. There are two other mechanisms to schedule parallel DAG tasks: federated scheduling [25] and decomposition-based scheduling [21]. In decomposition-based scheduling, a DAG task is transferred into a set of independent sporadic task by inserting artificial release time and artificial deadline. The decomposed subtasks of all the DAG tasks are scheduled based on GEDF scheduling policy in [21].

There are many works on scheduling MC systems since the seminal work by Vestal who first proposed the MC sequential task model and its analysis based on fixed-priority scheduling algorithm on uniprocessor platform [38]. Building upon Vestal's seminal work [38], there have been several approaches [9, 16, 11, 8, 24, 19, 4, 17, 23, 7, 5, 34] to design certification-cognizant scheduling of MC system for both uni- and multiprocessor. The work in [14] presents a recent survey on real-time scheduling of MC sequential tasks. To improve the quality of service for the LO-critical tasks, there are also works that consider that the LO-critical tasks are not dropped but provide delayed results, for example, by executing them less frequently after the system switches to the critical state (e.g., weakly hard MC task model [18], elastic MC task model [37, 36, 20]) or provides imprecise results [29, 13, 5, 34].

There are very few works on scheduling MC parallel tasks. Some works considers time-table based scheduling [2] or partitioned MC scheduling based on decomposition strategy [30]. However, such scheduling algorithms are not applicable to DAG tasks for which the internal structure is only known during runtime. The work in [27] and its extension in [26] consider federated scheduling of MC sporadic DAG tasks. The authors in [27, 26] also derived capacity augmentation bound of 3.67 for dual-critical tasks. It is also shown that the schedulability test based on the capacity augmentation bound in [27, 26] does not perform well in comparison to the schedulability test `MCFS-Improve` that is based on actual assignment of the tasks to the processors. However, the task assignment for each HH task in `MCFS-Improve` algorithm is not aware of how the other tasks are assigned to the processors and may fail to assign all the tasks to the processors even if there is another way to successfully assign the tasks. On the other hand, our proposed MCFQ algorithm does not finalize the assignment when analyzing each HH task rather finalize the assignment when analyzing the overall task assignment for all the tasks.

7 Conclusion

This paper presents a new schedulability analysis for federated scheduling of MC sporadic DAG tasks on multiprocessors. The salient feature of this analysis is that different alternatives to allocate each of the HH tasks to the processors during the typical and critical states of the system are considered. The particular alternative to allocate a HH task is selected such that all the tasks can be correctly scheduled on a given number of processors. The MCFQ algorithm also tries to maximize the fraction of the number of LH tasks that are never dropped. Experimental results show that the proposed schedulability test for MCFQ algorithm not only can schedule much larger number of random tasksets but also can improve the QoS of the system significantly in comparison to the state of the art. Investigating the schedulability of MC parallel tasks where more than one high-utilization tasks are scheduled on a set of dedicated processors is an interesting future work.

References

1 P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of ECRTS*, 2013.

2 S. Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *Proc. of RTNS*, 2012.

3 S. Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *Proc. of ECRTS*, 2014.

4 S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *Proc. of ECRTS*, 2012. doi:10.1109/ECRTS.2012.42.

5 S. Baruah, A. Burns, and Z. Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *Proc. of ECRTS*, 2016. doi:10.1109/ECRTS.2016.12.

6 S. Baruah, B. Chattopadhyay, H. Li, , and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Syst.*, 50(1):142–177, 2014. doi:10.1007/s11241-013-9184-2.

7 S. Baruah, A. Eswaran, and Z. Guo. MC-Fluid: Simplified and Optimally Quantified. In *Proc. of RTSS*, 2015. doi:10.1109/RTSS.2015.38.

8 S. Baruah, Haohan Li, and L. Stougie. Towards the Design of Certifiable Mixed-criticality Systems. In *Proc. of RTAS*, 2010. doi:10.1109/RTAS.2010.10.

9 S. Baruah and S. Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proc. of ECRTS*, pages 147–155, 2008.

10 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proc. of RTSS*, 2012.

11 Sanjoy Baruah, Alan Burns, and Robert Davis. Response-time analysis for mixed criticality systems. In *Proc. of RTSS*, 2011.

12 V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *Proc. of ECRTS*, 2013.

13 A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Proc. of WMC, RTSS*, 2013.

14 A. Burns and R. Davis. Mixed-criticality systems: A review. In *(available online), Tenth Edition*, January, 2018. URL: http://www-users.cs.york.ac.uk/~burns/review.pdf.

15 Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, A. Easwaran, and Insik Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *Proc. of ECRTS*, 2013.

16 François Dorin, Pascal Richard, Michaël Richard, and Joël Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46:305–331, 2010.

17 P. Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proc. of the ECRTS*, 2012.

18 Oliver Gettings, Sophie Quinton, and Robert I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proc. of RTNS*, 2015. doi:10.1145/2834848.2834850.

19 Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. of RTSS*, 2011. doi:10.1109/RTSS.2011.10.

20 Mathieu Jan, Lilia Zaourar, and Maurice Pitel. Maximizing the execution rate of low-criticality tasks in mixed criticality systems. In *Proc. of WMC, RTSS*, 2013. URL: http://www-users.cs.york.ac.uk/~robdavis/wmc2013/paper6.pdf.

21 X. Jiang, X. Long, N. Guan, and H. Wan. On the decomposition-based global edf scheduling of parallel real-time tasks. In *Proc. of RTSS*, 2016.

22 K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. of RTSS*, 2010.

23 J. Lee, K. M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee. MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors. In *Proc. of RTSS*, 2014. doi:10.1109/RTSS.2014.32.

24 Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proc. of RTSS*, pages 183–192, 2010.

25 J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Proc. of ECRTS*, 2014.

26 J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5):760–811, Sep 2017. doi:10.1007/s11241-017-9281-8.

27 J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu. Mixed-criticality federated scheduling for parallel real-time tasks. In *Proc. of RTAS*, April 2016. doi:10.1109/RTAS.2016.7461340.

28 Jing Li, K. Agrawal, Chenyang Lu, and C. Gill. Analysis of global edf for parallel tasks. In *Proc. of ECRTS*, 2013.

29 Di Liu, Jelena Spasic, Gang Chen, Nan Guan, Songran Liu, Todor Stefanov, and Wang Yi. EDF-VD Scheduling of Mixed-Criticality Systems with Degraded Quality Guarantees. In *Proc. of RTSS*, 2016. doi:10.1109/RTSS.2016.013.

30 Guangdong Liu, Ying Lu, Shige Wang, and Zonghua Gu. Partitioned multiprocessor scheduling of mixed-criticality parallel jobs. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2014. doi:10.1109/RTCSA.2014.6910497.

31 A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G.C. Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *Proc. of ECRTS*, 2015.

32 G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proc. of ECRTS*, pages 321–330, July 2012. doi:10.1109/ECRTS.2012.37.

33 OpenMP. Openmp application program interface. version 4.0. 2013.

34 R. Pathan. Improving the quality-of-service for scheduling mixed-criticality systems on multiprocessors. In *Proc. of ECRTS*, 2017.

35 A. Saifullah, K. Agrawal, Chenyang Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proc. of RTSS*, 2011.

36 H. Su, N. Guan, and D. Zhu. Service guarantee exploration for mixed-criticality systems. In *Proc. of RTCSA*, 2014. doi:10.1109/RTCSA.2014.6910499.

37 H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proc. of DATE*, 2013. doi:10.7873/DATE.2013.043.

38 S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proc. of RTSS*, pages 239–243, 2007. doi:10.1109/RTSS.2007.47.

Virtual Timing Isolation for Mixed-Criticality Systems

Johannes Freitag
Airbus, Munich, Germany
johannes.freitag@airbus.com

Sascha Uhrig
Airbus, Munich, Germany
sascha.uhrig@airbus.com

Theo Ungerer
University of Augsburg, Augsburg, Germany
theo.ungerer@informatik.uni-augsburg.de

Abstract

Commercial of the shelf multicore processors suffer from timing interferences between cores which complicates applying them in hard real-time systems like avionic applications. This paper proposes a virtual timing isolation of one main application running on one core from all other cores. The proposed technique is based on hardware external to the multicore processor and completely transparent to the main application i.e., no modifications of the software including the operating system are necessary. The basic idea is to apply a single-core execution based Worst Case Execution Time analysis and to accept a predefined slowdown during multicore execution. If the slowdown exceeds the acceptable bounds, interferences will be reduced by controlling the behavior of low-critical cores to keep the main application's progress inside the given bounds. Apart from the main goal of isolating the timing of the critical application a subgoal is also to efficiently use the other cores. For that purpose, three different mechanisms for controlling the non-critical cores are compared regarding efficient usage of the complete processor.

Measuring the progress of the main application is performed by tracking the application's Fingerprint. This technology quantifies online any slowdown of execution compared to a given baseline (single-core execution). Several countermeasures to compensate unacceptable slowdowns are proposed and evaluated in this paper, together with an accuracy evaluation of the Fingerprinting. Our evaluations using the TACLeBench benchmark suite show that we can meet a given acceptable timing bound of 4 percent slowdown with a resulting real slowdown of only 3.27 percent in case of a pulse width modulated control and of 4.44 percent in the case of a frequency scaling control.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Computer systems organization → Reliability

Keywords and phrases multicore, hard real-time systems, timing isolation, safety-critical systems, mixed-criticality design and assurance

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.13

Funding This work was partially supported by the German Federal Ministry of Education and Research within the project ARAMiS II with the funding ID 01IS16025Q and the ARTEMIS Joint Undertaking under grant agreement 621429 (EMC2).

1 Introduction

Several companies are seeking a new generation of autonomously piloted aircrafts for future mobility concepts. Vehicles like *Vahana*, *Pop-up*, *CityAirbus* [4], or Lilium Jet [18] will be ultra light-weight electrical helicopter-style vehicles providing a novel autonomous urban transportation concept. The avionic systems for this kind of aircraft need to implement most functionality available in current aircrafts while providing additional complex functionality for autonomous flying. Furthermore, the electronic systems must be optimized for weight and space in order to fit into this new generation of aircrafts.

A possible solution that enables the necessary integration of multiple avionic applications into less avionic computers is the use of (massive) multicore processors comprising eight or even more cores. Avionic systems show special requirements with respect to system reliability and availability because of their safety-critical nature.

Even though first ideas of the regulations on how to apply multicore systems in avionics are presented in the CAST-32 position paper and its follow-up CAST-32a [7], both authored from the Certification Authorities Software Team (CAST), concrete design details are still open. One of the major challenges in this context is the interference between applications since theoretically one application can compromise another one, at least in the timing domain. Accordingly, an essential requirement for certification is a clear and reliable isolation of safety-critical applications that needs to be demonstrated to the certification authorities.

One of the most important issues is the contention on the memory (sub-)system resulting from different applications on the cores since it has a major impact on the actual execution time of an application. This is based not only on queued accesses to the memory and interconnection systems but also on contention on shared caches.

For multicore systems, an approach to support execution of highly critical avionic (legacy) applications is the *Fingerprinting* technology presented in [11]. *Fingerprinting* continuously tracks the progress of an application by comparing the current state of execution to a virtual single-core execution of the same application. Unacceptable timing deviations caused by inter-core interferences can be mitigated by controlling the behavior of the non-critical cores. Furthermore, the approach used for slowing down the cores shall allow the most efficient possible usage of the other cores.

The contributions of this paper are

- an evaluation of the *Fingerprinting*'s accuracy,
- an analysis of the *Fingerprinting*'s (non-)intrusiveness on the main application,
- three possible approaches to influence the behavior of the low priority cores for interference reduction of the critical core,
- a complete external closed control loop (CCL) that guarantees virtual timing isolation between one main application and any other application running on a multicore system.

The remainder of this paper is organized as follows. The environment in which the approach applies as well as the relevant hardware configurations are presented in Section 2. Section 3 provides an overview of mature techniques and related work. The fingerprint technology is described in Section 4 while the actuators are presented in Section 5. Section 6 introduces the complete control loop. Sections 4 to 6 comprise individual evaluations. The paper concludes with Section 7 including an outlook on future work.

2 Setting the Scene

The avionic domain is a very defensive domain regarding novel technologies, mainly caused by possible safety issues. Hence, we focus on the use of multicores with only a single-core executing highly (safety) critical application (referred to as *main* application in the following) while the others run applications with lower criticality. With respect to the timing requirements examined in this paper, this means the first core is executing applications with hard deadlines which must never be missed while the other cores run weakly hard [6], soft, or non real-time applications. Accordingly, we are proposing a technique that enables performance and timing guarantees for one core on the cost of the other cores' performance.

In this approach we focus on critical applications that are executed periodically which is typically the case for avionic applications. An example is an application which in every loop reads data as input, processes the data and creates an output while the complete procedure happens in a cycle of 5ms to 100ms. Any type of algorithm can be computed and the execution of different code depending on the input is possible in every loop. A lightweight operating system can schedule multiple applications with fixed time slicing (e.g. as in integrated modular avionics (IMA)). However, the critical application shall not exchange data with the low priority applications. The aforementioned restrictions do not apply for the low non-critical applications running on the other cores. However, no timing guaranties can be provided for these applications and it must be possible to change the timing behavior arbitrarily without crashing neither the high-critical nor the low-critical application.

In order to reuse legacy software, guaranteeing a required performance shall be non-intrusive. Moreover, modifications to an operating system (if any) shall be restricted to a minimum to not increase system complexity too much. As appropriate standards and best-practices propose extra circuits external to the processor system to increase system reliability and safety (e.g. mentioned in [7]), we target at using such an external device for guaranteeing performance and timing. In the optimal case, this timing isolation shall be done in addition to the original tasks of a watchdog system.

2.1 Basic Idea

Our virtual timing isolation approach tracks the main application's progress on the basis of characterized behavior of hardware event counters integrated inside the core of a multicore. Examples for suitable events are the number of executed instructions, cache misses, and executed floating point operations based on a given time period. Periodically reading and resetting such counters results in a curve that is characteristic for an executed application, more specifically, for the progress of that application. When comparing a recorded reference curve to the performance counter values measured online, the current progress with respect to the reference execution can be measured.

In case the performance drops down, our timing isolation system is able to thwart the other cores to reduce interferences. An integrated closed loop controller is responsible for this task.

The hardware setup consisting of a main multicore processor and the timing isolation system is shown in Figure 1. In our demonstrator system, a Xilinx FPGA is used for the timing isolation system and implements all functionality required for measuring and influencing the performance of the main application running on one core of the multicore. The two systems are connected by the trace channels of the multicore (high speed serial link) which need to be a bidirectional connection. The available Aurora interface fulfills this requirement and provides access to the internal debug unit. Hence, the FPGA can read the performance counters via the debug unit and any action for controlling the cores can also be performed by this debug unit.

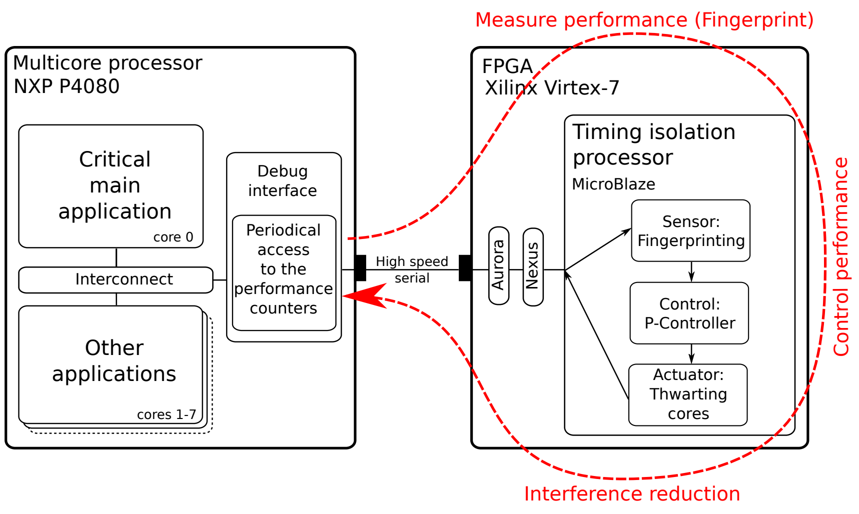


Figure 1 Hardware setup with a multicore processor under observation by the timing isolation system implemented in an FPGA.

Table 1 Different cache configurations used in the evaluations.

Realistic	
L1	on
L2	off
L3	off
Memory	ext. SDRAM

Max. Interferences	
L1	off
L2	off
L3	off
Memory	ext. SDRAM

Max. Traffic	
L1	off
L2	off
L3	int. SRAM
Memory	int. SRAM

2.2 Hardware Configuration

The system under observation is a NXP P4080 which is an eight-core multicore processor based on the PowerPC architecture (see Figure 1). All cores are configured to run at a nominal frequency of 1.5 GHz. The processor comprises three caching levels where the L1 (separated instruction and data) and L2 (shared instruction and data) caches are private to each core while the 2 MB L3 cache is shared between all cores [23]. Furthermore, the processor provides two memory controllers from which only one is used for the evaluations. Cache coherency as well as cache stashing is disabled for isolation of the cores.

For our evaluations we used different configurations of the caches (see Table 1) to demonstrate the different technologies under appropriate conditions. In all configurations the private L2 cache is disabled because enabling it increases core-local caches and reduces interferences between cores (due to lower miss-rates). Moreover, L3 is never used as shared cache since this would complicate a possible WCET analysis. The *Realistic* configuration uses the local L1 instruction and data caches and the external SDRAM as main memory. *Max. Interference* also disables both L1 caches to generate the maximum accesses from the cores to the interconnect. Since all accesses target the external SDRAM, they show a comparatively long latency. The *Max. Traffic* configuration is similar to *Max. Interference* but uses an internal SRAM (L3 used as SRAM) instead of the external SDRAM. This configuration generates the highest traffic on the interconnect due to the low latencies of accesses.

The timing isolation processor is implemented as a soft-core micro-controller (Xilinx MicroBlaze) running at 125 MHz inside a Xilinx Virtex-7 FPGA (see Figure 1). The FPGA is attached to the debug unit of the P4080 via a high speed serial *Aurora* link (2.5 GBit/s).

Thus, the FPGA is able to extract information from the multicore processor without stopping the cores, trigger frequency scaling and halt and continue cores. Since messages from and to the debug interface are wrapped into the NEXUS protocol [1], we developed a Nexus IP block to speed up the process of extracting data in the FPGA. The software for monitoring and controlling the main applications progress is executed by the MicroBlaze.

2.3 Benchmarks

For the evaluation of the approach two different benchmarks are used to demonstrate the effectiveness in a worst case interference scenario (read/write opponent) and a more realistic scenario (TACLeBench benchmark suite).

2.3.1 Read/Write Opponent

In order to create a worst case interference scenario benchmark, as much data has to be stored/loaded to and from memory as possible. In the presence of caches, every access to the data should be a cache miss in the private caches in order to create interferences. Therefore, the distance between two consecutive memory accesses has to be at least the size of a cache line. In the case of the P4080 this is 64 Bytes in the L1 and L2. This technique is described in more detail in [21]. For the evaluation, an assembler program was developed that consists of a loop that only executes either load or store instructions with a distance of 64 Bytes. Thus, the code fits into the instruction cache but not in the data cache.

2.3.2 TACLeBench

TACLeBench [10] is a benchmark suite comprising five packages of algorithms which are commonly used in embedded systems. In this paper only 19 algorithms from the TACLeBench (version 1.9) sequential package are used because these algorithms do not fit completely in the private caches like the other benchmark packages. Examples of the sequential algorithms are encrypting, sorting, dijkstra, H.264 block decoding and image recognition. These programs can be compiled independent of standard libraries and operating systems which makes them easy to adapt to our test system. The code size of the individual algorithms ranges from 117 to 2710 SLOCs.

Similar to a real avionic application we executed the 19 benchmark algorithms successively in a loop. In order to simulate a program that acts different for different input parameters the order of the algorithms can be defined to be random for each run.

3 Related Work

Multicore systems in avionic applications are still not wide spread. One reason is the difficulty to obtain suitable Worst Case Execution Time (WCET) estimates since application performance can drop significantly if multiple cores (i.e. applications) are sharing bus and memory [20]. Furthermore, it is not possible to identify all interference channels on COTS multicore processors [2]. Therefore, a WCET analysis on possible worst case scenarios leads to a high WCET overestimation (estimated WCET compared to unknown realistic WCET) for current COTS MPSoCs. Hence, the performance gain of the multicore is neglected.

Predictability by processor design is studied for example in [25], [15], and [27]. Furthermore, there exist several approaches to limit or even control the interferences between high and low critical tasks on multicore systems in software to relax the worst case scenario and, hence, improve WCET analysis results. These solutions focus on task or even thread granularity and are integrated into the scheduling of the system. The main idea of these approaches is counting e.g. bus accesses and limiting them by suspending the corresponding thread. Examples of such approaches are presented in [17], [21], [5], [26], [16], and [3]. An overview of these and other approaches is given in [14].

Even though these approaches are very interesting for newly developed applications, they are not suitable for combing multiple legacy single-core avionic applications on a multicore processor because the legacy applications or the underlying operating system would either have to be modified, which leads to a high effort in certification (because of increased system complexity), or restrict the applications in a way that the performance gain of the multicore is neglected.

A previous approach for characterizing an application's execution is presented in [9]. It is used in high performance systems to predict an application's future behavior and needs for adjusting architectural parameters for performance optimizations. It is not related to embedded real-time systems but successfully uses a similar, but intrusive, technology for tracking application's performance.

The use of feedback controllers in combination with real-time systems is not new. For example, a closed loop controller is used in [19] for dynamic resource allocation and power optimization of multicore processors. An example for closed loop control in a real-time scheduler is presented in [24] and [8] while a controller for thermal control of a multicore processor is introduced in [13]. However, all of these methods require intrusive measurements and no non-intrusive approach for controlling the interferences between cores by an external device has been presented in the past.

4 Progress Measurement using Fingerprints

Implementing a closed control loop requires a controlled system, a sensor, an actuator, and a control algorithm. The controlled system is the main application running on one core. As sensor element, we developed a *Fingerprinting* system that tracks the progress of an application transparently. The *Fingerprinting* is described in this section while the actuator and control elements are presented in Sections 5 and 6, respectively.

4.1 Fingerprinting

During the execution of an application a flow of instructions is executed. This flow is not homogeneous in terms of type of instructions (e.g. arithmetic, floating point, branch), source of the instructions (e.g. cache, internal scratchpad, external memory), and execution time of instructions (e.g. simple arithmetic, complex arithmetic, memory access). Accordingly, measuring for example the number of executed floating point instructions per time unit will lead to a characteristic curve of an application or a part of the application. If the application (or the relevant part of it) is executed several times the measured curves are very similar. For tracking the progress of a known application, its measured curve can be compared to the recorded reference curve of executed floating point instructions at any time.

In case an application executed on a multicore processor suffers from interferences with other applications on the shared memory hierarchy, its progress is slowed down. Slowing down the application will result in a stretched (in time) but shrunk (in the value range) curve. When comparing such a mutated measured curve with the original reference curve, the actual slowdown can not only be identified but also be quantified at any time during execution.

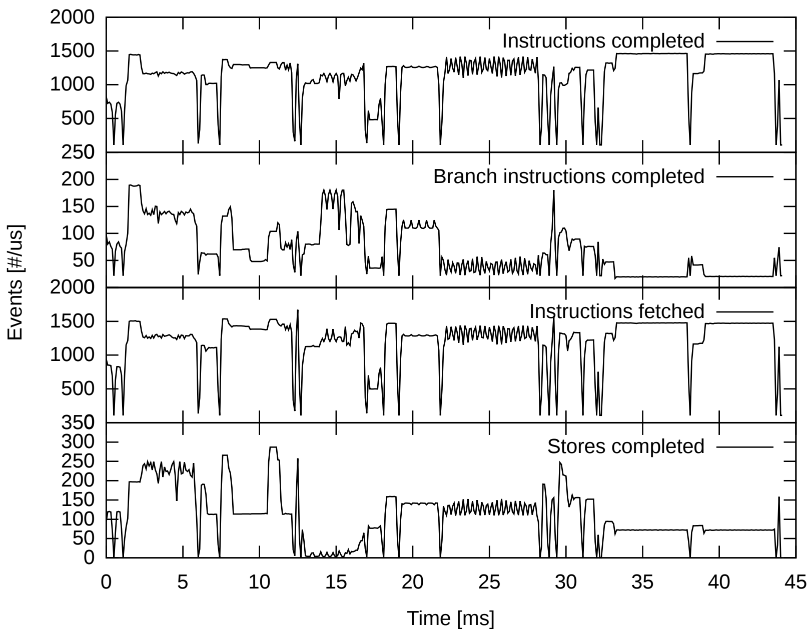


Figure 2 Measured curves of the four event counters: *Instructions completed* on top curve, *Instructions fetched* second, *Branches completed* third, and *Stores completed* in lowest curve when executing the sequential benchmarks of the TACLeBench benchmark suite.

Many current MPSoC (e.g. based on ARM, PowerPC) include performance counters implemented in hardware which can be configured to increment every time a given event is raised. While the amount of events which can be configured is usually more than 100, the amount of counters that can be incremented simultaneously is small (around four to six) [22]. Therefore, the events that are suitable for tracking have to be selected.

Figure 2 presents event counter curves of the TACLeBench *sequential* benchmarks (see Section 2.3.2) for the four event types *Instructions completed*, *Instructions fetched*, *Branches completed*, and *Stores completed*. These event types were used throughout the whole paper. For the fingerprints it is not relevant which event types are selected as long as it produces a continues stream of measurement data (which is for example not the case for floating point instructions) and the selected event types result in different curves. The displayed curves originate from a bare metal execution on a NXP P4080. The characteristics origin from the following algorithms within the TACLeBench in the following order: adpcm, anagram, audiobeam, cjpeg_transupp, cjpeg_wrbmp, epic, fmref, g723, gsm, h264, huff, ndes, petrinet, rijndael, statemate. These algorithms for example include jpeg transformations (7th to 12th ms), h264 decodings (21th to 28th ms) and AES decryption (32rd to 38th ms). In the figure it is visible that the characteristics are different for the type of algorithms executed as well as the monitored events for the same algorithm.

4.2 Creation of a Fingerprint Model

The *Fingerprint* model is obtained by executing the main application several (thousand) times. The performance counter values of the selected events are recorded with the frequency defined by the algorithm running on the timing isolation processor (100μs period in this case which is identical to the tracking frequency). Afterward, the recorded characteristics

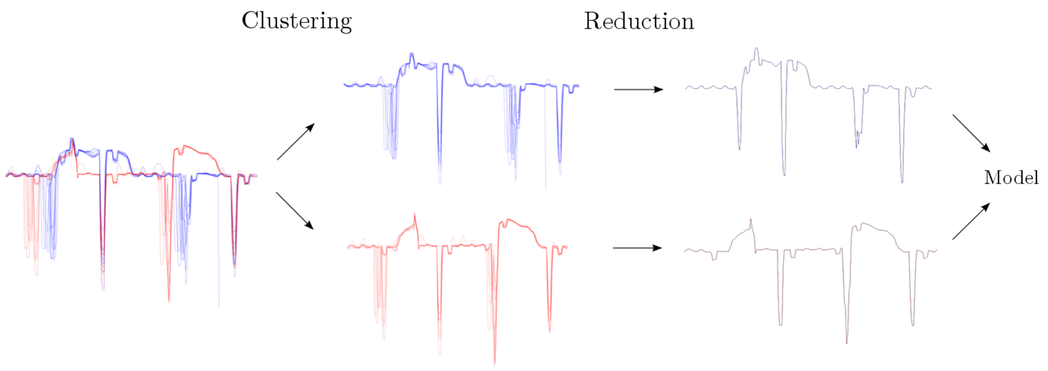


Figure 3 Generation process of the Fingerprint model. The raw data (left) is clustered by a k-means algorithm and is reduced to the median curve to build up the Fingerprint model.

are clustered using a *bisecting k-means* algorithm applying the distance function[1]

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} [|x_i - y_i| > \delta_{max}] \tag{1}$$

with runtime measurement vector $\mathbf{x}$, centroid vector $\mathbf{y}$, length n of the pattern and the maximum difference between two data points δ_{max}. This distance function does not sum up the differences between each measurement point but it counts the number of samples with an error higher than the given δ_{max}. In comparison to the standard distance function this function is not sensitive to drops in the curve as displayed in Figure 3 on the left. In case of two curves where these drops are slightly shifted, the overall distance in the standard function would be big even though the rest of the curve fits perfectly. With the given distance function, errors bigger than δ_{max} are taken equally into account which better clusters the main streams within the recorded data.

The *bisecting k-means* algorithm was chosen because no predefined number of clusters has to be given. Instead, the number of clusters is resulting from a defined maximum distance d_{max}. Thus, only fingerprints with a distance $d \leq d_{max}$ to each other are in one cluster. The centroid is defined randomly for the first iteration of the *bisecting k-means* algorithm and refined in the subsequent iterations until the clusters reach their final states. The medians of the resulting cluster centroids are combined into a tree model, the *Fingerprint*. Figure 3 shows an example process flow of the Fingerprint generation. In the first step an overlay of the recorded curves is displayed which are clustered in the second step. Afterwards, the medians of the clusters are computed and the original curves are discarded. Finally, the tree model is created.

The *Fingerprint* is represented by a graph as shown in Figure 4. The root node is the starting point of the application[2]. A node describes a sequence of counter values characterizing a part of the applications execution. One node consists of at least the number of simultaneously compared samples during tracking (4 samples in the current implementation) and depending on the application, all the samples until to the end of the period of the

[1] Please note the Iverson brackets: $[P] = \begin{cases} 1 & \text{if } P \text{ is true;} \\ 0 & \text{otherwise.} \end{cases}$

[2] Note that the application must be a typical embedded application that is executed periodically.

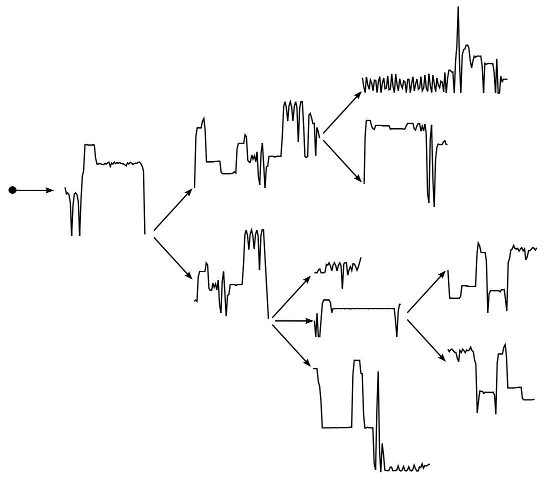


Figure 4 Example of a *Fingerprint* tree model.

application or the next branch. Each node has at least one successor node, except for the last node representing the end of execution in one iteration. In case of multiple successor nodes they represent (at least) two different paths of following characteristics. The split into multiple paths can be caused by different execution paths of the application or by different environmental conditions e.g., the first execution can have a different path than the following executions due to cold caches and warmed-up caches. It can also happen that different execution paths are not explicitly visible in the *Fingerprint* if these paths show similar characteristics as others.

4.3 Tracking the Progress of Applications in Real-Time

During actual execution time, the timing isolation system again reads the performance counter values. In this case, the values are compared to the stored *Fingerprint* model and the actual path along the tree is tracked. In contrast to the generation of the *Fingerprint* model which can be created off-line on a powerful compute node, timing is crucial for the tracking phase. The comparison between the measurements and the stored characteristic sequences has to be performed in real-time. Furthermore, it has to be done with the limited performance of a micro-controller as it would economically not make sense to observe the multicore with a high power single-core processor or even a multicore processor. In our setup the tracking is performed by a Microblaze which is a soft core micro-controller. For performance reasons the similarity is determined with a simple distance function

$$d(t) = \sum_{i=t-n}^{t} (x_i - F_i)^2 \tag{2}$$

with $\mathbf{x}$ runtime measurement vector, $\mathbf{F}$ Fingerprint vector, t discrete sampling time step from the beginning of the current period and n number of samples to compare. In the current implementation, in every iteration the recent four samples are compared. Comparison of only four samples is sufficient because the resulting similarity is accumulated over time. Thus, the decision whether the complete application run fits to the *Fingerprint* is not only based on the most recent comparison but on all the measurements.

Since measurements and *Fingerprint* never match exactly because of different execution environments (cache state, concurrent bus and memory accesses) or just because of jitter at

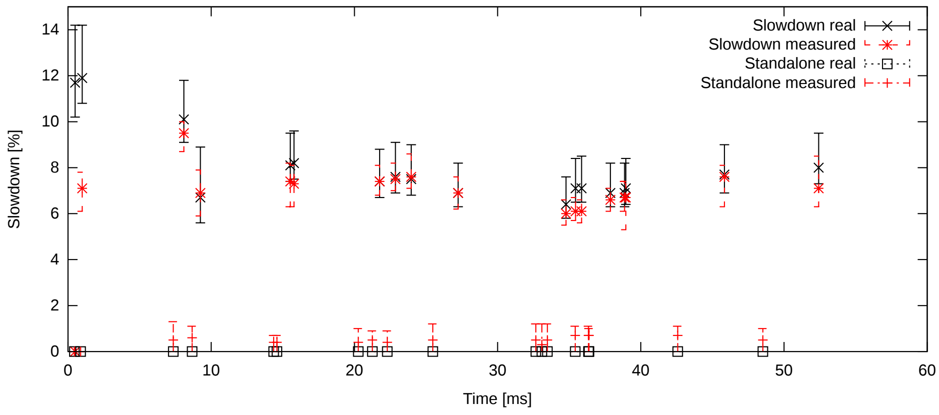


Figure 5 Quality of the quantification over the runtime of the TACLeBench suite in standalone and with seven opponent cores (*Slowdown*) using the *Realistic* cache configuration (L1 cache is enabled). The plotted dots represent the mean values while the bars reflect the minimum and maximum value measured.

the measuring points, the tracking algorithm is based on similarities, not on exact equality. This is of special importance at the nodes of the *Fingerprint* model because here the algorithm has to decide which path to continue.

Because the selection of the future path is based on similarity, there is an uncertainty at the decision point. To make the tracking robust despite of this uncertainty, our *Fingerprinting* implementation is able to track multiple different paths in parallel. In case of further branches in the tree, the most probable paths are followed and less probable paths are dropped. This decision is based on the matching of the runtime values with the path until the decision point is reached. In the current implementation four paths can be tacked simultaneously.

When determining the similarity of an actual measurement sequence to the model, the actual values are compared to the original model. Furthermore, a slowed down version of the model is computed by shifting the original model which is also compared to the measured values. In case of a better fit of the slowed down version the delay of the actual execution is determined. For the following comparisons the complete model is shifted in order to fit to the measurements. This enables the algorithm to track the application also in case of a delayed execution, e.g. by bus and/or memory contention. A slowdown can not only be detected, it can also be quantified during the execution.

4.4 Precision of the Interference Quantification over Time

For the evaluation of the precision of the quantification algorithm we instrumented the TACLeBench suite. The instrumentation is inserted at the start and after every benchmark. Thus, 20 milestones are inserted into the 19 algorithms of the TACLeBench suite subsequently. Each instrumentation consists of a time measurement within the multicore processor, which is stored in the RAM of the P4080 for later readout, and a special trace message which is sent to the quantification FPGA. Therefore, the time measurements can be used to calculate the actual slowdown which can be compared to the interference quantification values at the time these messages are received.

For the comparison, the TACLeBench was first executed standalone in order to record the time measurements without slowdown as shown in Figure 5 “Standalone real”. All the different measurements are performed 100 times and the mean values are plotted. The bars are indicating the minimum and maximum values. The cache configuration in this experiment is *Realistic* (L1 cache is enabled). At the same time the slowdown was measured by the FPGA displayed in the "Standalone measured" curve. Here it is visible that there is a slight overestimation in the interference quantification of around 1% in some cases.

In the second step, the benchmark was executed with seven *write* opponents causing an average total slowdown of around 8%. However, the slowdown over time varies as it is depending on the different algorithms and their memory access behavior. The actual slowdown is shown in Figure 5 "Slowdown real" while the slowdown detected by the interference quantification with fingerprints is labeled as "Slowdown measured". Overall, the real and the measured slowdown are matching very well. For example the final (at 52ms) actual average slowdown value is 8.0% compared to a measured slowdown of 7.1%. In the case of the "Standalone" execution, the final average overestimation of the slowdown is only 0.5%. In total, the average deviation of the average value of less than 1%. However, in the beginning of the run (first millisecond) the values do not match. This is due to the fact that the quantification algorithm takes a small start period of around 1 ms to align the measured curve with the curves in the model. However, once this alignment is fixed the matching is very responsive as can be seen in the figure. An evaluation on which slowdowns can be reliably detected and how fast these can be detected is presented in [11].

4.5 Non-intrusiveness of the Read-Out Process

In order not to create any further interference on the critical application, the read-out overhead of the progress tracking should be as small as possible (non-intrusive). However, as the *Fingerprinting* approach relies on the performance counter values (see Section 4.1) which reside inside the cores these values have to be accessed from outside the SoC. The extraction process including the possible interference channels are displayed in Figure 6. Every extraction is triggered by an external signal sent by the external timing isolation system.

Once an external signal is received, the first step (1 in Figure 6) inside the SoC is the transfer of the performance counter values to the memory-mapped *Performance Monitor Counter Capture Registers* as the performance counter registers inside the cores are not accessible by the debug interface. This transfer is triggered by a signal from the debug interface to the respective core. In the manual of the e500mc cores [22] it is not specified where the *Performance Monitor Counter Capture Registers* are located and how the transfer is implemented. However, measurements showed that this happens in a non intrusive way as no delay of a program executed on the core could be observed.

In the second step (2 in Figure 6) the *Performance Monitor Counter Capture Registers* are accessed by the *Triggered memory-mapped access* unit of the debug interface. This is implemented as a usual memory mapped access. Therefore, the interconnect is used to transport the data to the debug interface. This is a possible interference channel as the interconnect is also used by the cores when these access the memory, the shared cache or the I/O interfaces.

For a reliable tracking four 32bit performance counter values per core need to be extracted as mentioned in Section 4.1. Depending on the read out frequency the bandwidth needed on the interconnect varies. For an example extraction frequency of 1 MHz (1 μs period) a bandwidth of 128 Mbit/s per observed core is needed. However, if the performance counters of all cores shall be extracted in parallel at this frequency the resulting bandwidth is 1 Gbit/s. Even though NXP claims that the P4080 provides 0.8 Tbps coherent read bandwidth [23], interference is measurable even if only one core is observed.

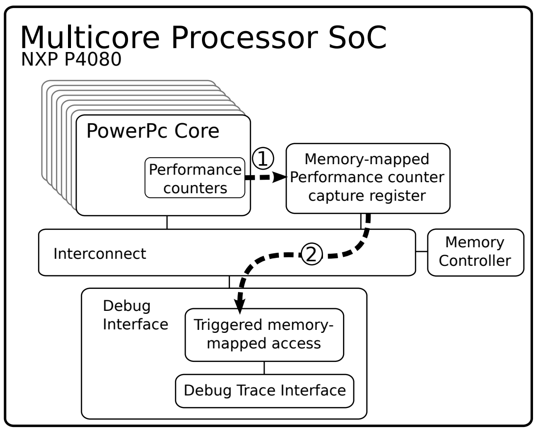


Figure 6 Possible interference channels on the example of an NXP P4080 SoC with eight cores.

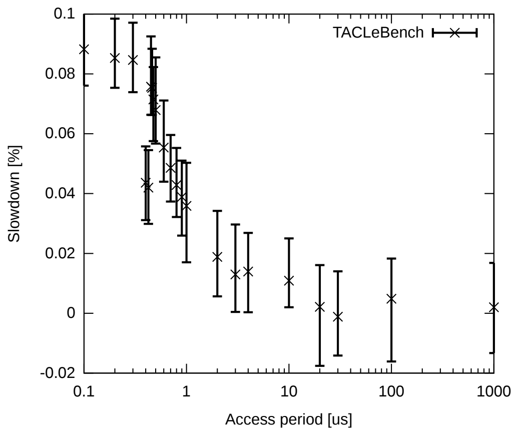


Figure 7 Slowdown of the TACLeBench execution on one core depending on the access period of the performance counter read out process.

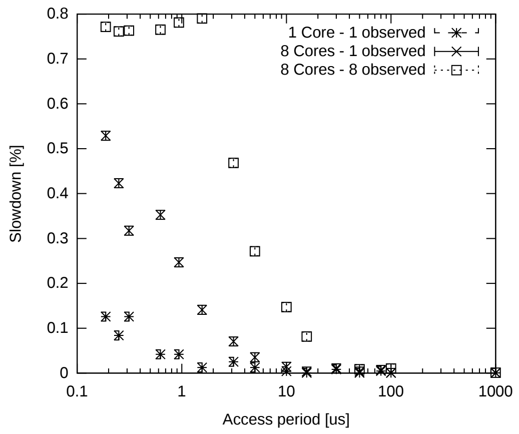


Figure 8 Slowdown of the read benchmark to the on-chip SRAM (L3 cache configured as SRAM) while the L1 and L2 caches are disabled.

The interference measured for the execution of the sequential TACLeBench benchmarks on one core while the remaining cores are idle is shown in Figure 7. The slowdown s is defined as

$$s(p) = \left(\frac{x(p)}{x_{unobserved}} - 1\right) * 100 \qquad (3)$$

with the access period p, the execution time without observation $x_{unobserved}$ and the execution time for a given period for reading out the PMC capture registers $x(p)$.

The bars are the respective observed min/max values. At some points the bars are below zero. This results from the fact that the execution time is varying even without disturbance from the read out process. These slight variations are a result of cache, interconnect and memory mechanisms.

It can be recognized that the slowdown is very small (around 0.09%) even at access frequencies of 10 Mhz. For access periods larger than 20 μs the interference is not measurable. The result is identical for the case that the performance values are extracted from the core executing the benchmark as well as from any core that is in idle mode. Even though the interference is decreasing with a higher access period there are two measurements that are lower than expected (around 0.4 μs). We assume this is because of a synchronization of the memory accesses of the TACLeBench with the memory mapped accesses of the debug interface.

The intrusiveness analysis using the TACLeBench benchmarks shows the potential impact on a real application. However, in order to determine the worst case interference of the read out process an application was developed that almost only performs memory accesses. Furthermore, the subsequent memory accesses read/write data from/to addresses with 64 byte distance which is the size of one cache line. Additionally, the L1 and L2 caches are disabled while the L3 cache is used as SRAM memory (*Maximum Traffic* cache configuration). Therefore, every load instruction initiates a transaction in the interconnect which is considered as the worst case.

The slowdown of this application is displayed in Figure 8 for three configurations. In the first configuration the application runs on one core while the remaining cores are idle. In the second and third configuration the application is executed on all eight cores simultaneously. The extraction process is performed on one core or all the cores. The slowdown is determined similar to the TACLeBench analysis with Equation 3 but normalized to the eight core execution without reading the counter registers. Thus, the slowdown resulting from the inter-core interference is eliminated.

The measured slowdown is not significantly higher compared to the TACLeBench analysis for the one core execution. However, when all the eight cores are used for execution, the slowdown is around six times higher in case only one core is observed which is still a very low slowdown. The higher interference for the eight core execution results from the utilization of the interconnect from the cores. For access periods larger than 20 μs (50 kHz) the interference is again not measurable. In case all the cores are observed simultaneously the interference is much higher. As expected, the interference is around eight times higher compared to the case where only one core was monitored. The slowdown reaches a maximum at around 1.02 μs and the slowdown is not increasing with decreasing access periods. At this point the maximum speed of the triggered memory mapped access of the debug interface is reached. However, for extraction periods above 50 μs the interference is also not measurable.

5 Performance Control by Interference Reduction

The presented *Fingerprinting* is used as the sensor element of the closed control loop. In this section the *actuator* which influences the performance of the other cores and, hence, the interferences is explained. A simple technique for reducing the interferences with the main application is halting and resuming the opponent cores based on a threshold. For example, whenever the slowdown of the main application rises over 5% the other cores are halted and continued once the slowdown drops under 5%. This actuator is effective and stops the interferences of the opponent cores but it is heavily intrusive for the tasks running on the other cores and might have severe effects depending on the executed application. Apart from the main goal of isolating the timing of the critical application a subgoal is also to efficiently use the other cores. Therefore, two more advanced and less intrusive actuators are presented in this section. These actuators are *pulse width modulated interferences* and *frequency scaling*. We present an extended evaluation of *pulse width modulated interferences*[12] and the new approach using a *frequency scaling* methodology.

5.1 Pulse Width Modulated Interferences

Modern multicore systems like the P4080 provide means to halt and continue cores individually. Whenever a core is halted, the clocks are still running, but the core is not fetching or executing instructions [22]. Thus, no accesses to the memory are performed and the interference is stopped. Both actions can be triggered by messages on the back channel of the trace interface, i.e. by writing to control registers. This means that the timing isolation processor (see Figure 1) is able to control the activity of the cores individually from an external device.

To provide not only a binary (on/off) way of setting the performance of the cores, we implemented a (software-based) Pulse Width Modulated (PWM) enabling/disabling of the individual cores, according to the signals from the closed loop controller in short time ranges. We have chosen a PWM period of $1ms$ which is equal to 10 times the $100\mu s$ required to track the application performance. Hence, we can reduce the performance of cores competing with our main core in steps of 10%. During the duty cycle all the opponent cores are active, while for the rest of the period the cores are stopped.

5.2 Frequency Scaling

A slowdown of an interfering low priority core can also be done by a frequency downscaling. This is highly depending on the processor architecture. For example, the number of different clocks and the divider steps are varying between processors. In the case of the P4080 only four possible configurations can be selected: Two different clock sources and to each clock source a divider (divide by two) can be applied. However, in order to have a maximum frequency range, the highest frequency selectable clock frequency is 1.5GHz while the lowest frequency is 800MHz. Therefore, the possible frequencies are 400MHz, 750MHz, 800MHz and 1.5GHz which can be configured for each core individually. This configuration is also possible during runtime from an external device via the debug interface. Since, it is possible to scale down the frequency of an individual core the execution on that core can be slowed down and thus, the interferences with the main core is reduced.

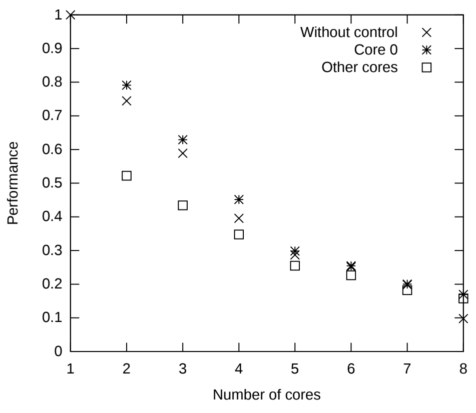
5.3 Evaluation of PWM and Frequency Scaling

For the evaluation of the effectiveness of both the PWM and the frequency scaling approach we used the TACLeBench and the read/write algorithm in three different scenarios:

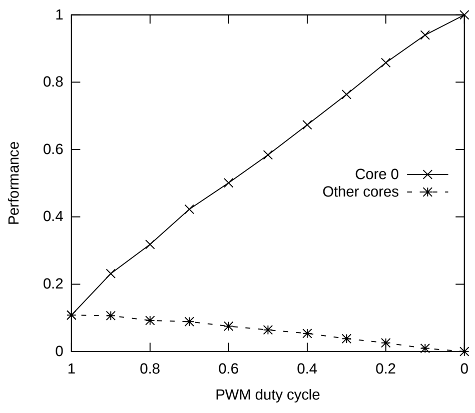
1. Read with seven read opponents (Figure 9): shows the worst case interference scenario,
2. TACLeBench with seven write opponents (Figure 10): shows a realistic application (this benchmark is application oriented and generates realistic traffic on the shared interconnect and memory and profits from local data caches) with worst case opponents,
3. TACLeBench with seven TACLeBench opponents (Figure 11): shows a realistic application on core 0 with realistic opponents.

All the scenarios were evaluated in two different cache configurations: *Realistic* (L1 is enabled) and *Maximum Interferences* (no caches enabled).

For the evaluation of the frequency scaling the performance of the applications was measured without frequency scaling of the opponent cores in the first step. All cores were running with 1.5GHz which is labeled in the figures as *Without control.* In scenario one and three the performance is identical for all applications on all core as the applications are identical. In scenario two the *Without control* performance is depicted individually. In a second step the opponent cores are set to 400MHz which is the minimum configurable speed for the P4080 when the maximum speed is configured to 1.5 GHz. The performance of core 0



(a) Frequency scaling for a varying number of opponents. The opponents are scaled down to 400 MHz while core 0 is running at 1.5 GHz.



(b) PWM halt and resume of the opponent cores for a varying duty cycle while seven opponent cores are running.

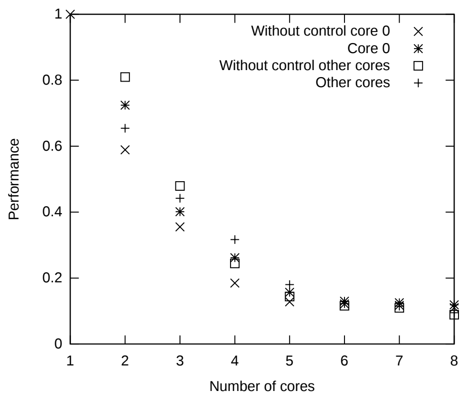
Figure 9 Frequency Scaling and PWM efficiency with the read algorithm (see chapter 2.3.1) on core 0 as well as read opponents on the other seven cores. These measurements were conducted with the *Maximum interference* cache configuration (all caches disabled). A separate analysis (not shown here) indicated that the curves are very similar for enabled local caches as the read algorithm is designed to cause maximum stress on the interconnect and does not take advantage of caches.

as well as the performance of the other cores was measured. As expected, it is visible that the performance of the other cores drops while the performance of core 0 is increased. However, the amount is highly depending on the scenario and cache configuration. The measurements were taken for a varying number of opponent cores. For example, in case of *Number of cores* is four, core 0 is running with 1.5GHz, cores 1 to 3 are running with 400MHz and cores 5 to 8 are idle in the controlled case. In the case of only one core, there is obviously no data plotted for *Core 0* and *Other cores* but this data point was used to normalize the measurements.

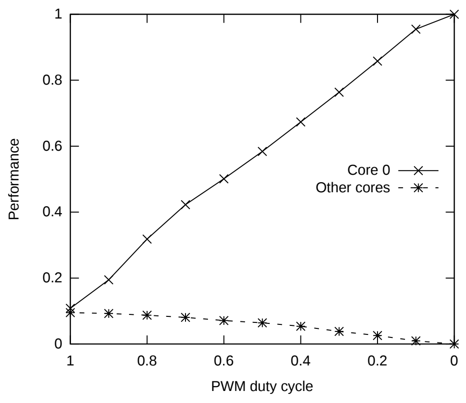
During the evaluation of the PWM approach all eight cores are utilized while the duty cycle of all seven opponent cores is varying from 0 to 100% in steps of 10%. As result, the execution time for the main application as well as the opponent applications is measured.

The results for scenario one with the read algorithm on the main core and up to seven read opponents is shown in Figure 9. It can be observed that the frequency scaling is not able to reduce the interferences from the opponent core enough to keep core 0 at a performance level higher than 90%. The increase of performance in comparison to the uncontrolled case is only around 4% for one opponent core (number of cores equal to two) and is completely negligible for seven opponent cores (number of cores equal to eight). This effect is the same for both cache configurations and can be explained by the cache behavior of the algorithm. Even though the opponent cores are executing instructions with one fourth of the speed, the memory interface is still jammed by the opponents because the memory is even slower. In contrast to that, the results for the PWM approach shown in Figure 9b reveals that even in the case of running seven opponents in parallel, the performance of the main application can be fully recovered. This shows that in the worst case the frequency scaling is not sufficient but the PWM approach can control the performance of the main application at the cost of heavily slowing down the opponents.

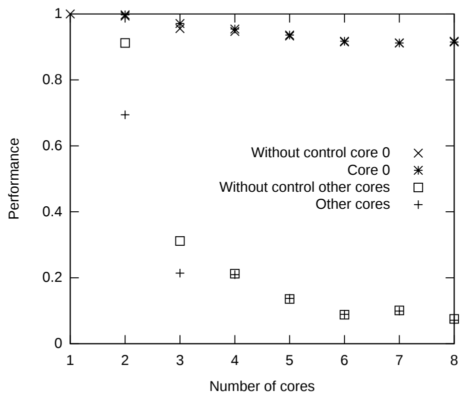
The more realistic scenario of TACLeBench with seven write opponents is shown in Figure 10. In contrast to the other scenarios there are four curves displayed for the frequency scaling instead of three. This is because there are different applications running on core



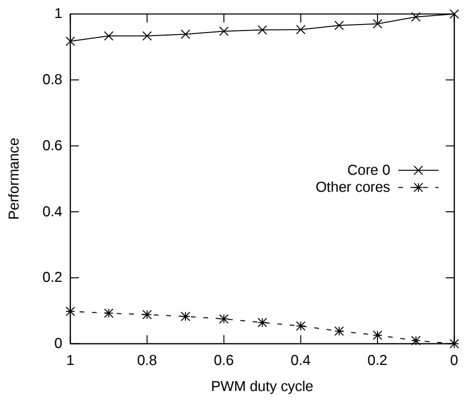
(a) Frequency scaling of the opponents to 400 MHz. No caches enabled.



(b) PWM halt and resume of seven opponents. No caches enabled.

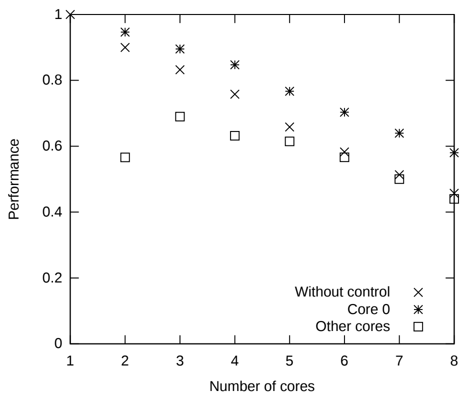


(c) Frequency scaling of the opponents to 400 MHz. L1 cache enabled.

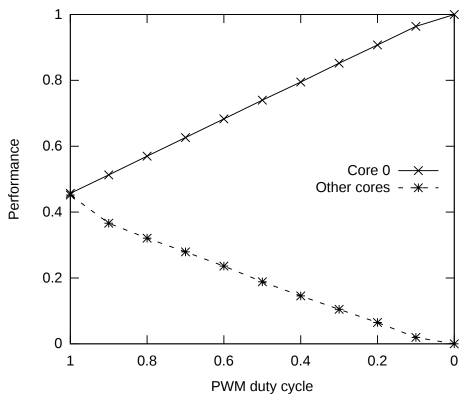


(d) PWM halt and resume of seven opponents. L1 cache enabled.

Figure 10 Frequency scaling and PWM efficiency with TACLe on core 0 and write opponents on the other cores. For Figure **a** and **b** the *Maximum interference* cache configuration applies while for Figure **c** and **d** the *Realistic* cache configuration (L1 cache enabled) was used.



(a) Frequency scaling of the opponents to 400 MHz.



(b) PWM halt and resume of seven opponents for a varying duty cycle.

Figure 11 Frequency scaling and PWM efficiency with TACLe on core 0 and TACLe opponents on the other cores. The measurements were taken with the *Maximum interference* cache configuration.

0 and the other cores which are evaluated separately. In the case of no caches the results are similar to the results in the *read/read* scenario. However, if the L1 cache is enabled the performance of the TACLeBench does not drop below 90% even with seven *write* opponents. The effect of the frequency scaling is not significant because of the cache behavior of the *write* algorithm like in the *read/read* scenario.

For the scenario of TACLeBench with seven TACLeBench opponents the results are displayed in Figure 11. It is visible that the frequency scaling has a significant effect on the performance of the application on core 0. Especially in the case of one and two opponents (two and three cores in Figure 11a) the frequency scaling increases the performance to over 90%. However, even though a performance increase of around 15% compared to the uncontrolled case is visible in the eight core case, frequency scaling is not sufficient for advancing the performance to a level of over 90%. Additional measurements (not shown in the figure) show that in the case of the *Realistic* cache configuration the loss in performance of the TACLe benchmark on core 0 is negligible and the performance of core 0 with seven opponents is still 99%.

Concluding this evaluation, frequency scaling is less efficient for improving performance of core 0 compared to PWM. On the other hand, frequency scaling affects applications running in parallel to core 0 less than PWM.

6 Closed Control Loop

We used two algorithms as control element, a simple threshold-based one and a proportional controller. Both techniques affect all concurrent cores synchronously. The threshold-based algorithm disables the concurrent cores when the slowdown of the main application exceeds a given threshold and enables the cores again when the slowdown falls below the same threshold again. The second technique uses a proportional controller with an actuator based on the PWM activity control and the frequency scaling, respectively, as described in the previous section.

As mentioned in Section 5.1, it is possible to control the cores individually which allows for idling only the low priority cores which create a high traffic on the shared resources. In order to detect the individual core interference, one performance counter in every low priority core has to be read periodically in addition to the performance counters in the critical core. This performance counter is configured with the event *bus interface unit accesses*. Thus, it counts the actual number of L2 cache misses which creates contention. The non-critical cores with the highest number of *bus interface unit accesses* can than individually be slowed down by one of the proposed techniques.

However, for this evaluation of the effectiveness of the control loop we used the TACLe benchmarks as main application and the aforementioned *Write* algorithm as opponents running on seven cores in parallel. We set a maximum slowdown of 4% as target performance of the main application compared to the standalone execution. Individual core interference detection is irrelevant in that case as all opponent cores are running the same application (worst case interference). Therefore, the low priority cores are equally slowed down while the critical application is untouched.

The results of the evaluation are shown in Figure 12, 13 and 14. In the figures, the progress of the TACLeBench over time is displayed in the upper part and the measured slowdown over time in the lower part. The upper part presents the number of executed instruction per μs. For comparison, the standalone (no opponent applications) and the uncontrolled (seven opponent applications without control) executions are displayed. The

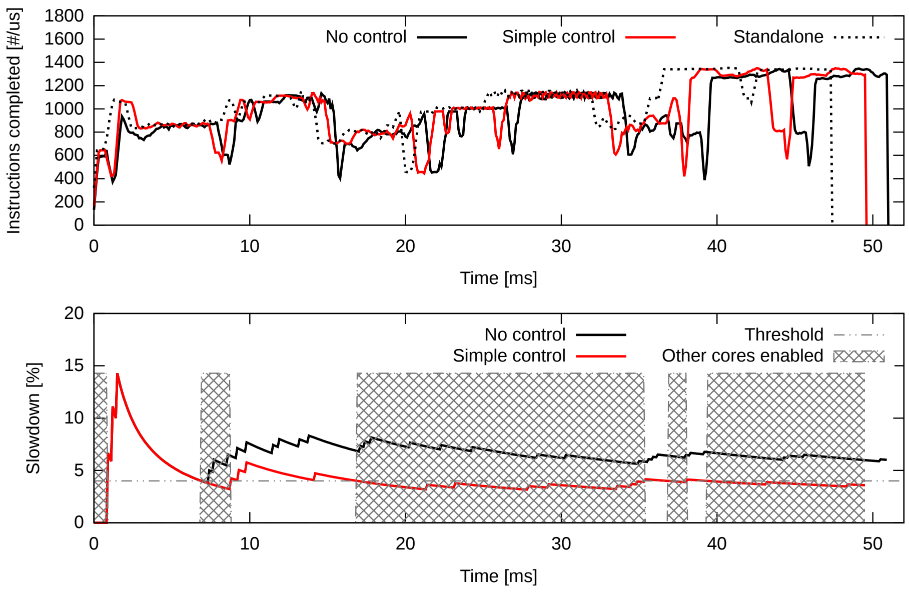


Figure 12 TACLeBench performance over time without control and with applied simple threshold controller.

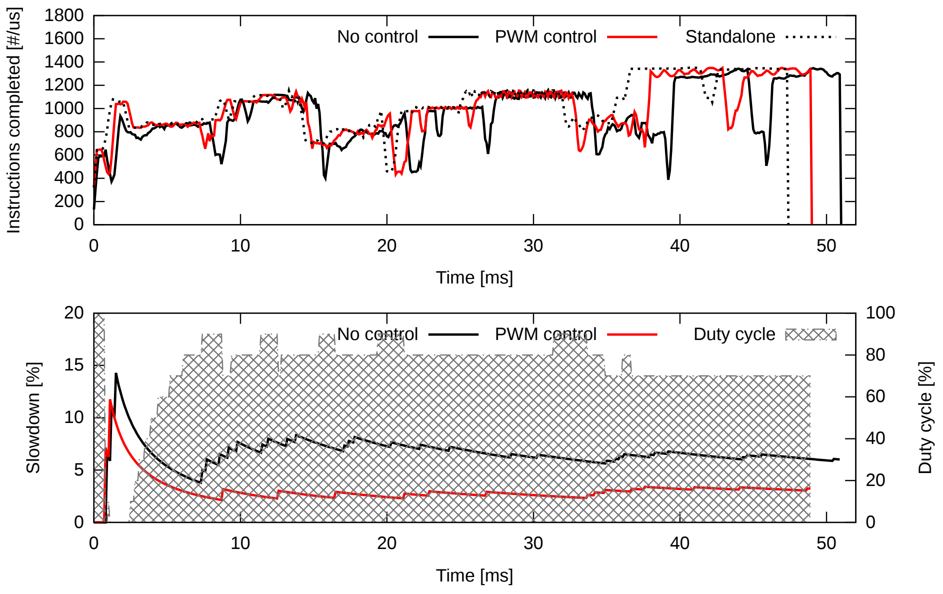


Figure 13 TACLe performance over time without control and with applied PWM controller.

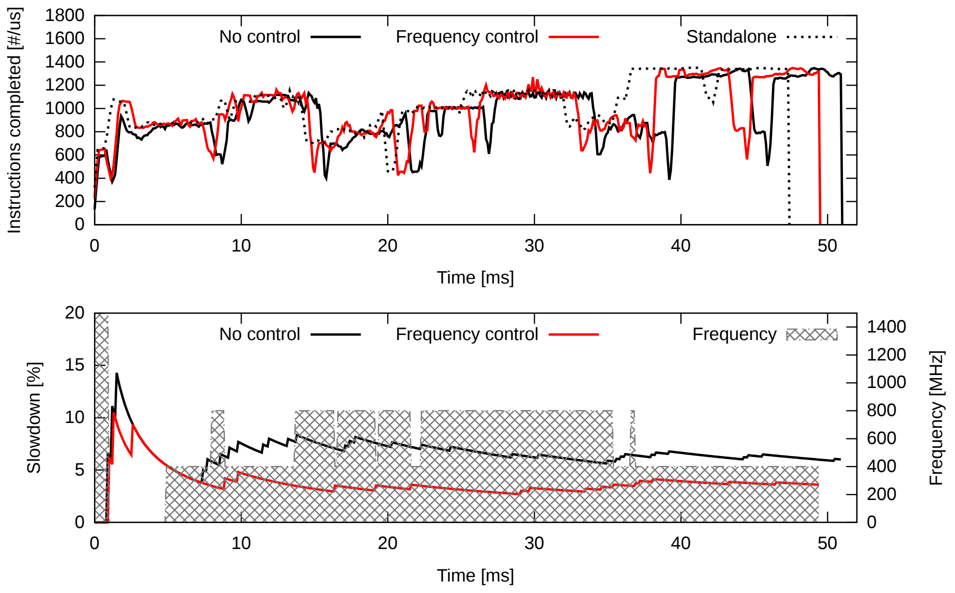


Figure 14 TACLe performance over time without control and with applied frequency scaling controller.

uncontrolled execution takes about 8% longer than the standalone run. The diagrams in the lower part of the figures represent the slowdown of the main application as tracked by the *Fingerprinting*. Since the tracking of progress is based on discrete steps, the performance reductions are manifested in sharp steps. The following phases of smooth performance increases are caused by relative distribution of a slowdown over a longer time, i.e. a one-time delay at the start of the application of 5% is reduced over the total execution time to a much lower slowdown.

In Figure 12 the results of the threshold controller are displayed. The dotted line represents the threshold (4%) i.e. the maximum target slowdown of the main application. The gray shaded boxes identify the times when the other seven cores are active. No grey shading means that the other cores are disabled by the control mechanism. It is visible that the opponent cores are disabled whenever the measured slowdown is higher than the threshold value which keeps the total slowdown in the end at a measured slowdown of 3.59%. The actual total slowdown is 4.44% (measured by comparing the times it took for executing the benchmark in the standalone and controlled case) which means an underestimation of the slowdown by the *Fingerprinting* and an exceedance of the threshold by less than 0.5%. During the total run of one TACLeBench benchmark the opponents are executed for 67% and halted for 33% of the time.

The behavior of the PWM controller is shown in Figure 13. The duty cycles of the competing cores are set according to the measured slowdown. A slowdown of less than 2% allows full performance for all cores, a slowdown above 7% leads to complete disabled competing cores. Between 7% and 2%, the duty cycles are adjusted in 10% steps from 10% to 90% (one step per half percent of slowdown). The grey shaded areas represent the duty cycles of the PWM core activation signal. As can be observed, the 4% target slowdown of the main application is reached after completion (3.23% measured while the actual slowdown was 3.27%). Moreover, the active phases of the competing cores are much longer in time but less intensive. Since we are using a PWM signal, this means that the cores are active for many but smaller periods. With this PWM control, the seven *bad guys* get 74% of the cores' performance while the main application still meets the performance requirements which is an advantage of 7% over the threshold based actuator. Moreover, actual slowdown of the main application is better than the targeted acceptable threshold of 4%.

The frequency scaling approach is displayed in Figure 14. The possible frequencies of the opponent cores are 400MHz, 800MHz and 1.5GHz. Furthermore, the core can be halted. Similar to the PWM approach a slowdown of less than 2% allows full performance for all cores, a slowdown above 7% leads to completely disabled competing cores. Between 7% and 2%, the frequencies are adjusted in linear intervals. The grey shaded areas represent the frequencies of the opponent cores. The slowdown of the main application is reduced with a total measured value of 3.60% (real slowdown: 4.44%). However, this was not possible by only scaling down the cores. During the period of high interference in the beginning of the execution the opponent cores had to be halted for a sufficient reduction of the interferences. An assessment of the cores processing time compared to the aforementioned approaches does not make sense in this case. The frequency scaling of a core cannot be compared with halting and continuing a core because the performance of a scaled down core is highly dependent on the instructions executed.

The scenarios one and three show slight violations in actual slowdown compared to the target threshold. This is because of an underestimation of the slowdown by the *Fingerprinting* caused by the technologies latency. Adding a safety margin when defining the acceptable bounds could help here.

7 Conclusion and Future Work

This paper presents a virtual timing isolation of one main application running on one core from all other cores of a multicore processor. The proposed technique is based on hardware external to the multicore and completely transparent to the main application. The basic idea is to apply a single-core execution based Worst Case Execution Time analysis and to accept a predefined slowdown during multicore execution. If the slowdown exceeds predefined acceptable bounds, interferences will be reduced by thwarting other cores to keep the main application's progress inside the bounds.

We evaluated the accuracy of the transparent tracking of the application's progress (*Fingerprinting*), the effectiveness of different thwarting techniques, and the performance of a complete closed control loop using a simple P-controller. The latter shows that it is possible to transparently enable an application staying within given timing bounds even though there are a maximum of seven opponents flooding the shared interconnect with traffic. Evaluations indicated a slight underestimation of the application's slowdown which could be compensated by adding a safety margin. Determining a suitable range for this safety margin is part of future work.

It is planned to extend the thwarting in order to affect only cores driving high traffic on the interconnect instead of all competing cores. This can be reached by evaluating the interconnect accesses of the other cores to identify cores with high influence. Moreover, a combination of frequency scaling and PWM driven thwarting would be interesting for more effective and fine-grained interference control. Also using a more complex control algorithm like a full PID controller could be useful to increase performance of competing cores. The target of future research will be enabling more than one core running hard real-time applications.

References

1 The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface, 2012.

2 Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J Cazorla. On the Tailoring of CAST-32A Certification Guidance to Real COTS Multicore Architectures. In *12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2017.

3 Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2017.2`.

4 Airbus. Future of urban mobility. 2018. http://www.airbus.com/innovation/urban-air-mobility.html.

5 S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, Aug 2012. `doi:10.1109/RTCSA.2012.48`.

6 G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, Apr 2001. `doi:10.1109/12.919277`.

7 Certification Authorities Software Team (CAST). Position Paper CAST-32A: Multi-core Processors. November 2016. URL: `https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf`.

8 Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning Schedulers for Legacy Real-time Applications. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 55–68, New York, NY, USA, 2010. ACM. `doi:10.1145/1755913.1755921`.

9 Evelyn Duesterwald and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *In International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–231, 2003.

10 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

11 Johannes Freitag and Sascha Uhrig. Dynamic interference quantification for multicore processors. In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pages 1–6, Sept 2017. `doi:10.1109/DASC.2017.8101991`.

12 Johannes Freitag and Sascha Uhrig. Closed Loop Controller for Multicore Real-Time Systems. In Mladen Berekovic, Rainer Buchty, Heiko Hamann, Dirk Koch, and Thilo Pionteck, editors, *Architecture of Computing Systems – ARCS 2018*, pages 45–56, Cham, 2018. Springer International Publishing.

13 Yong Fu, Nicholas Kottenstette, Chenyang Lu, and Xenofon D. Koutsoukos. Feedback Thermal Control of Real-time Systems on Multicore Processors. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 113–122, New York, NY, USA, 2012. ACM. `doi:10.1145/2380356.2380379`.

14 S. Girbal, X. Jean, J. Le Rhun, Daniel Gracia Pérez, and M. Gatti. Deterministic platform software for hard real-time systems using multi-core COTS. In *2015 IEEE/AIAA 34th*

Digital Avionics Systems Conference (DASC), pages 8D4–1–8D4–15, Sept 2015. doi:10.1109/DASC.2015.7311481.

15 Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, Hadi Ahmadi Balef, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi. *NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications*, pages 491–530. Springer Netherlands, Dordrecht, 2017. doi:10.1007/978-94-017-7267-9_17.

16 N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. doi:10.1109/RTAS.2016.7461323.

17 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed Run-time WCET Controller for Concurrent Critical Tasks in Mixed-critical Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 139:139–139:148, New York, NY, USA, 2014. ACM. doi:10.1145/2659787.2659799.

18 Lilium GmbH. Lilium Home Page. 2018. https://lilium.com/.

19 M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Power Optimization in Embedded Systems via Feedback Control of Resource Allocation. *IEEE Transactions on Control Systems Technology*, 21(1):239–246, Jan 2013. doi:10.1109/TCST.2011.2177499.

20 J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, May 2012. doi:10.1109/EDCC.2012.27.

21 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *ECRTS*, pages 109–118. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.20.

22 NXP Semiconductors. *e500mc Core Reference Manual*, 2013. Rev. 3.

23 NXP Semiconductors. *P4080 QorIQ Multicore Communication Processor Reference Manual*. NXP Semiconductors, rev 2 edition, 2014.

24 D. R. Sahoo, S. Swaminathan, R. Al-Omari, M. V. Salapaka, G. Manimaran, and A. K. Somani. Feedback control for real-time scheduling. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, volume 2, pages 1254–1259 vol.2, May 2002. doi:10.1109/ACC.2002.1023192.

25 Martin Schoeberl, Sahar Abbaspourseyedi, Alexander Jordan, Evangelia Kasapaki, Wolfgang Puffitsch, Jens Sparsø, Benny Akesson, Neil Audsley, Jamie Garside, Raffaele Capasso, Alessandro Tocchi, Kees Goossens, Sven Goossens, Yonghui Li, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, Peter Puschner, André Rocha, and Cláudio Silva. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

26 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, and Richard Bradford. Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors. Technical report, 2014.

27 T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic,

J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *2013 Euromicro Conference on Digital System Design*, pages 363–370, Sept 2013. doi:10.1109/DSD.2013.46.

AdaptMC: A Control-Theoretic Approach for Achieving Resilience in Mixed-Criticality Systems

Alessandro Vittorio Papadopoulos
Mälardalen University
Västerås, Sweden
alessandro.papadopoulos@mdh.se

Enrico Bini
University of Turin
Turin, Italy
bini@di.unito.it

Sanjoy Baruah
Washington University
St. Louis (MO), USA
baruah@wustl.edu

Alan Burns
University of York
York, UK
alan.burns@york.ac.uk

Abstract

A system is said to be resilient if slight deviations from expected behavior during run-time does not lead to catastrophic degradation of performance: minor deviations should result in no more than minor performance degradation. In mixed-criticality systems, such degradation should additionally be criticality-cognizant. The applicability of control theory is explored for the design of resilient run-time scheduling algorithms for mixed-criticality systems. Recent results in control theory have shown how appropriately designed controllers can provide guaranteed service to hard-real-time servers; this prior work is extended to allow for such guarantees to be made concurrently to multiple criticality-cognizant servers. The applicability of this approach is explored via several experimental simulations in a dual-criticality setting. These experiments demonstrate that our control-based run-time schedulers can be synthesized in such a manner that bounded deviations from expected behavior result in the high-criticality server suffering no performance degradation and the lower-criticality one, bounded performance degradation.

2012 ACM Subject Classification Computing methodologies → Control methods, Computer systems organization → Real-time systems

Keywords and phrases mixed criticality, control theory, run-time resilience, bounded overloads

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.14

Supplement Material ECRTS Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.4.2.1`

Funding This work was partially supported by the Swedish Foundation for Strategic Research under the project "Future factories in the cloud (FiC)" with grant number GMT14-0032. This work is also supported by NSF grants CNS 1409175, CPS 1446631, and CNS 1563845.

1 Introduction

There is an increasing trend in embedded systems towards implementing multiple functionalities upon a shared platform. It may be the case that all these functionalities are not equally important to the overall correctness of the embedded system; one widely-studied model for representing timing requirements in such systems was proposed by Vestal in a seminal paper [33]. Vestal observed that "In many applications, the consequences of missing a deadline vary in severity from task to task. In RTCA DO 178B, for example, system safety analysis assigns to each task a *criticality level* (ranging from A to D), where erroneous behavior by a level A task might cause loss of aircraft but erroneous behavior by a level D task might at worst cause inconvenient or suboptimal behavior."[1] Vestal went on to conjecture that "the higher the degree of assurance required that actual task execution times will never exceed the WCET parameters used for analysis, the larger and more conservative the latter values become in practice." (This conjecture appears reasonable. Very conservative WCET-estimation tools have been developed, typically based upon static analysis of code, that yield WCET bounds that may be very large, but that we can trust to a very high level of assurance. Less conservative WCET-estimation tools, which are typically measurement based, tend to obtain smaller estimates, but these estimates may be trust-worthy to lower levels of assurance since the worst-case behaviors of the system may not have become revealed during the measurements.) The "Vestal model" for representing, and validating the correctness of, mixed-criticality systems is based upon this conjecture. In this model,

- **§1.** A fixed number of distinct criticality levels are defined throughout the system. In this paper, we will assume that there are two such criticality levels, designated LO and HI, with the interpretation that functionalities designated as being of the LO criticality level need to have their correctness validated to a lower level of assurance than functionalities designated as being of the HI criticality level.
- **§2.** Each piece of code in the system is characterized as being of one of the criticality levels LO or HI, and by two WCET parameter estimates. One WCET estimate is determined using tools and techniques consistent with the lower criticality level LO, while the other estimate is determined using tools and techniques consistent with the higher criticality level HI.
- **§3.** Prior to run-time, the correct timing behavior (e.g., meeting deadlines) of all the functionalities are validated under the assumption that each piece of code will execute for a duration not exceeding its LO-criticality WCET estimate; in addition, the correct timing behavior of the HI-criticality functionalities (but not the LO-criticality ones) are validated under the assumption that each piece of code may execute for a duration up to its HI-criticality WCET estimate.

[1] RTCA DO 178B is a guideline dealing with the safety of safety-critical software used in certain avionics systems. Although the term "criticality" typically has a precise technical meaning in most safety standards documents, its use in [33], and subsequent use in much of the mixed-criticality scheduling theory literature, appears to be in a rather general sense as a designation of the level of assurance against failure that is desired. In this paper we are using the term in this more general sense, in keeping with prior literature in mixed-criticality scheduling.

1.1 Verification versus resilience

The Vestal approach to modeling and analysis of mixed-criticality systems, as originally proposed [33], is concerned solely with *verification* – determining, prior to run-time, whether a system will behave correctly during run-time if its run-time behavior is compliant with the models used to represent it. Clearly, such pre-runtime verification is desirable in safety-critical systems. There is an additional aspect of correctness that is also desirable: the system's run-time behavior should be *resilient* or robust in the event that run-time behavior does not conform to the models that were assumed during verification; if this happens, a robust system design ensures that performance degrades gracefully, if at all. *It is this run-time resilience aspect of system behavior that is the primary focus of this paper.* (While the precise semantics of graceful degradation should be for a particular system may depend upon the characteristics of the system, some general principles are applicable; for example, less important aspects of system functionalities should be compromised before more important ones.)

The Vestal model of [33] and its derivatives and generalizations have formed the basis of a large body of research: schedulability tests, scheduling algorithms, etc. – see, e.g., [5, 6] for a survey. Much of this research is focused upon the pre-runtime verification aspect of correctness rather than the run-time resilience. For instance, many mixed-criticality scheduling algorithms allow for LO-criticality pieces of code to be aborted if any piece of code executes beyond its LO-criticality WCET estimate. Such a scheduling algorithm may still pass the pre-runtime verification test (since such tests are only concerned with the correctness of the HI-criticality functionalities under such circumstances), but would not be considered resilient. Some recent research has attempted to provide some resilience to LO-criticality pieces of code in the event of some piece of code executing beyond its LO-criticality WCET estimate; these approaches are reviewed in Section 7.

1.2 This research

In this paper, we explore the use of control-theoretic principles to achieve resilience in mixed-criticality systems. We consider over-runs of HI-criticality pieces of code (in the sense of them executing for more than their LO-criticality WCET estimates) to be *rare events* that are best coped with by run-time adaptability. Some over-runs can be masked by under-runs by other HI-criticality pieces of code; others will require system-wide adaptation. These adaptations should be commensurate with the scale of the over-run – dropping all LO-criticality pieces of code because a single HI-criticality piece of code has executed for slightly more than its LO-criticality WCET is clearly an over-reaction. A resilient system should cope with uncertainty in a measured way.

Some recent advances in real-time control (see, e.g., [22] and the references therein) have motivated us to explore whether the desired resilience can be achieved using a control-theoretic approach. The scheduling strategy we propose has the HI-criticality workload executing within an execution-time server that is provisioned with a budget sufficient to satisfy the LO-criticality WCET requirements of this HI-criticality workload; another, similar, server is used to encapsulate the execution requirements of the LO-criticality workload. At run-time if the HI-criticality server's budget proves inadequate for meeting the execution requirements of the HI-criticality workload (due to some HI-criticality pieces of code executing for more than their LO-criticality WCET estimates) then the system is deemed to have suffered a *disturbance* or *perturbation*. We employ a *control feedback* mechanism to govern budget allocations going forward from the disturbance. This control-theoretic feedback approach allows a number of questions to be answered concerning the run-time behavior of the scheduling strategy, such as

- How long following a disturbance will it take the system to return to a non-perturbed state?
- What guaranteed level of service can be obtained for the LO-criticality workload?
- What is the maximum *magnitude* of disturbance that can be accommodated allowing for stable control and for the HI-criticality workload to remain schedulable?

1.3 Organization

The remainder of this paper is organized as follows. Section 2 presents the background for this work, while Section 3 presents AdaptMC, the proposed approach, in detail. Section 4 discusses how AdaptMC is designed and tuned, while Section 5 presents how hard real-time guarantees can be provided, by means of the calculation of the supply bound function. Section 6 presents a numerical evaluation of AdaptMC. Section 7 reviews the related work, while Section 8 concludes the paper.

2 Background

The use of feedback control to allocate resources has traditionally been applied to time-varying workloads [28, 7, 1], and the kinds of offered guarantees have been probabilistic or soft real-time. Recently, however, a control scheme called the *Self-Adaptive Server (SAS)* has been proposed [22], that provides both good average behavior and hard real-time guarantees. Such a guarantee is given by computing the supply bound function [21, 18, 27, 2] of a periodic resource supply controlled by feedback [17].

The main idea behind SAS is as follows. Each server in the system is assigned a budget of time to execute. The server is allowed execute more or less than the budget, but at the next round it will be assigned a budget that is corrected with a term that is proportional to the over- or under-run of the server. In [22] this simple, yet effective, control structure is analyzed under the assumption that the maximum over- or under-run are bounded. The designed controller is proven to effectively adapt the budget at run-time, while the supply bound function associated with the controller can be computed offline.

3 The Proposed Approach

We are concerned with mixed-criticality systems in which the LO-criticality WCET values represent typical or common-case behavior: executions *rarely* exceed these WCET values and when they do, it is typically *by small amounts*. We seek to devise resilient scheduling strategies for such mixed-criticality systems. As briefly stated in Section 1, our proposed scheduling strategy uses two servers, one each for servicing the HI-criticality and LO-criticality workloads.[2] In dimensioning these servers' budgets, our objective will be to modestly over-allocate the HI-criticality server in the sense that "most of the time" we would expect the entire provisioned budget to not be needed. If an occasional modest over-run occurs in

[2] For the kinds of application systems that we are interested in, work (in the form of "jobs") is typically generated by recurrent – periodic and sporadic – tasks; determining appropriate budget and period parameters for servers capable of accommodating the computational requirements of such recurrent tasks is an important issue that has been widely studied in the real-time scheduling community [18, 27, 2]. However, the issue of dimensioning such servers is orthogonal to the focus of this paper and we will not discuss it further, instead assuming that some appropriate scheme is used to determine appropriate server parameters such that if all jobs execute at their LO-criticality WCET estimates, then each server is able to correctly execute those jobs for which it is responsible.

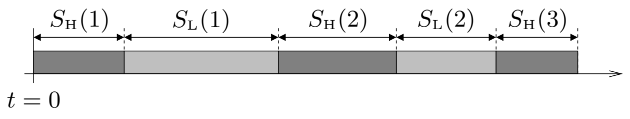


Figure 1 Server schedule over time.

the amount of execution required by the HI-criticality server (say, by an amount x over the budgeted amount), our run-time scheduling strategy is to allow the HI-criticality server to over-execute by this entire amount x, and then reduce the budget for the LO-criticality server by an amount somewhat smaller than x. Informally speaking, the hope is that after dealing with this one-time over-run, the HI-criticality server will not need to use its entire budgeted amount for some duration, and hence can compensate the LO-criticality server over this duration. However, (as we will see) our control-based scheduling strategy is robust to scenarios in which the HI-criticality server over-runs for an extended duration as well; if this happens, the LO-criticality server ends up getting under-served over an extended duration.

In order to develop a control-based strategy capable of achieving these goals, we needed to extend and adapt SAS (Self-Adaptive Server) [22] in several directions. The feedback mechanism derived in this paper is an extension of SAS to the mixed-criticality context that enables:

1. the adjustment of server budgets based on disturbances at both HI-criticality and LO-criticality servers (achieved by *cross gains* of the controller), and
2. the exploitation of the asymmetric nature of disturbances that are permitted for the LO-criticality server (which may occasionally be under-served but never receives more than its budgeted amount) to provide less conservative supply bound functions.

The presence of these two characteristics, needed in the mixed-criticality context, renders the results in [22] inapplicable directly; hence the extensions reported here. Section 3.1 below describes the adaptive scheduling strategy we have developed; the control algorithm underpinning this strategy is described in Section 3.2

3.1 Run-Time Scheduling Strategy

We propose a 2-levels hierarchical scheduler with two schedulers at the top level, one for servicing LO-criticality work and the other, for servicing HI-criticality work (see Figure 1). Let $\bar{Q}_{\text{H}}$ and $\bar{Q}_{\text{L}}$ denote the *target budgets* for the two servers, and $\bar{P} = \bar{Q}_{\text{H}} + \bar{Q}_{\text{L}}$ the *target period*. We will describe later the manner in which values are assigned to these target budget parameters; intuitively speaking, we would assign them values such that under normal circumstances (i.e., all jobs completing within their LO-criticality WCET estimates) a periodic schedule with period $\bar{P}$ in which the HI-criticality server executing for a duration $\bar{Q}_{\text{H}}$ is followed by the LO-criticality server executing for a duration $\bar{Q}_{\text{L}}$, would meet all timing requirements for all the HI-criticality and the LO-criticality workload.

During run-time these two servers are repeatedly scheduled alternately. Let us refer to the k'th time that both servers are scheduled as the *k'th round*. Let $Q_{\text{H}}(k)$ and $Q_{\text{L}}(k)$ denote the *tentative budgets* that the control algorithm computes at the end of the k'th round, for allocating to the two servers for the $(k+1)$'th round. Initially, we have $Q_{\text{H}}(0) = \bar{Q}_{\text{H}}$ and $Q_{\text{L}}(0) = \bar{Q}_{\text{L}}$; i.e., for the first round the tentative budgets are set to be equal to the target budgets.

Now suppose that during the $(k+1)$'th round for some k, the HI-criticality server needs to execute for a duration **greater** than this tentative budget $Q_{\text{H}}(k)$ in order to ensure the correct execution of all HI-criticality jobs (budget overrun). We allow it to do so, and let $S_{\text{H}}(k+1)$

denote the duration for which it executes – $S_{\text{H}}(k+1)$ is called the *actual budget* assigned to the HI-criticality server during the $(k+1)$'th round, and $\varepsilon_{\text{H}}(k) = \big(S_{\text{H}}(k+1) - Q_{\text{H}}(k)\big)$ is called the *disturbance* experienced by the HI-criticality server, i.e., the discrepancy between the target and actual budget. In response to such a disturbance, our control algorithm modifies the tentative budgets $Q_{\text{H}}(k+1)$ and $Q_{\text{L}}(k+1)$ computed for both servers for the next round, to compensate for the budget overrun and preserve the bandwidth.

3.2 The Control Algorithm

As stated earlier, our control-based scheduler is designed under the assumption that jobs executing beyond their LO-criticality WCET estimates will be rare events. The target budget $\bar{Q}_{\text{H}}$ for the HI-criticality server should be chosen to somewhat exceed the minimum needed in order to accommodate the LO-criticality WCET requirements for all the HI-criticality jobs; hence, if only one or a few jobs over-run their LO-criticality WCETs during a round, such over-runs are often masked by the excess budget and by under-runs of other HI-criticality jobs. It should only rarely be the case that such over-runs during any round get expressed as disturbances (i.e., as an $\varepsilon_{\text{H}}(k)$ value for some k); in the rare events when this does happen, our control algorithm requires that it be of magnitude that is bounded by an a priori known constant $\bar{\varepsilon}_{\text{H}}$: $|\varepsilon_{\text{H}}(k)| \le \bar{\varepsilon}_{\text{H}}$.

In order to accommodate these disturbances in the HI-criticality servers, our control algorithm will occasionally under-schedule the LO-criticality server, providing it a supply $S_{\text{L}}(k+1)$ that is strictly less than the tentative budget $Q_{\text{L}}(k)$ that had been computed for it – when this happens, the LO-criticality server is said to experience a disturbance $\varepsilon_{\text{L}}(k) = \big(S_{\text{L}}(k+1) - Q_{\text{L}}(k)\big)$. We assume that such a disturbance will also be of magnitude that is bounded by another a priori known constant $\bar{\varepsilon}_{\text{L}}$, i.e., maximum budget over-run of the LO-criticality server.

Analogously, our run-time scheduler also bounds the "negative" disturbance to the HI-criticality server: the amount by which the actual amount of execution supplied during a round is less than the tentative budget, to have a magnitude no greater than $\bar{\varepsilon}_{\text{H}}$. Summarizing the above discussion on disturbances, we obtain the following bounds on the magnitudes of the disturbances that could be experienced by both the servers:

$$-\bar{\varepsilon}_{\text{H}} \le \varepsilon_{\text{H}}(k) \le \bar{\varepsilon}_{\text{H}}, \quad -\bar{\varepsilon}_{\text{L}} \le \varepsilon_{\text{L}}(k) \le 0. \tag{1}$$

As we had stated earlier, the actual budgets $S_{\text{H}}(k+1)$ and $S_{\text{L}}(k+1)$ assigned to the servers may be expressed as being equal to the computed tentative budgets $Q_{\text{H}}(k)$ and $Q_{\text{L}}(k)$, plus the disturbances $\varepsilon_{\text{H}}(k)$ and $\varepsilon_{\text{L}}(k)$.

$$\begin{aligned} S_{\text{H}}(k+1) &= Q_{\text{H}}(k) + \varepsilon_{\text{H}}(k) \\ S_{\text{L}}(k+1) &= Q_{\text{L}}(k) + \varepsilon_{\text{L}}(k) \end{aligned}$$

The tentative budgets $Q_{\text{H}}(k+1)$ and $Q_{\text{L}}(k+1)$ that are computed by the control algorithm may similarly be expressed as the sum of tentative budgets computed for the previous round and a corrective term (called the "*control signal*") denoted $u_{\text{H}}(k)$ and $u_{\text{L}}(k)$, that is computed by the control algorithm at the end of each round:

$$\begin{aligned} Q_{\text{H}}(k+1) &= Q_{\text{H}}(k) + u_{\text{H}}(k) \\ Q_{\text{L}}(k+1) &= Q_{\text{L}}(k) + u_{\text{L}}(k) \end{aligned}$$

Letting

$$\boldsymbol{x}(k) = \begin{bmatrix} S_{\mathrm{H}}(k) \\ S_{\mathrm{L}}(k) \\ Q_{\mathrm{H}}(k) \\ Q_{\mathrm{L}}(k) \end{bmatrix}, \quad \boldsymbol{u}(k) = \begin{bmatrix} u_{\mathrm{H}}(k) \\ u_{\mathrm{L}}(k) \end{bmatrix}, \quad \boldsymbol{\varepsilon}(k) = \begin{bmatrix} \varepsilon_{\mathrm{H}}(k) \\ \varepsilon_{\mathrm{L}}(k) \end{bmatrix},$$

one can express the *control system dynamics* – the change in values of the actual and tentative budgets across rounds that we have discussed above – in a more compact form, as follows:

$$\boldsymbol{x}(k+1) = \overbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\boldsymbol{A}} \boldsymbol{x}(k) + \overbrace{\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}}^{\boldsymbol{B}_u} \boldsymbol{u}(k) + \overbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}}^{\boldsymbol{B}_\varepsilon} \boldsymbol{\varepsilon}(k). \tag{2}$$

We now discuss how the control signals are computed by the control algorithm (this computation is commonly referred to as the *control strategy*). In designing the controller, we assign values to four real-valued *gain* parameters $K_{\mathrm{HH}}, K_{\mathrm{HL}}, K_{\mathrm{LH}}$, and K_{LL} – the parameter design is discussed in Section 4 – and compute the control signals as follows:

$$\begin{aligned} u_{\mathrm{H}}(k) &= K_{\mathrm{HH}}\big(\bar{Q}_{\mathrm{H}} - S_{\mathrm{H}}(k)\big) &&+ {}^{K_{\mathrm{HL}}}/_{\gamma}\big(\bar{Q}_{\mathrm{L}} - S_{\mathrm{L}}(k)\big), \\ u_{\mathrm{L}}(k) &= \gamma K_{\mathrm{LH}}\big(\bar{Q}_{\mathrm{H}} - S_{\mathrm{H}}(k+1)\big) &&+ K_{\mathrm{LL}}\big(\bar{Q}_{\mathrm{L}} - S_{\mathrm{L}}(k)\big). \end{aligned} \tag{3}$$

The parameters $K_{\mathrm{HH}}, K_{\mathrm{HL}}, K_{\mathrm{LH}}$, and K_{LL} weigh the discrepancy between the target and actual budgets; the values assigned to these parameters reflect the effect each discrepancy has on the control signal. (Observe that in computing the control signal $u_{\mathrm{L}}(k)$ that will be applied to the LO-criticality server, we are able to exploit the fact that the value of $S_{\mathrm{H}}(k+1)$ is already known when the LO-criticality server is scheduled during the $(k+1)$'th round; we therefore choose to exploit this fact to compute a "better" values for $u_{\mathrm{L}}(k)$.)

By substituting the control strategy as represented by Eqn (3) into Eqn (2), rearranging terms, and letting γ denote the ratio of the target budgets, i.e., $\gamma = {}^{\bar{Q}_{\mathrm{L}}}/_{\bar{Q}_{\mathrm{H}}}$, the closed-loop system dynamics may be represented as follows:

$$S_{\mathrm{H}}(k+1) = Q_{\mathrm{H}}(k) + \varepsilon_{\mathrm{H}}(k) \tag{4}$$

$$S_{\mathrm{L}}(k+1) = Q_{\mathrm{L}}(k) + \varepsilon_{\mathrm{L}}(k) \tag{5}$$

$$Q_{\mathrm{H}}(k+1) = Q_{\mathrm{H}}(k) + K_{\mathrm{HH}}(\bar{Q}_{\mathrm{H}} - S_{\mathrm{H}}(k)) + {}^{K_{\mathrm{HL}}}/_{\gamma}(\bar{Q}_{\mathrm{L}} - S_{\mathrm{L}}(k)) \tag{6}$$

$$Q_{\mathrm{L}}(k+1) = Q_{\mathrm{L}}(k) + K_{\mathrm{LL}}(\bar{Q}_{\mathrm{L}} - S_{\mathrm{L}}(k)) + K_{\mathrm{LH}}\gamma(\bar{Q}_{\mathrm{H}} - S_{\mathrm{H}}(k+1)) \tag{7}$$

or, in a more compact way:

$$\boldsymbol{x}(k+1) = \boldsymbol{A}_{CL}\, \boldsymbol{x}(k) + \boldsymbol{B}_Q\, \bar{\boldsymbol{Q}} + \boldsymbol{B}_{\varepsilon,CL}\, \boldsymbol{\varepsilon}(k) \tag{8}$$

with

$$\boldsymbol{A}_{CL} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -K_{\mathrm{HH}} & -\frac{K_{\mathrm{HL}}}{\gamma} & 1 & 0 \\ 0 & -K_{\mathrm{LL}} & -\gamma K_{\mathrm{LH}} & 1 \end{bmatrix}, \quad \boldsymbol{B}_Q = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ K_{\mathrm{HH}} & \frac{K_{\mathrm{HL}}}{\gamma} \\ \gamma K_{\mathrm{LH}} & K_{\mathrm{LL}} \end{bmatrix},$$

$$\bar{\boldsymbol{Q}} = \begin{bmatrix} \bar{Q}_{\mathrm{H}} \\ \bar{Q}_{\mathrm{L}} \end{bmatrix}, \quad \boldsymbol{B}_{\varepsilon,CL} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ -\gamma K_{\mathrm{LH}} & 0 \end{bmatrix}.$$

The eigenvalues of $\boldsymbol{A}_{CL}$ determine the convergence time towards the value $\overline{\boldsymbol{x}}$ for the system state. These can be obtained from the characteristic polynomial of $\boldsymbol{A}_{CL}$:

$$p(z) = z^4 - 2z^3 + (K_{\text{HH}} + K_{\text{LL}} + 1)\, z^2 - (K_{\text{HH}} + K_{\text{LL}} + K_{\text{HL}}K_{\text{LH}})z + K_{\text{HH}}K_{\text{LL}}. \tag{9}$$

Since the considered system is linear, we can use the superposition principle[3], and consider separately the effect of $\bar{\boldsymbol{Q}}$ and $\boldsymbol{\varepsilon}$ on the evolution of $\boldsymbol{x}$. The z-transform of (8) is:

$$\boldsymbol{X}(z) = (z\boldsymbol{I} - \boldsymbol{A}_{CL})^{-1} \left(\boldsymbol{x}(0) + \boldsymbol{B}_Q\bar{\boldsymbol{Q}} + \boldsymbol{B}_{\varepsilon,CL}\boldsymbol{E}(z)\right) \tag{10}$$

Evaluating the transfer function from the error ε to the state x for $z = 1$ computes, in control theoretical terms, the asymptotic effect of the unitary constant disturbance ε on the state x; in the considered case, evaluating $(z\boldsymbol{I} - \boldsymbol{A}_{CL})^{-1}\boldsymbol{B}_{\varepsilon,CL}$ for $z = 1$ yields:

$$(\boldsymbol{I} - \boldsymbol{A}_{CL})^{-1}\boldsymbol{B}_{\varepsilon,CL} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}$$

that proves that the effect of $\boldsymbol{\varepsilon}$ on $\boldsymbol{S}$ (the first two rows) vanishes asymptotically to zero independently of the values assigned to the gain parameters. The effect of a unitary constant disturbance on the budgets $\boldsymbol{Q}$, on the other hand, is to compensate $\boldsymbol{\varepsilon}$ by reducing the budget of exactly a unity so that value of $\boldsymbol{S}$ will compensate perfectly the disturbance $\boldsymbol{\varepsilon}$.

4 Designing the Control Algorithm

In Section 3 we described how the control logic can be used to adjust the resource budgets allocated to HI and LO-criticality servers. In this section, we are going to explore the assignment of values to the control gain parameters K_{HH}, K_{HL}, K_{LH}, and K_{LL} such that the resulting budget dynamics are guaranteed to possess the desirable control-theoretic properties of *compensation* and *stability*.

▶ **Definition 1** (Compensation property). A single disturbance $\varepsilon(k)$ on the HI/LO-criticality server results in an opposite or null effect on the value of $S(k+1)$ (i.e., the actual budget) of the LO/HI-criticality server, i.e.,

$$\exists n > 0 : \varepsilon_i(k) = -\alpha(k+n)u_j(k+n), \quad \alpha(k+n) \geq 0, i, j \in \{\text{H}, \text{L}\}, \text{ and } i \neq j.$$

The intuition of the compensation property is that whenever the HI-criticality server exceeds its budget ($S_{\text{H}}(k+1) > Q_{\text{H}}(k)$), the LO-criticality server compensates for this disturbance by temporarily reducing its budget. On the other hand, when the LO-criticality server requires less time for its execution ($S_{\text{L}}(k+1) < Q_{\text{L}}(k)$), then the HI-criticality server will be allowed to temporarily increase its budget. Finally, when the HI-criticality server executes for less time ($S_{\text{H}}(k+1) < Q_{\text{H}}(k)$), then the LO-criticality server can temporarily increase its budget.

The overall objective is to both preserve the bandwidth of the two servers, and to reach the target period $\bar{P} = \bar{Q}_H + \bar{Q}_L$.

[3] The *superposition principle* for linear systems states that the net response caused by multiple stimuli upon such a system is equal to the sum of the responses that would have been caused by each individual stimulus.

▶ **Theorem 2.** *If*

$$K_{\text{HH}} > 0,\ K_{\text{HL}} \geq 0,\ K_{\text{LH}} \geq 0,\ K_{\text{LL}} > 0, \tag{11}$$

then the system (8) *exhibits the compensation property.*

Proof. First, let us consider the case when $K_{ii} > 0$, $K_{\text{HL}} = K_{\text{LH}} = 0$, $i \in \{\text{H}, \text{L}\}$ makes the HI- and LO-criticality systems completely decoupled. It is trivial to show that the compensation property holds, since ε_{H} has no effect on the LO-criticality server, and ε_{L} has no effect on the HI-criticality server.

Therefore, we focus on the case $K_{ij} > 0$, $i, j \in \{\text{H}, \text{L}\}$. Since we are dealing with a linear system, we can consider the effect of the disturbances separately, and then use the superposition principle. Without loss of generality, let us consider a positive disturbance $\varepsilon_{\text{H}} > 0$, and an initial condition $S_i(0) = Q_i(0) = \bar{Q}_i$, $i \in \{\text{H}, \text{L}\}$. First, consider the case when $K_{ij} > 0$, $i, j \in \{\text{H}, \text{L}\}$. ε_{H} has the effect of increasing the value of S_{H}, according to (4), without affecting immediately the value of S_{L}, according to (5). If $K_{ij} > 0$, $i, j \in \{\text{H}, \text{L}\}$, an increasing value of S_{H} will make decrease both the tentative budgets, as per (6), and (7). Therefore, in the next step, the tentative budget allocated to the two servers is decreased, with the effect that S_{H} is above the desired budget $\bar{Q}_{\text{H}}$, while S_{L} is below the desired budget $\bar{Q}_{\text{L}}$.

Analogous considerations can be done for the respective negative case. This concludes the proof. ◀

Notice that the compensation property of the control scheme of (8) relates to the transient behavior caused by the occurrence of a disturbance – it does not guarantee that the effect of a disturbance will eventually vanish. Hence a second essential property of the control scheme of (8) is *stability.* If stability is not guaranteed, then it is not possible to preserve the bandwidth, and not even to preserve the target period $\bar{P}$. We want the effect of transient perturbations to be transient, and desire that the actual server budgets tend towards the specified target budget values. Theorem 2 guarantees some properties on the initial transient, but it does not guarantee the convergence of the system behavior towards the desired budget; guaranteeing such convergence is equivalent, in control theory terminology, to requiring stability of the controlled system.

Stability of discrete-time systems, such as the one specified by Expression (8), is guaranteed if and only if the roots of the characteristic polynomial $p(z)$ of (9) are within the unit circle over the complex plane $\mathbb{C}$. That is

$$p(z) = 0 \quad \Rightarrow \quad |z| < 1.$$

Such a condition on the polynomial $p(z)$ can be translated into a condition over the coefficients of the polynomial and, in turn, into a condition over the control gains K_{HH}, K_{HL}, K_{LH}, and K_{LL}. Jury's stability criterion (see, for example, [23, Sec 3.15.2]) offers a necessary and sufficient condition for the stability of a discrete-time system in the form of a set of inequalities which are functions of the coefficients of the characteristic polynomial. By applying Jury's criterion to the polynomial $p(z)$ of (9), one can obtain four analytic conditions on the values of the parameters K_{ij}, $i, j \in \{\text{H}, \text{L}\}$ that guarantee stability. We do not present these conditions here since they are quite lengthy and complex, but point out that they can be computed through a symbolic manipulation tool[4] from the expression of $p(z)$.

[4] We used the Matlab function available at `https://se.mathworks.com/matlabcentral/fileexchange/13904-jury` in combination with the Matlab symbolic toolbox.

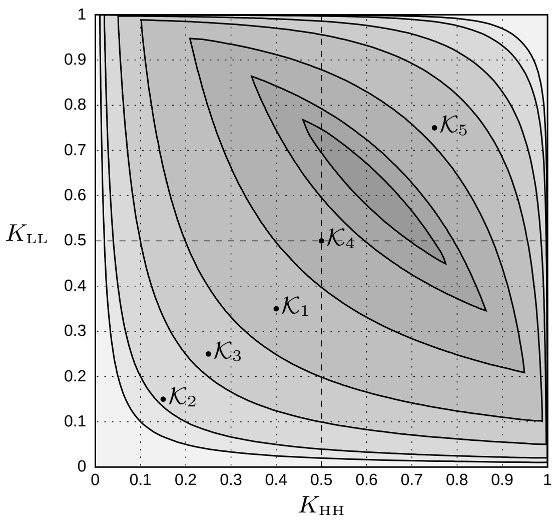


Figure 2 Region of feasible control gains. The illustrated regions correspond to the values of $K_{\rm i} \in \{0, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.35\}$, respectively from the larger region to the smaller one. Black dots represent the gains of the controllers selected for the examples illustrated in Section 6.

The intersection of the inequalities (11) with the stability conditions that are obtained with the Jury criterion describes the region of the feasible controller gains that guarantee both the compensation property and the stability of the controller. Figure 2 shows the contour plot of the stability regions for the parameters $K_{\rm HH}$, $K_{\rm LL}$, for different values of $K_{\rm i} = K_{\rm HL}K_{\rm LH}$ (identified in the figure with different colors). Notice that the region is symmetric with respect to the plane $K_{\rm HH} = K_{\rm LL}$, and that for increasing $K_{\rm i}$ the stability region shrinks. Moreover, for $K_{\rm i} = 0$, the stability region is $0 < K_{\rm HH} < 1$, and $0 < K_{\rm LL} < 1$.

5 Bounding the Resource Supply

Feedback control for real-time resource allocation was initially used for tracking time-varying workloads [28, 7, 1]. Because of the unpredictable nature of variations, the type of offered guarantees are probabilistic or soft real-time. Recently, however, it was shown that a control scheme can provide both a good average behavior *and* hard-real-time guarantees [22]. Such a guarantee was given by computing the "supply bound function" of a periodic resource supply controlled by a feedback loop such as the one described by Expression (8).

Bounds to supply functions are a commonly used abstraction for modeling the minimum amount of a computing resource that is available over time [21, 18, 27, 2]. They have demonstrated their applicability to realistic use cases (e.g., avionics [12]) and there exist measurement-based tools to determine them from actual system execution traces [20]. Let us briefly recall the main concepts. Let $s(t)$ be the indicator function of the availability of a resource:

$$s(t) = \begin{cases} 1 & \text{the resource is available at time } t \\ 0 & \text{the resource is not available at time } t, \end{cases} \tag{12}$$

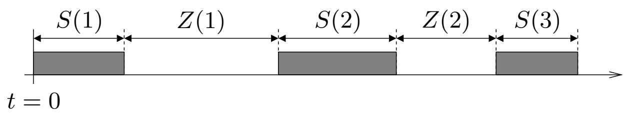


Figure 3 Active intervals interleaved with idle intervals.

Then the *supply bound function* $\mathsf{sbf}(t)$ is such that it is

$$\forall t_0, t, \quad \mathsf{sbf}(t) \leq \int_{t_0}^{t_0+t} s(\tau)\, d\tau. \tag{13}$$

Clearly, from (13), the bound $\mathsf{sbf}(t)$ may not be unique. The aim of much of the research in this area is to find valid bounds $\mathsf{sbf}(t)$ fulfilling (13), which are as high as possible.

In [22], the resource availability schedule is modeled as a sequence of *active* intervals of duration $S(k)$ in which the resource is provided, alternating with intervals of *idle* time of duration $Z(k)$. An example of such a schedule and the corresponding representation by means of the sequences $S(k)$ and $Z(k)$ is illustrated in Figure 3. Such a model offers some advantages over the traditional model by the indicator function of a schedule (as in Eq. 12). In fact, it was proved (Lemma 1 in [22]) that the supply function lower bound $\mathsf{sbf}(t)$ can be written as a function of the sequences of active and idle intervals. Specifically, it was shown that if the resource offered by a schedule is modeled by a sequence of supply intervals of length $\{S(k)\}_{k=1,2,\ldots}$ interleaved by a sequence of idle intervals of length $\{Z(k)\}_{k=1,2,\ldots}$, then the following constitutes a valid supply bound function for this resource availability:

$$\mathsf{sbf}(t) = \min\left\{t - \sigma_Z(n), \sigma_S(n)\right\}, \quad t \in \mathbb{I}_n, n \in \mathbb{N} \tag{14}$$

with the sequence of intervals $\{\mathbb{I}_n\}_{n\in\mathbb{N}}$ defined as

$$\mathbb{I}_n = \begin{cases} \big[0, \sigma_Z(1)\big) & n = 0 \\ \big[\sigma_Z(n) + \sigma_S(n-1), \sigma_Z(n+1) + \sigma_S(n)\big) & n \geq 1 \end{cases} \tag{15}$$

and with

$$\sigma_S(n) = \inf_{n_0} \sum_{k=n_0}^{n_0+n-1} S(k), \quad \sigma_Z(n) \quad = \sup_{n_0} \sum_{k=n_0}^{n_0+n-1} Z(k), \tag{16}$$

properly extended at $n = 0$ with $\sigma_S(0) = \sigma_Z(0) = 0$. The worst-case nature of the bound is condensed in $\sigma_S(n)$ that is the smallest sum of the lengths of n consecutive active intervals (respectively, $\sigma_Z(n)$ is the largest sum of the length of n consecutive idle intervals). Figure 4 illustrates an example of supply function $\mathsf{sbf}(t)$. In the figure, we also draw on top the extent of the intervals $\mathbb{I}_n$.

5.1 Characterizing the Server Supply Functions

One criticism of many mixed-criticality scheduling algorithms that have been proposed is that the LO-criticality workload is severely penalized (e.g., dropped entirely) in the event of the mixed-criticality system behavior exceeding its LO-criticality specifications. As stated earlier, this violates the principle of resilience or robustness, which requires that slight deviations from LO-criticality specifications should result in slight degradation of performance (in mixed-criticality settings, to only the LO-criticality workload). In this section, we discuss

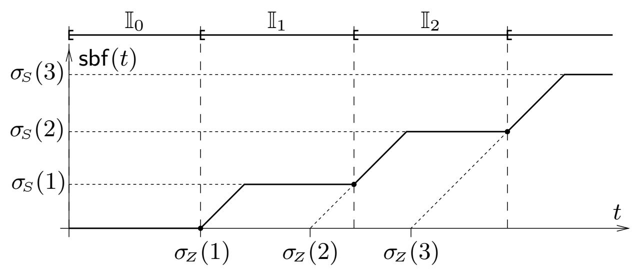


Figure 4 An example of supply bound function $\mathsf{sbf}(t)$ for a resource supply described by sequences $S(k)$ and $Z(k)$ of active and idle intervals.

how an appropriate assignment of values to the gains of the controller $K_{\text{HH}}, K_{\text{HL}}, K_{\text{LH}}$, and K_{LL} enables such resilience by guaranteeing some resource supply to the LO-criticality server.

Our overall approach is inspired by, and based upon, the analysis proposed by Papadopoulos et al. [22]. However, there are several differences in the server requirements/assumptions between our model and the model in [22], that renders the main result (Theorem 1 of [22, page 231]) inapplicable for our purposes.

- First, while disturbances were assumed in [22] to have symmetric bounds, in this paper the LO-criticality server may only experience a a *negative* disturbance, as in (1); equivalently, the LO-criticality server is never allowed to execute beyond the tentative budget that is computed for it by the control strategy.
- Second, in our mixed-criticality run-time algorithm, the servers assigning the computing resource are *coupled* by cross gains K_{HL} and K_{LH}: letting $i, j \in \{\text{H}, \text{L}\}$, it is possible to correct the server budget $S_i(k+1)$ based on any disturbance $\varepsilon_j(k)$. This enables a more prompt compensation.

The following theorem characterizes the relationship between the run-time behavior of the two servers, and enables us to determine the supply function of both the HI-criticality and LO-criticality servers. In the theorem we use the notation $h_{ij}(k)$, $g_{ij}(k)$, and $r_{ij}(k)$ to denote the *impulse*, *step*, and *ramp* responses, respectively, of the system with input $\varepsilon_j(k)$ and output $S_i(k)$, with $i, j \in \{\text{H}, \text{L}\}$ (see Appendix A for the definitions of the considered input signals).

▶ **Theorem 3.** *Consider a pair of* HI*-criticality and* LO*-criticality servers, whose budgets $S_{\text{H}}(k)$ and $S_{\text{L}}(k)$ are subject to disturbances $\varepsilon_{\text{H}}(k)$ and $\varepsilon_{\text{L}}(k)$ respectively, with closed-loop system dynamics as specified by Equation (8). If the disturbances are bounded as specified by (1), then the supply function $\mathsf{sbf}_{\text{H}}(t)$ of the* HI*-criticality server is as specified in Equation (14) with*

$$\begin{aligned} \sigma_S(n) &= n\bar{Q}_{\text{H}} - \bar{\varepsilon}_{\text{H}}\mathcal{N}_{\text{HH}}(n) - \frac{\bar{\varepsilon}_{\text{L}}}{2}\big(\mathcal{I}_{\text{HL}}(n) + \mathcal{N}_{\text{HL}}(n)\big), \\ \sigma_Z(n) &= n\bar{Q}_{\text{L}} + \bar{\varepsilon}_{\text{H}}\mathcal{N}_{\text{LH}}(n) + \frac{\bar{\varepsilon}_{\text{L}}}{2}\big(\mathcal{J}_{\text{LL}}(n) + \mathcal{N}_{\text{LL}}(n)\big), \end{aligned} \tag{17}$$

and the supply function $\mathsf{sbf}_{\text{L}}(t)$ of the LO*-criticality server is as specified in Equation (14) with*

$$\begin{aligned} \sigma_S(n) &= n\bar{Q}_{\text{L}} - \bar{\varepsilon}_{\text{H}}\mathcal{N}_{\text{LH}}(n) - \frac{\bar{\varepsilon}_{\text{L}}}{2}\big(\mathcal{I}_{\text{LL}}(n) + \mathcal{N}_{\text{LL}}(n)\big), \\ \sigma_Z(n) &= n\bar{Q}_{\text{H}} + \bar{\varepsilon}_{\text{H}}\mathcal{N}_{\text{HH}}(n) + \frac{\bar{\varepsilon}_{\text{L}}}{2}\big(\mathcal{J}_{\text{HL}}(n) + \mathcal{N}_{\text{HL}}(n)\big). \end{aligned} \tag{18}$$

The coefficients $\mathcal{N}_{ij}(n)$, $\mathcal{I}_{i\text{L}}(n)$, and $\mathcal{J}_{i\text{L}}(n)$ used in the equations above are set as

$$\begin{aligned}
\mathcal{N}_{ij}(n) &= \sum_{k=0}^{\infty} \left| g_{ij}(k) - g_{ij}(k-n) \right| \\
\mathcal{I}_{i\text{L}}(n) &= \sup_{k} \left\{ r_{i\text{L}}(k) - r_{i\text{L}}(k-n) \right\} \\
\mathcal{J}_{i\text{L}}(n) &= \sup_{k} \left\{ r_{i\text{L}}(k-n) - r_{i\text{L}}(k) \right\}
\end{aligned} \tag{19}$$

with $i, j \in \{\text{H}, \text{L}\}$ corresponding to the LO*-criticality and* HI*-criticality servers, respectively.*

Proof. In the appendix (Appendix A). ◀

Theorem 3 enables us to determine the supply function of both the HI-criticality and LO-criticality servers. In the next section, several design choices for the control gain parameters are illustrated and discussed; it is shown how different desired behaviors can be achieved by an appropriate choice of gain parameters.

6 Evaluation via Simulation

By characterizing the run-time dynamics of both the HI-criticality and the LO-criticality server, Equation (8) and Theorem 3 allow us to estimate the system response to different kinds of transient deviations from the expected "common-case" behavior, as characterized by the LO-criticality WCET estimates. We now explore, via some simulation experiments, (i) the manner in which the choice of gain parameter values influences the precise nature of resilience exhibited by the run-time scheduler, and (ii) how our proposed scheme compares with a simpler alternative strategy that is not based on the application of control-theoretic principles.

6.1 The Influence of Parameter Values

A closed-form solution of the dynamics of the system (8) may be obtained with the Lagrange formula for the solution of a set of linear difference equations (see, e.g., [23, Section 12.3.5, Eq. (12.3-34a)] for a text-book discussion). We consider the following set of parameters that are expressed as $\mathcal{K}_i = \{K_{\text{HH}}, K_{\text{HL}}, K_{\text{LH}}, K_{\text{LL}}\}$:

$$\mathcal{K}_1 = \{0.4, 0.1, 0.1, 0.35\}, \quad \mathcal{K}_2 = \{0.15, 0.1, 0.1, 0.15\}, \quad \mathcal{K}_3 = \{0.25, 0.1, 0.1, 0.25\},$$
$$\mathcal{K}_4 = \{0.5, 0.1, 0.1, 0.5\}, \quad \mathcal{K}_5 = \{0.75, 0.1, 0.1, 0.75\}$$

Notice that all the selected sets of parameters satisfy the stability conditions, and the compensation property conditions, and therefore lie in the region as depicted in Figure 2.

We considered the case of the following target budgets: $\bar{Q}_\text{H} = 10$, $\bar{Q}_\text{L} = 8$, i.e., $\gamma = 0.8$, and $\varepsilon_\text{H} = 1$, $\varepsilon_\text{L} = 1$. The resulting supply functions are presented in Figure 5. One can see that the supply function associated with $\mathcal{K}_1$ is higher than the others.

If keeping with common practice in control theory, we also analyzed the controller response to a *constant* disturbance. Figure 6 shows the effect of the disturbance while varying the values of K_{ij}, $i, j \in \{\text{H}, \text{L}\}$. From Figure 6 we conclude that the best value for the parameters is $\mathcal{K}_1$, since it provides a faster convergence to the target budget, and with negligible oscillations.

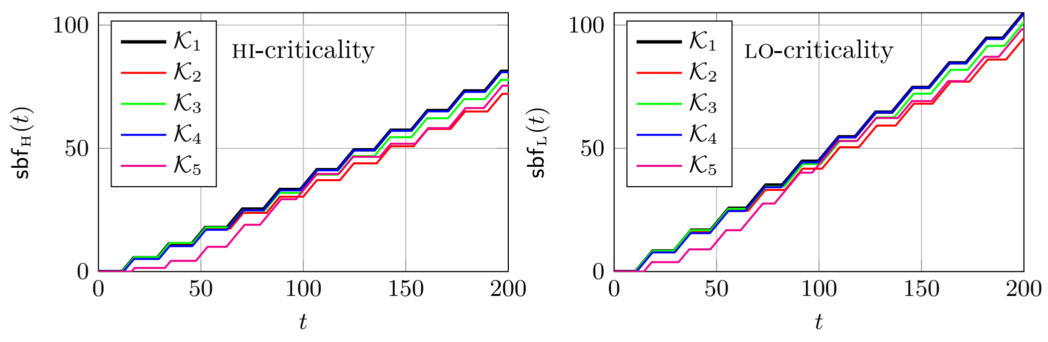


Figure 5 Supply functions for the considered set of control parameters.

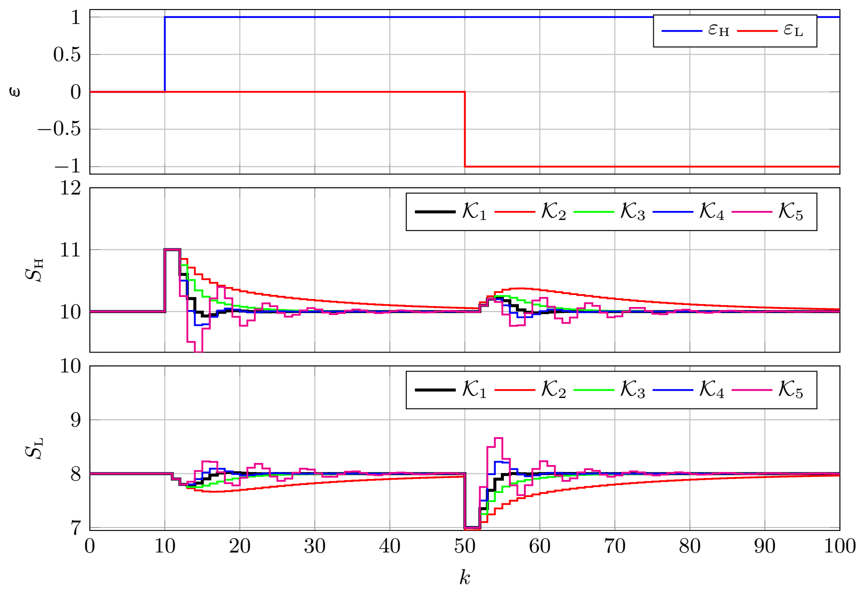


Figure 6 Effect of constant disturbances with various selection of K_{ij}, $i, j \in \{\mathrm{H}, \mathrm{L}\}$.

6.2 Comparison with an Alternative Scheme

We now compare the presented approach with a *Period-Preserving Approach (PPA)*, described next. Based upon the findings described in Section 6.1 above, in these experiments we have selected the parameter values $\mathcal{K}_1$ for AdaptMC.

In the PPA the HI-criticality and LO-criticality servers execute in sequence and periodically, with a fixed period P (equal to the target period for AdaptMC). Within each period, the HI-criticality server executes as much as it needs, allowing for any overrun, and the remaining budget of the period is allocated to the LO-criticality server. Formally, with the introduced notation:

$$S_{\mathrm{H}}(k+1) = Q_{\mathrm{H}}(k) + \varepsilon_{\mathrm{H}}(k)$$
$$S_{\mathrm{L}}(k) = P - S_{\mathrm{H}}(k+1)$$

where P now is a fixed value. PPA represents the simplest and most intuitive way to compensate for non-ideal executions of the HI-criticality server.

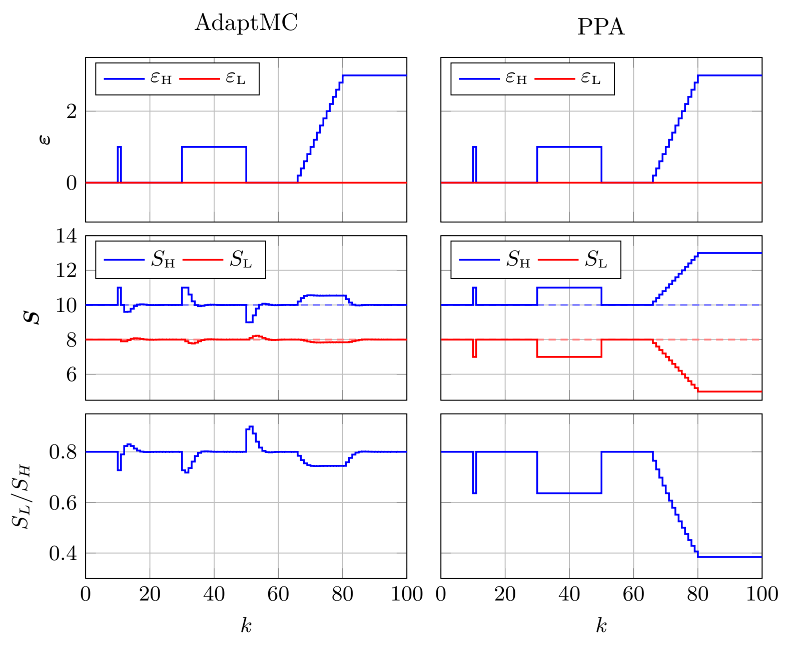


Figure 7 Comparison between AdaptMC and PPA.

In order to present the main differences between AdaptMC and PPA, we consider a scenario in which three types of disturbances occur in the system: impulse, constant, and linearly increasing. (In a well-designed mixed-criticality system, the most common form of deviation from expected behavior should be of the kind best modeled as an *impulse* disturbance – an overload that lasts for just one round and occurs rarely enough that the effect of one such overload will have completely dissipated by the time the next one occurs.)

The system is initialized as $S_{\mathrm{H}}(0) = Q_{\mathrm{H}}(0) = \bar{Q}_{\mathrm{H}} = 10$, and $S_{\mathrm{L}}(0) = Q_{\mathrm{L}}(0) = \bar{Q}_{\mathrm{L}} = 8$, $P = 18$ and no disturbance ε is present. An impulse overrun occurs at round 10, a constant overrun occurs between rounds 30 and 50, and a linearly increasing disturbance begins at round 65, and increases until it becomes of magnitude $\bar{\varepsilon}_{\mathrm{H}}$. Figure 7 summarizes the obtained numerical results. The graphs in the first row show the time evolution of the HI-criticality server overruns: this is the disturbance, and is the same for the AdaptMC and PPA. The graphs in the second row compares the actual time executed by the two servers with the two methods. AdaptMC reacts to the disturbances by trying to preserve the target budgets, and making minor adjustments to the tentative budgets. PPA, on the other hand, favors the overruns of the HI-criticality server, while the execution of the LO-criticality server is severely affected. Finally, the last row of Figure 7 shows the ratio between the bandwidth allocated for the LO-criticality server, i.e., S_{L}/P, and the actual bandwidth allocated for the HI-criticality server, i.e., S_{H}/P. We call this, the *bandwidth ratio*, and it is defined as: $S_{\mathrm{L}}/S_{\mathrm{H}}$. The target bandwidth is $\bar{Q}_{\mathrm{L}}/P = 8/18$, and $\bar{Q}_{\mathrm{H}}/P = 10/18$, i.e., the target bandwidth ratio is $\bar{Q}_{\mathrm{L}}/\bar{Q}_{\mathrm{H}} = 8/10$. The average bandwidth ratio allocated with AdaptMC is much closer to the target bandwidth ratio than with PPA , and even the maximum deviation from the target bandwidth is minimized by AdaptMC thanks to the feedback scheme.

7 Related Work

The key property of the control-theoretic approach to budget control described in this paper is the dynamic manner in which it modifies budgets to deal with different sizes and types of task overruns; this stands in sharp contrast to the approach adopted in most other scheduling schemes for mixed-criticality systems. In these schemes during run-time the system is defined to be in one of two modes of behaviors. In the LO-criticality or "normal" mode all tasks are executing within their LO-criticality WCET estimates and all deadlines (of both HI- and LO-criticality tasks) are being met. As soon as any HI-criticality task executes for more than its LO-criticality WCET estimate then there is a system-wide mode change to the HI-criticality mode. In this new mode the behavior of the system is quite different. The change to the HI-criticality mode occurs even if a single HI-criticality task executes for a miniscule amount more than its LO-criticality WCET estimate or, at the other extreme, if all HI-criticality tasks execute at their HI-criticality WCET estimate. The system responds in the same way: there is no attempt to define behaviors that are commensurate with the magnitude of the overrun (the disturbance or perturbation as defined in this paper).

Following a criticality mode change there are a number of approaches that have been developed to define the degraded behavior of the system in the HI-criticality mode. The most extreme is to just implement the assumptions made during the verification of the system. Here, in the HI-criticality mode, only the HI-criticality tasks are guaranteed; hence all the LO-criticality tasks can be abandoned (aborted). This is clearly an unacceptable approach as no attempt is made to survive the overrun; there is no resilience in the run-time behavior of the system. Forms of resilience that have been developed include:

1. Reduce priorities of the LO-criticality tasks [3], or similar with EDF scheduling [13].
2. Increase the periods and deadlines of LO-criticality jobs [32, 31, 15, 30, 29, 25], called *task stretching*, the *elastic task model* or *multi-rate*.
3. Impose only a weakly-hard constraint on the LO-criticality jobs [9].
4. Decrease the computation times of some or all of the LO-criticality tasks [4], perhaps by utilizing an imprecise mixed-criticality (IMC) model [19, 24] or budget control [10].
5. Abandon LO-criticality work in a disciplined sequence [8, 14, 11, 26, 16].

A flexible scheme utilizing hierarchical scheduling is proposed by Gu et al. [10]. They differentiate between minor violations of LO-criticality execution time which can be dealt with within a component (an internal mode change) and more extensive violations that requires a system-wide external mode change.

By removing entirely the notion of a mode change (and hence a single perhaps quite severe change in system behavior), the approach proposed in this paper results in more gradual and measured responses to rare temporal glitches, such responses being automatically delivered by the developed feedback scheme.

8 Conclusions and Future Work

In this paper we have shown how a control-theoretic approach based upon servers can be used to manage the budgets allocated to dual-criticality workloads. The control strategy developed automatically responds to minor perturbations in the needs of the HI-criticality server with minimum and bounded degradation in the service provided to the LO-criticality server. The controller is defined by four "*gain*" parameters whose values must be constrained in order to ensure stable and appropriate (compensated) control; nevertheless there remains considerable freedom for the designer to tune the behavior of the controller. This has been demonstrated by some simple examples.

This initial study has been limited to just two criticality levels and two servers (one per level). Future work will first look to increase the number of levels supported, and to investigate if there is any benefit to be gained from having more than one HI-criticality server (and more than one LO-criticality server).

References

1 Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 71–80, Austix (TX), USA, dec 2002.

2 Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the* 4^{th} *ACM International Conference on Embedded Software*, pages 95–103, Pisa, Italy, 2004.

3 Sanjoy K. Baruah and Alan Burns. Implementing mixed criticality systems in Ada. In A. Romanovsky, editor, *Proc. of Reliable Software Technologies - Ada-Europe 2011*, pages 174–188. Springer, 2011.

4 Alan Burns and Sanjoy K. Baruah. Towards a more practical model for mixed criticality systems. In *Proc. 1st Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 1–6, 2013.

5 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Computer Surveys*, 50(6):1–37, 2017.

6 Alan Burns and Robert I. Davis. Mixed-criticality systems: A review (10th edition). (Accessed on Apr 8th, 2018), 2018. URL: `http://www-users.cs.york.ac.uk/~burns/review.pdf`.

7 Anton Cervin and Johan Eker. Feedback scheduling of control tasks. In *Proceedings of the 39th IEEE Conference on Decision and Control*, pages 4871–4876, 2000. `doi:10.1109/CDC.2001.914702`.

8 Tom Fleming and Alan Burns. Incorporating the notion of importance into mixed criticality systems. In Liliana Cucu-Grosjean and Robert I. Davis, editors, *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 33–38, 2014.

9 Oliver Gettings, Sophie Quinton, and Robert I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proc. 23rd International Conference on Real-Time Networks and Systems (RTNS 2015)*, pages 237–246, 2015.

10 Xiaozhe Gu and Arvind Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 47–56. IEEE, 2016.

11 Xiaozhe Gu, Arvind Easwaran, Kieu-My Phan, and Insik Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *Proc. 27th ECRTS*, pages 13–24. IEEE, 2015.

12 Ana Guasque, Patricia Balbastre, and Alfons Crespo. Real-time hierarchical systems with arbitrary scheduling at global level. *Journal of Systems and Software*, 119:70–86, 2016.

13 Pengcheng Huang, Georgia Giannopoulou, Nikolay Stoimenov, and Lothar Thiele. Service adaptions for mixed-criticality systems. In *Proc. 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Singapore, 2014.

14 Pengcheng Huang, Pratyush Kumar, Nikolay Stoimenov, and Lothar Thiele. Interference constraint graph – a new specification for mixed-criticality systems. In *Proc. 18th Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2013.

15 Mathieu Jan, Lilia Zaourar, and Maurice Pitel. Maximizing the execution rate of low criticality tasks in mixed criticality system. In *Proc. 1st WMC, RTSS*, pages 43–48, 2013.

16 Jaewoo Lee, Hoon Sung Chwa, Linh T. X. Phan, Insik Shin, and Insup Lee. MC-ADAPT: Adaptive task dropping in mixed-criticality scheduling. *ACM Trans. Embed. Comput. Syst.*, 16:163:1–163:21, 2017.

17 Alberto Leva and Martina Maggio. Feedback process scheduling with simple discrete-time control structures. *IET control theory & applications*, 4(11):2331–2342, 2010.

18 Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Proceedings of the 15-th Euromicro Conference on Real-Time Systems*, pages 151–158, Porto, Portugal, 2003.

19 D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees. In *Proc. IEEE RTSS*, pages 35–46, 2016.

20 Martina Maggio, Juri Lelli, and Enrico Bini. A tool for measuring supply functions of execution platforms. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 39–48. IEEE, 2016.

21 Aloysius K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the* 7^{th} *IEEE Real-Time Technology and Applications Symposium*, pages 75–84, Taipei, Taiwan, 2001.

22 Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Systems*, 51(3):221–246, 2015.

23 Paraskevas N. Paraskevopoulos. *Modern Control Engineering*. Automation and Control Engineering. Taylor & Francis, 2001.

24 Risat Mahmud Pathan. Improving the quality-of-service for scheduling mixed-criticality systems on multiprocessors. In Marko Bertogna, editor, *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

25 Saravanan Ramanathan, Arvind Easwaran, and Hyeonjoong Cho. Multi-rate fluid scheduling of mixed-criticality systems on multiprocessors. *Real-Time Systems*, Online First, 2017.

26 Jiankang Ren and Linh Thi Xuan Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *Proc. 27th ECRTS*, pages 25–36. IEEE, 2015.

27 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24^{th} *Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, dec 2003.

28 John A. Stankovic, Chenyang Lu, Sang H. Son, and Gang Tao. The case for feedback control in real-time scheduling. In *Proceedings of the Euromicro Conference on Real-Time*, York, U.K., jun 1999.

29 Hang Su, Peng Deng, Dakai Zhu, and Qi Zhu. Fixed-priority dual-rate mixed-criticality systems: Schedulability analysis and performance optimization. In *Proc. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 59–68. IEEE, 2016.

30 Hang Su, Nan Guan, and Dakai Zhu. Service guarantee exploration for mixed-criticality systems. In *Proc. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2014.

31 Hang Su and Dakai Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, pages 147–152, 2013.

32 Hang Su, Dakai Zhu, and Daniel Mosse. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Proc. RTCSA*, 2013.

33 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

A Proof of Theorem 3

Before entering the details of the proofs, we remind that a linear time-invariant (LTI) system can be uniquely characterized by its *impulse response* $h(k)$ that is the output $y(k)$ when the system is stimulated with an impulsive input $u(k)$

$$u(k) = \begin{cases} 1 & k = 0 \\ 0 & \text{otherwise.} \end{cases}$$

In next lemmas, we are also using the *step response*

$$g(k) = \sum_{i=0}^{k} h(k), \tag{20}$$

and the *ramp response*

$$r(k) = \sum_{i=0}^{k} g(i) \tag{21}$$

of a LTI system.

Thanks to the linear and time-invariance of the system, the output $y(k)$ to any input $u(k)$ is given by the *convolution* of the impulse response $h(k)$ and the input $u(k)$, that is

$$y(k) = h(k) \otimes u(k) = \sum_{i=0}^{k} u(i)h(k-i).$$

With these basic notions recalled, next we state a technical lemma that bounds the output $y(k)$ of a LTI system when the input $u(k)$ belongs to a bounded interval $[\tilde{u} - \bar{\varepsilon}, \tilde{u} + \bar{\varepsilon}]$.

▶ **Lemma 1.** *Given an asymptotically stable discrete-time LTI system with impulse response $h(k)$, step response $g(k)$, input $u(k)$, and output*

$$y(k) = h(k) \otimes u(k).$$

If the input $u(k)$ is bounded as follows

$$u(k) = \tilde{u} + \varepsilon(k), \qquad \tilde{u} \in \mathbb{R}, \ -\bar{\varepsilon} \le \varepsilon(k) \le \bar{\varepsilon},$$

then, the output $y(k)$ is bounded by

$$|\tilde{u}| \inf_k \{\operatorname{sign}(\tilde{u}) g(k)\} - \bar{\varepsilon} \, \|h\|_1 \le y(k) \le |\tilde{u}| \sup_k \{\operatorname{sign}(\tilde{u}) g(k)\} + \bar{\varepsilon} \, \|h\|_1, \tag{22}$$

with the ℓ_1-norm of a signal defined as

$$\|h\|_1 = \sum_{k=0}^{\infty} |h(k)|.$$

Proof. By definition of $y(k)$ as convolution of the impulse response $h(k)$ with the input signal $u(k)$, it follows

$$\begin{aligned}
y(k) &= \sum_{i=0}^{k} u(i)h(k-i) = \sum_{i=0}^{k} (\tilde{u} + \varepsilon(i))h(k-i) \\
&= \tilde{u} \sum_{i=0}^{k} h(k-i) + \sum_{i=0}^{k} \varepsilon(i)h(k-i) \\
&= \tilde{u}\, g(k) + \sum_{i=0}^{k} \varepsilon(i)h(k-i) \\
&\leq |\tilde{u}| \sup_k \{\operatorname{sign}(\tilde{u}) g(k)\} + \bar{\varepsilon}\, \|h(k)\|_1
\end{aligned}$$

with

$$\|h(k)\|_1 = \sum_{k=0}^{\infty} |h(k)|.$$

Analogously

$$y(k) \geq |\tilde{u}| \inf_k \{\operatorname{sign}(\tilde{u}) g(k)\} - \bar{\varepsilon}\, \|h(k)\|_1,$$

which concludes the proof. ◀

The next Corollary determines the upper and lower bounds to the sum of n consecutive outputs, by exploiting Lemma 1.

▶ **Corollary 1.** *Given an asymptotically stable discrete-time LTI system, if the input $u(k)$ bounded as follows*

$$u(k) = \tilde{u} + \varepsilon(k), \qquad \tilde{u} \in \mathbb{R},\ -\bar{\varepsilon} \leq \varepsilon(k) \leq \bar{\varepsilon}.$$

Then, the sum of n consecutive outputs is bounded by

$$-\big(|\tilde{u}|\, \mathcal{I}(n) + \bar{\varepsilon}\, \mathcal{N}(n)\big) \leq \sum_{k=n_0}^{n_0+n-1} y(k) \leq |\tilde{u}|\, \mathcal{J}(n) + \bar{\varepsilon}\, \mathcal{N}(n), \tag{23}$$

with

$$\mathcal{N}(n) = \sum_{k=0}^{\infty} |g(k) - g(k-n)|, \tag{24}$$

$$\mathcal{I}(n) = \sup_k \{-\operatorname{sign}(\tilde{u})(r(k) - r(k-n))\} \tag{25}$$

$$\mathcal{J}(n) = \sup_k \{\operatorname{sign}(\tilde{u})(r(k) - r(k-n))\} \tag{26}$$

and $g(k)$ and $r(k)$ being the step and ramp response, respectively.

Proof. The output $y(k)$ of a LTI system is the convolution of the impulse response $h(g)$ and the input $u(k)$

$$y(k) = h(k) \otimes u(k).$$

Because of the linearity of the convolution, the sum of n consecutive output is

$$\sum_{i=k}^{k+n-1} y(i) = \left(\sum_{i=k}^{k+n-1} h(i)\right) \otimes u(k) = \big(g(k) - g(k-n)\big) \otimes u(k).$$

Finally, by applying Equation (22) of Lemma 1, Equation (23) of the Corollary follows. ◀

Proof of Theorem 3. Let us first determine the supply function $\mathsf{sbf}_{\mathrm{H}}(t)$ of the HI-criticality server. We aim at modeling the resource supplied to the HI-criticality server as a sequence of active intervals of lengths $S(k)$, interleaved by a sequence of idle intervals of lengths $Z(k)$ that corresponds to the schedule of the LO-criticality server. Formally,

$$S(k) = S_{\mathrm{H}}(k), \qquad Z(k) = S_{\mathrm{L}}(k). \tag{27}$$

In fact, by doing so, Lemma 1 of [22] can give us the supply function of (14) through the proper value of $\sigma_S(n)$ and $\sigma_Z(n)$, as defined in (16).

First of all, the system of (8) that determines the dynamics of $S_{\mathrm{H}}(k)$ is linear. Hence, by the superposition principle the output $S_{\mathrm{H}}(k)$ is equal to the sum of three components:

1. the output $\bar{Q}_{\mathrm{H}}$ when $\varepsilon_{\mathrm{H}}(k) = 0$ and $\varepsilon_{\mathrm{L}}(k) = 0$,
2. the output $y_{\mathrm{HH}}(k)$ when $\bar{Q}_{\mathrm{H}} = 0$ and $\varepsilon_{\mathrm{L}}(k) = 0$, and
3. the output $y_{\mathrm{HL}}(k)$ when $\bar{Q}_{\mathrm{H}} = 0$ and $\varepsilon_{\mathrm{H}}(k) = 0$,

that is

$$S_{\mathrm{H}}(k) = \bar{Q}_{\mathrm{H}} + \underbrace{h_{\mathrm{HH}}(k) \otimes \varepsilon_{\mathrm{H}}(k)}_{y_{\mathrm{HH}}(k)} + \underbrace{h_{\mathrm{HL}}(k) \otimes \varepsilon_{\mathrm{L}}(k)}_{y_{\mathrm{HL}}(k)} \tag{28}$$

and $h_{Hi}(k)$ is the response of $S_{\mathrm{H}}(k)$ to an impulse on the input $\varepsilon_i(k)$, with $i \in \{L, H\}$.

Let us now compute $\sigma_S(n)$ that is, from (16), a lower bound to the sum of the length of n consecutive budgets $S_{\mathrm{H}}(k)$

$$\sigma_S(n) = \inf_{n_0} \sum_{k=n_0}^{n_0+n-1} S(k) = \inf_{n_0} \sum_{k=n_0}^{n_0+n-1} S_{\mathrm{H}}(k) = n\bar{Q}_{\mathrm{H}} + \inf_{n_0} \sum_{k=n_0}^{n_0+n-1} \big(y_{\mathrm{HH}}(k) + y_{\mathrm{HL}}(k)\big). \tag{29}$$

To bound the sum of n consecutive values of $y_{\mathrm{HH}}(k)$ and $y_{\mathrm{HL}}(k)$, we can invoke Corollary 1. Let us start with

$$y_{\mathrm{HH}}(k) = h_{\mathrm{HH}}(k) \otimes \varepsilon_{\mathrm{H}}(k).$$

From the hypothesis of (1), $\varepsilon_{\mathrm{H}}(k)$ is bounded by

$$-\bar{\varepsilon}_{\mathrm{H}} \le \varepsilon_{\mathrm{H}}(k) \le \bar{\varepsilon}_{\mathrm{H}}$$

and then Eq. (23) of Corollary (1) states that

$$-\bar{\varepsilon}_{\mathrm{H}} \mathcal{N}_{\mathrm{HH}}(n) \le \sum_{k=n_0}^{n_0+n-1} y_{\mathrm{HH}}(k),$$

with $\mathcal{N}_{\mathrm{HH}}(n)$ as in (19). Similarly, from the asymmetric bound to $\varepsilon_{\mathrm{L}}(k)$ of (1), from (23) it follows that

$$-\frac{\bar{\varepsilon}_{\mathrm{L}}}{2}(\mathcal{I}_{\mathrm{HL}}(n) + \mathcal{N}_{\mathrm{HL}}(n)) \le \sum_{k=n_0}^{n_0+n-1} y_{\mathrm{HL}}(k),$$

from which the expression of $\sigma_S(n)$ of (17) follows.

The expression of $\sigma_Z(n)$ of (17) can be found by following similar steps:

1. by setting the sequence of idle intervals $Z(k)$ equal to the sequence of the LO-criticality budgets $S_{\mathrm{L}}(k)$, as in (27);

2. by writing the sequence $S_{\mathrm{L}}(k)$ as the sum of $\bar{Q}_{\mathrm{L}}$ and the sequences $y_{\mathrm{LH}}(k)$ and $y_{\mathrm{LL}}(k)$ that corresponds to the responses to the disturbances $\varepsilon_{\mathrm{L}}(k)$ and $\varepsilon_{\mathrm{L}}(k)$ on $S_{\mathrm{L}}(k)$ (similarly as in (28); and
3. by exploiting Corollary 1 to bound $y_{\mathrm{LH}}(k)$ and $y_{\mathrm{LL}}(k)$.

   The expressions of $\sigma_S(n)$ and $\sigma_Z(n)$ give the expression of the $\mathsf{sbf}_{\mathrm{H}}(t)$.
   Analogously, by setting

$$S(k) = S_{\mathrm{L}}(k), \qquad Z(k) = S_{\mathrm{H}}(k),$$

and following the same steps illustrated above, it is possible to determine the proper values of $\sigma_S(n)$ and $\sigma_Z(n)$ of (18) and then the supply function $\mathsf{sbf}_{\mathrm{L}}(t)$ of the LO-criticality server. This concludes the proof. ◀

Verifying Weakly-Hard Real-Time Properties of Traffic Streams in Switched Networks

Leonie Ahrendts
Institute for Network and Computer Engineering, TU Braunschweig, Braunschweig, Germany
ahrendts@ida.ing.tu-bs.de

Sophie Quinton
Inria Grenoble Rhône-Alpes, Montbonnot, France
sophie.quinton@inria.fr

Thomas Boroske
Institute for Network and Computer Engineering, TU Braunschweig, Braunschweig, Germany
thomasb@ida.ing.tu-bs.de

Rolf Ernst
Institute for Network and Computer Engineering, TU Braunschweig, Braunschweig, Germany
ernst@ida.ing.tu-bs.de

Abstract

In this paper, we introduce the first verification method which is able to provide weakly-hard real-time guarantees for tasks and task chains in systems with multiple resources under partitioned scheduling with fixed priorities. Existing weakly-hard real-time verification techniques are restricted today to systems with a single resource. A weakly-hard real-time guarantee specifies an upper bound on the maximum number m of deadline misses of a task in a sequence of k consecutive executions. Such a guarantee is useful if a task can experience a bounded number of deadline misses without impacting the system mission. We present our verification method in the context of switched networks with traffic streams between nodes, and demonstrate its practical applicability in an automotive case study.

2012 ACM Subject Classification Computer systems organization → Embedded systems, Computer systems organization → Real-time systems, Software and its engineering → Formal software verification, Networks → Network performance analysis

Keywords and phrases embedded and cyber-physical systems, weakly-hard real-time systems and networks, timing analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.15

Funding This work has received funding from the German Research Foundation under the contract number 168/30-2.

1 Introduction

Modern embedded systems often have a distributed hardware platform, where the individual processing resources are linked by data buses or switched networks. A software application, which is mapped to such a platform, consists of a set of communicating tasks and has often to provide results within a limited response time. Timely communication between sender and receiver tasks is therefore a critical aspect in design and verification. In this paper, we concentrate on the timing behavior of traffic streams in switched networks like Switched Ethernet. By traffic stream we understand an infinite sequence of data transmissions between a sender and a receiver node of the network.

If the classical hard real-time paradigm is applied to a traffic stream, then the duration of a data transmission over the network must not violate a given end-to-end deadline. However, with increasing functionality and growing bandwidth demand of data transmission in modern embedded systems in the automotive or industrial domain, it becomes more and more difficult to fulfill the end-to-end deadlines of all traffic streams in unfavorable scheduling scenarios. A promising option is the shift to the weakly-hard real-time paradigm [1] which relaxes these timing requirements. Here a traffic stream is feasible from a timing perspective, if it does not exceed a certain budget of end-to-end deadline misses. For instance, a traffic stream may not miss more than m end-to-end deadlines in any k consecutive transmissions. The traffic stream is said to be (m, k)-constrained.

The practical justification of weakly-hard real-time paradigm in the context of communication builds on the observed robustness of many real-time software systems. In the field of image processing, a late transmission may result in a skipped frame. Given that the number and distribution of frame skips is appropriately bounded, it will not be noticeable to the human eye. In the field of control, an end-to-end deadline miss may cause the calculation of the control law to fail at time instant k so that no new control input is sent to the actuator at this instant. Several works could show that under given (m, k)-constraints the required control performance could be maintained [15] [9] [8] . Blind et al. [2] could show stability in the classical sense of Lyapunov for a networked control system, where the network is unreliable in the (m,k)-sense.

So far, verification techniques have been developed which allow to derive (m, k)-guarantees for tasks which are executed on a system with a *single* service-providing resource. A switched network, however, comprises *several* service-providing resources as detailed in Section 2. In this paper, we therefore provide a compositional verification method which is able to provide (m,k)-guarantees for multi-resource problems. The main challenge in extending an existing (m,k)-verification method to the multi-resource setting is to deal with inter-resource dependencies. Our approach builds on both

1. *Compositional Performance Analysis (CPA).* CPA [11] is a compositional framework to verify classical hard real-time properties, e.g., worst case response times. It deals with inter-resource dependencies by the formulation of a fixed-point problem.
2. *Typical Worst Case Analysis (TWCA)* TWCA [21] is one of the existing (m,k)-verification techniques for single resource systems.

We adapt and extend CPA and TWCA, calling the resulting procedure *TypicalCPA*. The paper is structured as follows. We begin by defining our system model, and then introduce the CPA approach. We continue by explaining the basic principle of TWCA, and reason how CPA and TWCA can be coupled. Finally, we perform and discuss experiments. An overview of related work is given before the conclusion.

2 Network Model

The system model represents a real-time network setting with unicast, multicast and broadcast streams and is depicted in Figure 1. The scope of the model includes, for instance, Switched Ethernet but is not limited to it. The main components of the network model are switches and nodes. A pair of nodes may communicate by sending frames over the network which are forwarded by the switches using appropriate output ports. The service of output ports for frame transmission is scarce and has to be arbitrated according to a static priority non-preemptive (SPNP) scheduling policy. The output ports therefore represent the service-providing resources R_k in the system [6].

An infinite sequence of frames between a source node and 1 [a subset of resp. all] destination node(s) is called a unicast [multicast resp. broadcast] stream. A unicast [multicast resp. broadcast] stream s_i is modeled as a linear [forked] chain of N tasks, where each task represents a hop in the route and is mapped to the output port of the respective switch. We call the set of N tasks contained in the stream s_i $\mathcal{T}_{s_i} = \{\tau_{i,1}, \tau_{i,2}, \ldots \tau_{i,N}\}$ and define the respective precedence constraints, e.g. for a unicast stream as $\tau_{i,1} \prec \tau_{i,2} \prec \ldots \prec \tau_{i,N}$. The first task in the stream s_i, is activated by an external event source. All successor tasks are activated by the termination events of their respective predecessor task in the chain. Each task $\tau_{i,j}$ in stream s_i has a non-unique priority p_i. The best case execution time (BCET) resp. worst case execution time (WCET) of task $\tau_{i,j}$, denoted as $C^-_{i,j}$ resp. $C^+_{i,j}$, represents the minimum resp. maximum frame delay in the switch plus the constant wire transmission time, and is independent of other traffic in the network. Dynamic delays resulting from contention at the switch output ports are considered in the response time computation of tasks. The maximum response time of a task $\tau_{i,j}$ is constrained by the relative deadline $d_{i,j}$, while the maximum network traversal time w.r.t. a stream s_i should not exceed the end-to-end deadline $D_i = \sum_j d_{i,j}$.

We describe the occurrence of activation events over time w.r.t. a task $\tau_{i,j}$ by the concept of event flows as well as by minimum and maximum event models.

▶ **Definition 1** (Event flow)**.** An event flow $e_{i,j}(t)$ is a function which returns the number of events which activate task $\tau_{i,j}$ within the time interval $[0, t)$ in a given execution run.

▶ **Definition 2** (Event model)**.** The minimum and maximum event models $\eta^-_{i,j}(\Delta t)$ and $\eta^+_{i,j}(\Delta t)$ indicate a lower and upper bound, respectively, on the number of activation events for task $\tau_{i,j}$ in any time interval $[t, t + \Delta t)$. Any event flow $e_{i,j}(t)$ of task $\tau_{i,j}$ is therefore constrained by

$$\forall t_1, t_2 : t_1 \leq t_2 : \eta^-_{i,j}(t_2 - t_1) \leq e_{i,j}(t_2) - e_{i,j}(t_1) \leq \eta^+_{i,j}(t_2 - t_1).$$

If convenient, we also use the pseudo-inverses of event models, i.e., the event distance functions. The event distance function $\delta^-_{i,j}(n)$ [$\delta^+_{i,j}(n)$] is the pseudo-inverse of event model $\eta^+_{i,j}(\Delta t)$ [$\eta^-_{i,j}(\Delta t)$].

▶ **Definition 3** (Event distance functions)**.** The minimum and maximum distance functions $\delta^-_{i,j}(n)$ and $\delta^+_{i,j}(n)$ indicate a lower and upper bound, respectively, on the temporal distance between the first and the last event of a sequence of n activation events for task $\tau_{i,j}$. For the special case $n \in \{0, 1\}$, the definition $\delta^-_{i,j}(n) = \delta^+_{i,j}(n) = 0$ applies.

3 Compositional Performance Analysis

CPA [11] is a verification framework which derives lower and upper bounds on the timing properties of distributed real-time software systems with partitioned scheduling. Computed timing properties include in particular the best case response times (BCRTs) and worst case response times (WCRTs) of tasks. CPA is implemented in Python as pyCPA [4], the basic libraries of pyCPA are available on-line [5]. The CPA method breaks the verification problem down into a set of local, i.e. resource-related, analysis problems. A subsequent analysis step then relates the local verification problems such that inter-resource dependencies are taken into account and a global fixed point problem is formulated.

▶ **Definition 4** (Attributes local & global)**.** The attribute «local» refers to parameters, properties etc. of a specific resource R_k and the associated (mapped) task set $\mathcal{T}_{R_k}$.
The attribute «global» refers, on the contrary, to parameters, properties etc. of the processing platform $\mathcal{P} = \bigcup_k R_k$ and the entire task set $\mathcal{T} = \bigcup_k \mathcal{T}_{R_k}$.

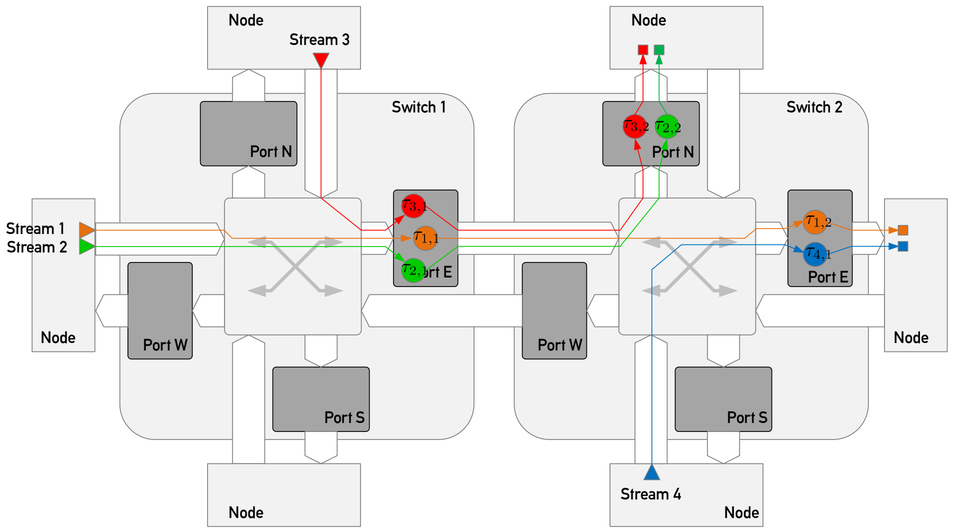


Figure 1 Network model. *The figure illustrates a network with six nodes and two switches. The output ports of a switch are named after the points of the compass. Four exemplary unicast streams are represented.*

3.1 Local Analysis

The local analysis focuses on the isolated resource R_k and derives the timing properties of the associated task set $\mathcal{T}_{R_k}$. The analysis objective is in particular to compute (a) the BCRT and WCRT for each task $\tau_{i,j} \in \mathcal{T}_{R_k}$, and (b) the output event model of each task $\tau_{i,j} \in \mathcal{T}_{R_k}$.

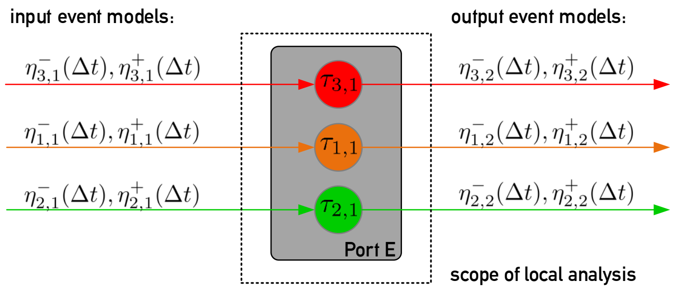

input event models:
$\eta^-_{3,1}(\Delta t), \eta^+_{3,1}(\Delta t)$
$\eta^-_{1,1}(\Delta t), \eta^+_{1,1}(\Delta t)$
$\eta^-_{2,1}(\Delta t), \eta^+_{2,1}(\Delta t)$
$\tau_{3,1}$
$\tau_{1,1}$
$\tau_{2,1}$
Port E
output event models:
$\eta^-_{3,2}(\Delta t), \eta^+_{3,2}(\Delta t)$
$\eta^-_{1,2}(\Delta t), \eta^+_{1,2}(\Delta t)$
$\eta^-_{2,2}(\Delta t), \eta^+_{2,2}(\Delta t)$
scope of local analysis

Figure 2 Scope and interface of the local CPA. *The figure shows as an example the output port E of switch 1 with mapped tasks.*

3.1.1 Computation of Response Times

In the following, we very briefly sketch the response time analysis for a task $\tau_{i,j}$ which is mapped to an SPNP-scheduled resource R_k. For a detailed presentation, please refer to [7]. To find the WCRT of task $\tau_{i,j}$, a scheduling scenario has to be known which induces the longest response time of task $\tau_{i,j}$. This worst case scenario is often called the *maximum level-$\tau_{i,j}$ busy period*. It is known to start if $\tau_{i,j} \cup hsp(\tau_{i,j})$[1]are activated synchronously and a

[1] We use $hsp(\tau_{i,j})$ to denote the set of tasks which have *higher or same priority* than task $\tau_{i,j} \in \mathcal{T}_{R_k}$ and are mapped to the same resource R_k. Likewise we write $lp(\tau_{i,j})$ to denote the set of tasks which

task in $lp(\tau_{i,j})$, which has just been activated before, causes the maximum blocking delay [3]. It closes as soon as the resource becomes idle w.r.t. $\tau_{i,j}$ and $hsp(\tau_{i,j})$-tasks. The processing behavior of task $\tau_{i,j}$ within the maximum level-$\tau_{i,j}$ busy period can be described by the so called *multiple event busy times* $B^+_{i,j}(q)$.

▶ **Definition 5.** The maximum q-event busy time $B^+_{i,j}(q)$ indicates the processing time of q consecutive activation events of task $\tau_{i,j}$ within the maximum level-$\tau_{i,j}$ busy period. $B^+_{i,j}(q)$ always starts with the beginning of the maximum level-$\tau_{i,j}$ busy period [17].

The busy times $B^+_{i,j}(q)$ depend on the input event models and WCETs of the tasks $\mathcal{T}_{R_k}$. It has been shown that the WCRT $R^+_{i,j}$ of task $\tau_{i,j}$ is among its response times in the maximum level-$\tau_{i,j}$ busy period, such that we can write

$$R^+_{i,j} = \max_{1 \le q \le K_{i,j}} \left\{ B^+_{i,j}(q) - \delta^-_{i,j}(q) \right\} \tag{1}$$

where $K_{i,j}$ is the maximum number of jobs of task $\tau_{i,j}$ contained in the maximum level-$\tau_{i,j}$ busy period. The BCRT of task $\tau_{i,j}$ can be approximated by its BCET $R^-_{i,j} = C^-_{i,j}$.

3.1.2 Computation of Output Event Distance Functions and Output Event Models

The local analysis problems are linked because precedence relations extend over tasks on different resources as illustrated in Figure 1. According to the synchronous task chain semantics, a termination event of a task $\tau_{i,j}$ is interpreted as an activation event by the successor task $\tau_{i,j+1}$. This interaction between tasks $\tau_{i,j}$ and $\tau_{i,j+1}$ can be quantified by the distance functions $\delta^+_{i,j+1}(n)$ resp. $\delta^-_{i,j+1}(n)$ indicating the maximum resp. minimum number of distance between any n consecutive termination events of task $\tau_{i,j}$ or, equivalently, activation events of task $\tau_{i,j+1}$. Firstly, let us present safe, easy-to-interpret bounds for the distance functions with $n \ge 2$ using the *jitter method* [16]

$$\delta^-_{i,j+1}(n) \ge \max \left\{ (n-1) \cdot C^-_{i,j}, \delta^-_{i,j}(n) - J^+_{i,j} \right\} \tag{2}$$

$$\delta^+_{i,j+1}(n) \le \delta^+_{i,j}(n) + J^+_{i,j} \tag{3}$$

Eq. 2 expresses that, in the worst case, n termination events at the output of task $\tau_{i,j}$ are closer by the maximum response time jitter $J^+_{i,j} = R^+_{i,j} - R^-_{i,j}$ than n activation events at the input of the same task. Also, the density of activation events increases with every stage of the task chain due to the accumulation of response jitter. Eq. 3 describes that, in the best case, the distance of n termination events grows with every stage of a task chain by the jitter $J^+_{i,j}$. Secondly, we introduce more accurate but less intuitive bounds which have been derived in [18] (*busy window method*)

$$\delta^-_{i,j+1}(n) \ge \max\{B^-_{i,j}(n-1), \min_{1 \le q \le q^+_{i,j}} \left\{ \delta^-_{i,j}(n+q-1) - B^+_{i,j}(q) \right\} + B^-_{i,j}(1)\} \tag{4}$$

$$\delta^+_{i,j+1}(n) \le \max_{1 \le q \le q^+_i} \left\{ \delta^+_{i,j}(n-q+1) + B^+_{i,j}(q) \right\} - B^-_{i,j}(1)\}. \tag{5}$$

According to the rules of network calculus [12], the event distance function $\delta^-_{i,j+1}(n)$ resp. $\delta^+_{i,j+1}(n)$ can even be more improved in accuracy if replaced by its superadditive closure

have *lower priority* than task $\tau_{i,j} \in \mathcal{T}_{R_k}$ and are mapped to the same resource R_k.

$\bar{\delta}^-_{i,j+1}(n)$ resp. subadditive closure $\bar{\delta}^+_{i,j+1}(n)$. In the following, we continue to write $\delta^-_{i,j+1}(n)$ resp. $\delta^+_{i,j+1}(n)$ (without bar) stating explicitly when we make use of the superadditivity property ($\delta^-_{i,j+1}(m+n) \geq \delta^-_{i,j+1}(m) + \delta^-_{i,j+1}(n)$) or subadditivity property ($\delta^+_{i,j+1}(m+n) \leq \delta^+_{i,j+1}(m) + \delta^+_{i,j+1}(n)$). Note again that the output event models $\eta^+_{i,j+1}(\Delta t)$, $\eta^-_{i,j+1}(\Delta t)$ can be obtained from the output event distance functions by pseudo-inversion.

It is desirable for efficiency reasons to have a finite representation of event distance functions, meaning that it is possible to construct the event distance functions for every n on the basis of a limited number of l known points. This can be achieved by approximating $\delta^-_{i,j+1}(n)$, $\delta^+_{i,j+1}(n)$ by bounds with a repetitive behavior. The approximation is very acceptable with regard to accuracy, if the repetition period is chosen large enough. In the particular context of this paper, repetitive bounds restrict the value range that needs to be processed by the algorithm given in Theorem 23. We concentrate in the following on $\delta^-(n)$ and its pseudo-inverse $\eta^+(\Delta)$, but analogous rules can be applied to $\delta^+(n)$ and $\eta^-(\Delta)$.

▶ **Lemma 6** (Repetitive extension of an event distance function). *Given the superadditive event distance function $\delta^-(n)$ for $1 \leq n \leq l$, an l-repetitive extension $\hat{\delta}^-(n)$ is defined by*

$$\hat{\delta}^-(n) = \begin{cases} 0 & \textit{for } 0 \leq n \leq 1 \\ \lfloor \frac{n-2}{l} \rfloor \cdot \delta^-(l) + \delta^-(n - \lfloor \frac{n-2}{l} \rfloor \cdot l) & \textit{for } n \geq 2. \end{cases}.$$

The l-repetitive extension $\hat{\delta}^-(n)$ is a lower bound for $\delta^-(n)$, s.t. $\forall n : \hat{\delta}^-(n) \leq \delta^-(n)$.

Proof. We have $\hat{\delta}^-(n) = \delta^-(n) = 0$ for $n \in \{0, 1\}$, and $\hat{\delta}^-(n) = \delta^-(n)$ for $2 \leq n \leq l+2$. For $n > l+2$, we make use of the superadditivity property $\delta^-(n_1) + \delta^-(n_2) \leq \delta^-(n_1+n_2)$ and set $x = \lfloor \frac{n-2}{l} \rfloor$: $\hat{\delta}^-(n) = x \cdot \delta^-(l) + \delta^-(n - x \cdot l) \leq \delta^-(x \cdot l) + \delta^-(n - x \cdot l) \leq \delta^-(n)$. ◀

▶ **Lemma 7** (Repetitive extension of an event model). *Given the subadditive event model function $\eta^+(\Delta t)$, a T-repetitive extension $\hat{\eta}^+_l(\Delta t)$ is defined by*

$$\hat{\eta}^+_l(\Delta t) = \left\lfloor \frac{\Delta t}{T} \right\rfloor \cdot \eta^+(T) + \eta^+(\Delta t - \left\lfloor \frac{\Delta t}{T} \right\rfloor \cdot T).$$

If $\hat{\delta}^-(n)$ is l-repetitive, then its pseudo-inverse $\hat{\eta}^+_l(\Delta t)$ must be $T = \hat{\delta}^-(l)$-repetitive.

Proof. This results from the symmetry of function inversion. ◀

3.2 Global Analysis

The global analysis now couples the local analysis problems according to the following iterative procedure, which is also depicted in Figure 3 (box entitled "original CPA"). Firstly, each header task of a stream $\tau_{i,1}$ has a known activation behavior bounded by $\eta^-_{i,1}(\Delta t), \eta^+_{i,1}(\Delta t)$ and imposed by external event sources. Since initially no event models are available for successor tasks in the stream, i.e. for $\tau_{i,j}$ with $j > 1$, they are initialized with the event model assigned to the header task $\tau_{i,1}$. The local analysis is then performed for each resource, such that response time bounds and output event models are obtained. The computed output event models are then *propagated* to the direct successor tasks, where they are interpreted as input event models. The local analysis is then repeated with the updated event models. If all propagated event models are identical to the event models used in the previous analysis run, a global fixed point is reached and the analysis terminates.

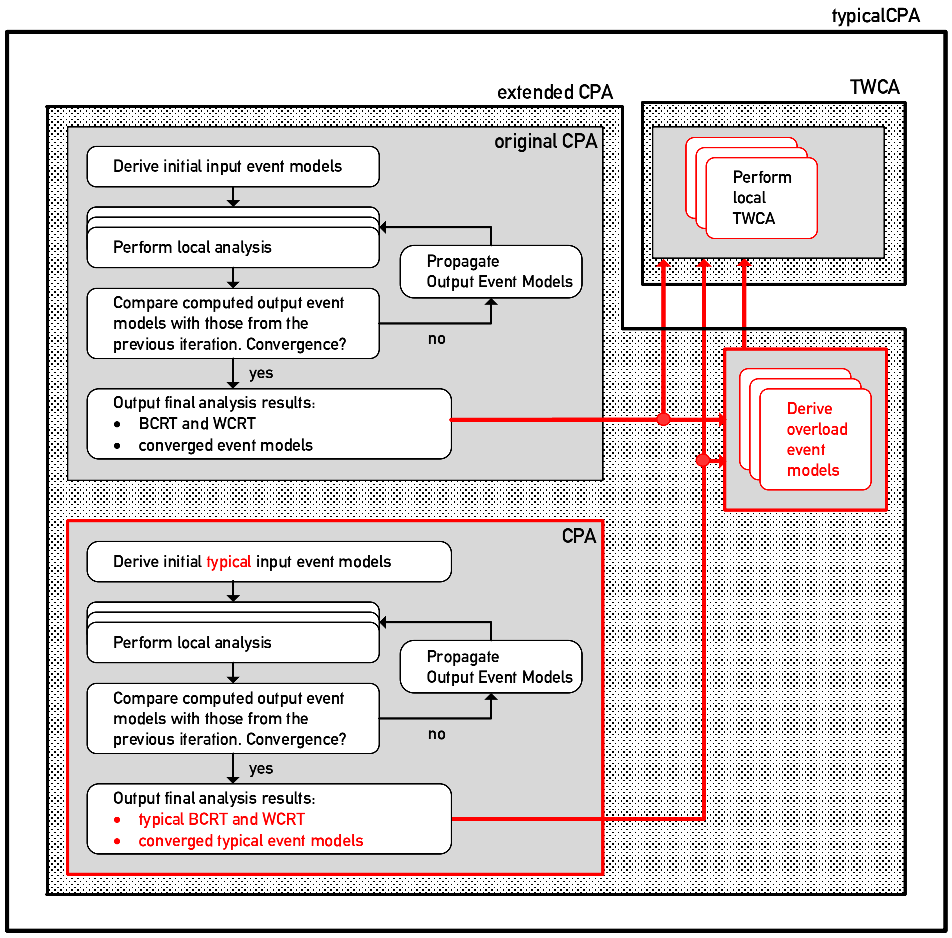

typicalCPA
extended CPA
TWCA
original CPA
Derive initial input event models
Perform local analysis
Propagate Output Event Models
Compare computed output event models with those from the previous iteration. Convergence?
no
yes
Output final analysis results:
• BCRT and WCRT
• converged event models
Perform local TWCA
Derive overload event models
CPA
Derive initial typical input event models
Perform local analysis
Propagate Output Event Models
Compare computed output event models with those from the previous iteration. Convergence?
no
yes
Output final analysis results:
• typical BCRT and WCRT
• converged typical event models

Figure 3 TypicalCPA. *The extended CPA also derives typical and overload event models as detailed in Section 5, which are then processed by a TWCA for each component.* New or adapted elements of CPA and TWCA are marked in red.

4 Typical Worst Case Analysis

Typical Worst Case Analysis (TWCA) models and analyzes systems with a single service-providing resource R_k under transient overload conditions. It provides weakly-hard real-time guarantees for tasks $\mathcal{T}_{R_k}$. In this section, we firstly present which extensions to the CPA system model presented in Section 2 are necessary to apply TWCA. Then the TWCA procedure is introduced together with a needed generalization of a schedulability criterion.

4.1 Extended System Model

The system model of CPA presented in Section 2 is a subset of the TWCA system model. The important extension of the CPA model by TWCA is that each task $\tau_{i,j}$ may be activated by events of two distinct classes, namely by *typical and overload events*. The idea is that in the exclusive presence of typical events, the task set $\mathcal{T}_{R_k}$ is schedulable. In contrast, the supplementary overload events are a potential cause for transient overload.

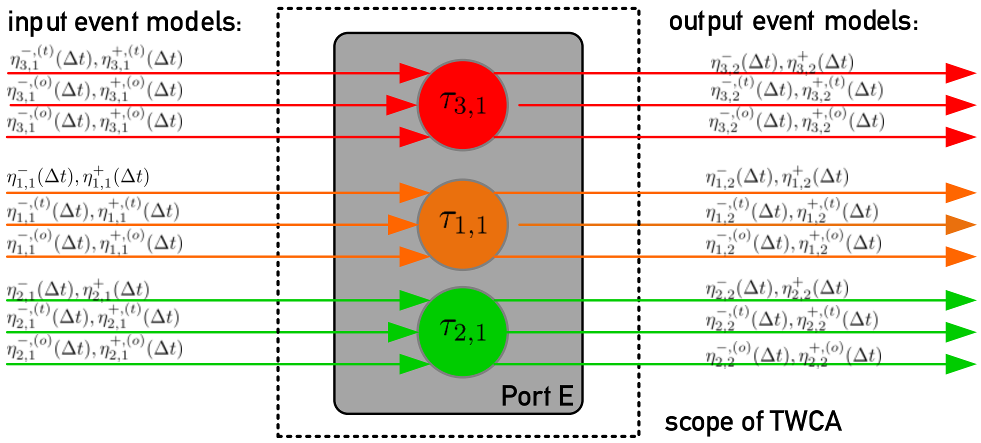

input event models:
$\eta^{-,(t)}_{3,1}(\Delta t), \eta^{+,(t)}_{3,1}(\Delta t)$
$\eta^{-,(o)}_{3,1}(\Delta t), \eta^{+,(o)}_{3,1}(\Delta t)$
$\eta^{-,(o)}_{3,1}(\Delta t), \eta^{+,(o)}_{3,1}(\Delta t)$
$\eta^{-}_{1,1}(\Delta t), \eta^{+}_{1,1}(\Delta t)$
$\eta^{-,(t)}_{1,1}(\Delta t), \eta^{+,(t)}_{1,1}(\Delta t)$
$\eta^{-,(o)}_{1,1}(\Delta t), \eta^{+,(o)}_{1,1}(\Delta t)$
$\eta^{-}_{2,1}(\Delta t), \eta^{+}_{2,1}(\Delta t)$
$\eta^{-,(t)}_{2,1}(\Delta t), \eta^{+,(t)}_{2,1}(\Delta t)$
$\eta^{-,(o)}_{2,1}(\Delta t), \eta^{+,(o)}_{2,1}(\Delta t)$
$\tau_{3,1}$
$\tau_{1,1}$
$\tau_{2,1}$
Port E
output event models:
$\eta^{-}_{3,2}(\Delta t), \eta^{+}_{3,2}(\Delta t)$
$\eta^{-,(t)}_{3,2}(\Delta t), \eta^{+,(t)}_{3,2}(\Delta t)$
$\eta^{-,(o)}_{3,2}(\Delta t), \eta^{+,(o)}_{3,2}(\Delta t)$
$\eta^{-}_{1,2}(\Delta t), \eta^{+}_{1,2}(\Delta t)$
$\eta^{-,(t)}_{1,2}(\Delta t), \eta^{+,(t)}_{1,2}(\Delta t)$
$\eta^{-,(o)}_{1,2}(\Delta t), \eta^{+,(o)}_{1,2}(\Delta t)$
$\eta^{-}_{2,2}(\Delta t), \eta^{+}_{2,2}(\Delta t)$
$\eta^{-,(t)}_{2,2}(\Delta t), \eta^{+,(t)}_{2,2}(\Delta t)$
$\eta^{-,(o)}_{2,2}(\Delta t), \eta^{+,(o)}_{2,2}(\Delta t)$
scope of TWCA

Figure 4 Scope and interface of TWCA.

▶ **Definition 8** (Local typical worst case)**.** If every task $\tau_{i,j} \in \mathcal{T}_{R_k}$ is only activated by typical events, then the task set $\mathcal{T}_{R_k}$ is schedulable even in the most unfavorable scheduling scenario (*local typical worst case*).

▶ **Definition 9** (Local worst case)**.** If every task $\tau_{i,j} \in \mathcal{T}_{R_k}$ is activated by both typical and overload events, then in the most unfavorable scheduling scenario (*local worst case*) the task set $\mathcal{T}_{R_k}$ is possibly unschedulable.

The occurrence of typical or overload activation events over time w.r.t. a task $\tau_{i,j}$ is also modeled by the concept of event flows, while the minimum and maximum frequency of typical and overload event arrival is described by event models. The corresponding definitions are given below, while Figure 4 shows the extended system model with the additional event models.

▶ **Definition 10** (Typical and overload event flows)**.** A typical event flow $e^{(t)}_{i,j}(t)$, resp. overload event flow $e^{(o)}_{i,j}(t)$, is a function which returns the number of typical, resp. overload, events which activate task $\tau_{i,j}$ within the time interval $[0, t)$ in a given execution run.

▶ **Definition 11** (Typical and overload event models)**.** The event models $\eta^{-,(t)}_{i,j}(\Delta t), \eta^{+,(t)}_{i,j}(\Delta t)$, resp. $\eta^{-,(o)}_{i,j}(\Delta t), \eta^{+,(o)}_{i,j}(\Delta t)$, indicate a lower and an upper bound on the number of typical, resp. overload, events which activate task $\tau_{i,j}$ within Δt.

▶ **Definition 12** (Decomposition)**.** Any observed event flow of task $\tau_{i,j}$ which satisfies the lower and upper bounds $\eta^{-}_{i,j}(\Delta t), \eta^{+}_{i,j}(\Delta t)$ can be partitioned in

(1) an event flow of typical events satisfying $\eta^{-,(t)}_{i,j}(\Delta t), \eta^{+,(t)}_{i,j}(\Delta t)$ and

(2) an event flow of overload events satisfying $\eta^{-,(o)}_{i,j}(\Delta t), \eta^{+,(o)}_{i,j}(\Delta t)$.

This implies that the maximum event model $\eta^{+}_{i,j}(\Delta t)$ is *decomposable*, s.t. $\eta^{+}_{i,j}(\Delta t) \leq \eta^{+,(t)}_{i,j}(\Delta t) + \eta^{+,(o)}_{i,j}(\Delta t)$. If $\eta^{+}_{i,j}(\Delta t) = \eta^{+,(t)}_{i,j}(\Delta t) + \eta^{+,(o)}_{i,j}(\Delta t)$ holds, then the maximum event model is said to be *exactly decomposable*. Please refer for illustration to Figure 5c.

The intuition related to the system model is that a computing platform may be designed to provide sufficient processing service for a typical workload. For instance, if all tasks have a periodic (= typical) activation pattern, then the task set is schedulable. If, however, some tasks experience additional sporadic (= overload) activations, then the task set may become unschedulable in unfavorable scheduling scenarios.

4.2 Basic Procedure

The objective of TWCA is to determine weakly-hard real-time guarantees for all tasks in the task set. More precisely, a deadline miss model (DMM) is obtained for every task $\tau_{i,j} \in \mathcal{T}_{R_k}$.

▶ **Definition 13** (Deadline miss model). A deadline miss model for a task $\tau_{i,j}$ is a function $dmm_{i,j} : \mathbb{N} \to \mathbb{N}$ with the property that out of any k consecutive jobs of task $\tau_{i,j}$, at most $dmm_{i,j}(k)$ might miss their deadline $d_{i,j}$.

To compute $dmm_{i,j}(k)$ under SPNP scheduling, TWCA quantifies the impact of overload activations. We summarize the procedure in the following steps.

1. Firstly TWCA derives the maximum impact which a single overload activation of a task $\tau_{m,n} \in hsp(\tau_{i,j})$ can have on the task $\tau_{i,j}$. The impact is counted by the maximum number jobs of task $\tau_{i,j}$ which can miss their deadline due to this overload activation, and is denoted as $N_{i,j}$.
2. It is computed how many overload activations of task $\tau_{m,n}$ can at most influence the k-sequence of task $\tau_{i,j}$. This number is given by $\eta^{+,(o)}_{m,n}(\Delta T^{i,j}_k)$, where $\Delta T^{i,j}_k$ describes the maximum time interval during which a k-sequence of task $\tau_{i,j}$ is sensitive to overload events.
3. The overall impact of task $\tau_{m,n}$ is then derived as the product $N_{i,j} \cdot \eta^{+,(o)}_{m,n}(\Delta T^{i,j}_k)$.
4. Finally, the impact of all $\tau_{m,n}$ tasks which may interfere with task $\tau_{i,j}$ is summed. Interfering tasks have higher or same priority (hsp) than task $\tau_{i,j}$.

Thus we have

$$dmm_{i,j}(k) = \sum_{\tau_{m,n} \in hsp(\tau_{i,j})} N_{i,j} \cdot \eta^{+,(o)}_{m,n}(\Delta T^{i,j}_k) \tag{6}$$

where

$$N_{i,j} = \# \left\{ q \in \mathbb{N}^+ | 1 \leq q \leq K_{i,j} \wedge d_{i,j} < R^+_{i,j}(q) \right\} \tag{7}$$

$$\Delta T^{i,j}_k \leq B^+_{i,j}(K_{i,j}) + \delta^+_{i,j}(k) + (R^+_{i,j} - C^+_{i,j}) \tag{8}$$

Please refer for a detailed explanation to [10].

4.3 Improved Procedure

The presented basic TWCA assumes that *every* isolated overload activation of a task $\tau_{m,n}$ which interferes with task $\tau_{i,j}$ causes at most $N_{i,j}$ deadline misses. The approach presented in [21] improves over the basic TWCA by considering that often actually the *combined* effect of overload from several interferer tasks is required to cause a deadline miss of task $\tau_{i,j}$. We introduce therefore the following definitions.

▶ **Definition 14** (Combination). A local combination $C \subseteq \mathcal{T}_{R_k}$ is a set of tasks which may experience both typical as well as overload activation events, whereas the tasks of the complementary set, $\mathcal{T}_{R_k} \setminus C$, experience only typical activation events.

▶ **Definition 15** (Unschedulable combinations). $R^{+,C}_{i,j}$ denotes the longest response time of task $\tau_{i,j} \in \mathcal{T}_R$, assuming that only tasks in C experience overload activations. A combination C is said to be schedulable w.r.t. to task $\tau_{i,j}$, if $R^{+,C}_{i,j} \leq d_{i,j}$, otherwise it is unschedulable. The set of unschedulable combinations w.r.t. to task $\tau_{i,j}$ is called $\mathcal{U}_{i,j}$.

Note that special local combinations are $C = \emptyset$ and $C = \mathcal{T}_{R_k}$. In this context, $R_{i,j}^{+,\mathcal{T}_{R_k}}$ is the usual worst case response time and $R_{i,j}^{+,\emptyset}$ is called typical worst case response time.

The improved TWCA [21] is now based on the fact that the sensitivity interval $\Delta T_k^{i,j}$ of the k-sequence of task $\tau_{i,j}$ can be divided into a sequence of busy periods [13]. The timing behavior of busy periods is mutually independent, because of the idle times which separate them. Within in any such busy period, an unschedulable combination is necessary to cause at most $N_{i,j}$ deadline misses of task $\tau_{i,j}$ within this interval. A single task $\tau_{m,n}$ can be part of unschedulable combinations at most $\Omega_{m,n} = \eta_{m,n}^{+,(o)}(\Delta T_k^{i,j})$ times, which corresponds to the maximum number of overload activations in $\Delta T_k^{i,j}$.

Let $x_C \in \mathbb{N}$ count the number of busy periods in $\Delta T_k^{i,j}$, which suffer from an unschedulable combination $C \in \mathcal{U}_{i,j}$. Then the DMM can be obtained by solving the following optimization problem

$$dmm_{i,j}(k) = \max\ N_{i,j} \sum_{C\colon C \in \mathcal{U}_{i,j}} x_C \tag{9}$$

$$\text{s.t.} \sum_{C,(m,n)} x_C \leq \Omega_{m,n} \tag{10}$$

$$\text{with } C,(m,n) : (\tau_{m,n}\ \in\ hsp(\tau_{i,j}) \cup \tau_{i,j}) \wedge (\tau_{m,n} \in C) \wedge (C \in \mathcal{U}_{i,j})$$

To determine whether a combination C is schedulable or not, a fast schedulability criterion is required. We rely on the criterion presented in [21], but generalize it for (1) non-unique priorities, and (2) the general relation where the maximum event models are not exactly decomposable. The generalization is presented in Theorem 16; notation and explanations of the theorem contents are given in the corresponding proof and Figure 5.

▶ **Theorem 16** (Generalized schedulability criterion). *Equation 11 formulates a schedulability criterion for task $\tau_{i,j}$ under a given combination C.*

$$\forall l \in K_{i,j} : \sum_{\forall \tau_{m,n} : \tau_{m,n} \in hsp(\tau_{i,j}) \cup \tau_{i,j} \wedge \tau_{m,n} \notin C} wl_{over}^{(m,n),l} \geq \Lambda_{i,j}^l - \Gamma_{i,j}^l. \tag{11}$$

The following abbreviations are used

$$\Lambda_{i,j}^l = B_{i,j}^+(l) - \delta_{i,j}^-(l) - d_{i,j}$$
$$\Gamma_{i,j}^l = \sum_{\tau_{m,n} \in hp(\tau_{i,j})} C_{m,n}^+ \cdot [\eta_{m,n}^+(B_{i,j}^+(l) - C_{i,j}^+) - \eta_{m,n}^+(\Delta t_{i,j}^l)]$$
$$\Delta t_{i,j}^l = \delta_{i,j}^-(l) + d_{i,j} - C_{i,j}^+$$
$$wl_{over}^{(m,n),l} = \begin{cases} C_{m,n}^+ \cdot \left(\eta_{m,n}^+(\Delta t_{i,j}^l) - \eta_{m,n}^{+,(t)}(\Delta t_{i,j}^l)\right) & \textit{for } \tau_{m,n} \in hp(\tau_{i,j}) \\ C_{m,n}^+ \cdot \left(\eta_{m,n}^{+]}(\delta_{i,j}^-(l)) - \eta_{m,n}^{+],(t)}(\delta_{i,j}^-(l))\right) & \textit{for } \tau_{m,n} \in sp(\tau_{i,j}) \cup \tau_{i,j} \end{cases}$$

Proof. Let us verify the schedulability of task $\tau_{i,j}$ under a given combination C, i.e. we verify whether $R_{i,j}^{+,C} \leq d_{i,j}$ is true. We start from the unschedulable local worst case with $C' = \mathcal{T}_{R_k}$, which is represented by the maximum level-$\tau_{i,j}$ busy period which contains $K_{i,j}$ jobs of task $\tau_{i,j}$ (cf. Figure 5a). If the task $\tau_{i,j}$ is schedulable in the local worst case, then it schedulable for every combination and the problem is solved. If, however, task $\tau_{i,j}$ is unschedulable in the local worst case, then some of the $K_{i,j}$ jobs of task $\tau_{i,j}$ miss their deadline. The lth job of $\tau_{i,j}$ exceeds its deadline in the local worst case by (cf. also Figure 5)

$$\Lambda_{i,j}^l = R_{i,j}^+(l) - d_{i,j} = B_{i,j}^+(l) - \delta_{i,j}^-(l) - d_{i,j}.$$

If its deadline is enforced by removing overload, an amount of workload $\Gamma^l_{i,j}$ will disappear automatically. Namely the workload from interfering activations which occur after the deadline but before the non-preemptive execution of the lth job. Jobs of tasks with the same priority (sp) as $\tau_{i,j}$ do not contribute to $\Gamma^l_{i,j}$, because they influence the response time of the lth job only if they have arrived earlier than or simultaneously with this job.

$$\Gamma^l_{i,j} = \sum_{\tau_{m,n} \in hp(\tau_{i,j})} C^+_{m,n} \cdot [\eta^+_{m,n}(B^+_{i,j}(l) - C^+_{i,j}) - \eta^+_{m,n}(\delta^-_{i,j}(l) + d_{i,j} - C^+_{i,j})]$$

The RHS of inequality 11 describes the smallest amount of overload of interfering tasks that needs removed for sufficient schedulability of the lth job of $\tau_{i,j}$ in the maximum busy period. The LHS of Eq. 11 describes how much overload is removed compared to the local worst case, if we assume combination C (cf. Figure 5b for $C = \emptyset$). Under combination C, all tasks $\tau_{m,n} \notin C$ experience only typical activations and their overload is not present. In other words, the tasks $\tau_{m,n} \notin C$ follow their event model $\eta^{+,(t)}_{m,n}(\Delta t)$. In particular, an amount of overload per task $\tau_{m,n}$

$$wl^{(m,n),l}_{over} = \begin{cases} C^+_{i,j} \cdot \left(\eta^+_{m,n}(\Delta t^l_{i,j}) - \eta^{+,(t)}_{m,n}(\Delta t^l_{i,j})\right) & \text{for } \tau_{m,n} \in hp(\tau_{i,j}) \\ C^+_{i,j} \cdot \left(\eta^{+]}_{m,n}(\delta^-_i(l)) - \eta^{+],(t)}_{m,n}(\delta^-_i(l))\right) & \text{for } \tau_{m,n} \in sp(\tau_{i,j}) \cup \tau_{i,j} \end{cases}$$

is removed which impacts the response time of the lth job of task $\tau_{i,j}$. Namely, the interfering overload of $hp(\tau_{i,j})$-tasks until the timely nonpreemptive execution of job $\tau_{i,j}(l)$ is absent. Likewise, the overload of all $sp(\tau_{i,j})$-jobs and overload jobs of $\tau_{i,j}$ are absent, which interfere if they arrive before or simultaneously with job $\tau_{i,j}(l)$.[2] ◀

5 Typical Compositional Performance Analysis

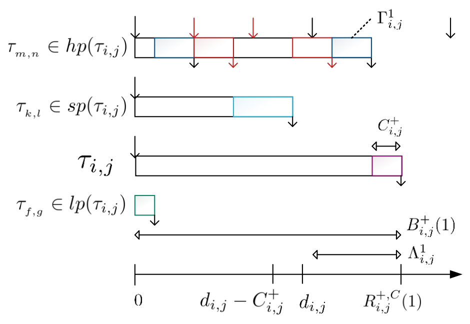
The new framework TypicalCPA, which we develop in this paper, combines CPA and TWCA such that weakly-hard real-time guarantees can be given for tasks in a multi-resource system. More concretely, the local analysis method TWCA will performed for each component after an extended CPA has terminated. This is illustrated in Figure 3. To apply TWCA as a local analysis method, for each task minimum and maximum event models together with the corresponding minimum and maximum typical and overload event models have to be provided. The state-of-the-art CPA, however, computes as a result, besides BCRT and WCRT, so far only the converged minimum and maximum event models of each task (not their typical and overload variants) and thus has to be extended.

In the following we assume that the complete set of event models – $(\eta^-_{i,1}(\Delta t), \eta^+_{i,1}(\Delta t))$, $(\eta^{-,(t)}_{i,1}(\Delta t), \eta^{+,(t)}_{i,1}(\Delta t))$ and $(\eta^{-,(o)}_{i,1}(\Delta t), \eta^{+,(o)}_{i,1}(\Delta t))$ – is given for the header tasks $\tau_{i,1}$, since they are activated by external event sources. The problem to be addressed is how to derive these event models for all successor tasks in the context of CPA such that they can be used for the subsequent TWCA.

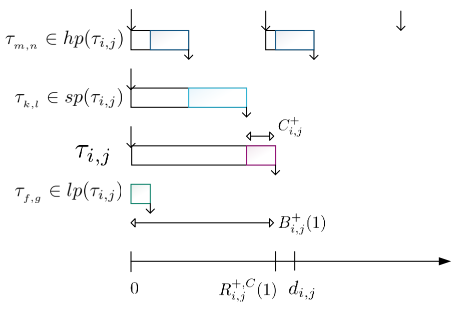
5.1 Basic Definitions

We begin by introducing the concept of a *global combination* describing the activation behavior of each task $\tau_{i,j}$ contained in the global task set $\mathcal{T}$. Due to the existing precedence constraints in a stream s_i, the activation behavior of any task $\tau_{i,j}$ with $j > 1$ is fully determined by

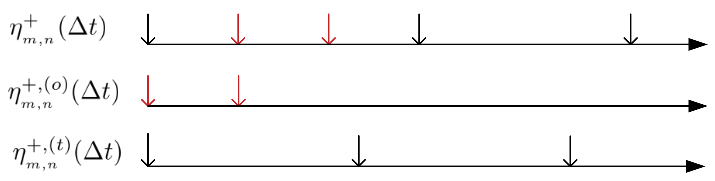
[2] The notation $\eta^{+]}(\Delta t)$ expresses that the maximum event model refers to the *closed* time interval $[0, t]$.



(a) Local worst case busy window with $C = \mathcal{T}_k$, $K_{i,j} = 1$.


$\tau_{m,n} \in hp(\tau_{i,j})$
$\tau_{k,l} \in sp(\tau_{i,j})$
$C^+_{i,j}$
$\tau_{i,j}$
$\tau_{f,g} \in lp(\tau_{i,j})$
$B^+_{i,j}(1)$
0
$R^{+,C}_{i,j}(1)$
$d_{i,j}$

(b) Local typical worst case busy window with $C = \emptyset$.


$\eta^+_{m,n}(\Delta t)$
$\eta^{+,(o)}_{m,n}(\Delta t)$
$\eta^{+,(t)}_{m,n}(\Delta t)$

(c) Exemplary decomposition of the maximum event model $\eta^+_{m,n}(\Delta t)$ in the maximum overload event model $\eta^{+,(o)}_{m,n}(\Delta t)$ and the maximum typical event model $\eta^{+,(t)}_{m,n}(\Delta t)$ for task $\tau_{m,n}$.

Figure 5 Theorem 16: Generalized schedulability criterion.

the respective predecessor task and therefore in the end by the header task $\tau_{i,1}$. It is thus sufficient to include the activation behavior of the header tasks in the definition of a global combination.

▶ **Definition 17** (Global combination). A global combination $C_g \subseteq \{\tau_{i,1} | \, \forall i : \tau_{i,1} \in \mathcal{T}\}$ is a set of header tasks which may experience both typical as well as overload activations. All other header tasks follow their typical event model.

Special global combinations are the *global typical combination* with $C_g = \emptyset$, and the *global worst case combination* with $C_g = \{\tau_{i,1} | \, \forall i : \tau_{i,1} \in \mathcal{T}\}$.

▶ **Definition 18** (Schedulability of a global combination). We say a global combination C_g is schedulable if and only if under all possible scheduling scenarios (1) all streams can satisfy their end-to-end deadlines $D_{i,j}$ and (2) every task meets its local deadline $d_{i,j}$.

We require that the given event models of the header tasks are such that the following schedulability constraints are respected.

▶ **Definition 19** (Global typical worst case). If the system behaves according to the global typical combination, then the task set $\mathcal{T}$ is schedulable even in the most unfavorable scenario (global typical worst case).

▶ **Definition 20** (Global worst case). If the system behaves according to the global worst case combination, the task set $\mathcal{T}$ is possibly unschedulable in the most unfavorable scenario (global worst case).

We would like to mention that for computing weakly-hard real-time guarantees, naturally only systems which *are* unschedulable in the global worst case are of interest.

5.2 Computation of Minimum and Maximum Event Models

While for the header tasks the minimum and maximum event model $\eta^-_{i,j}(\Delta t)$, $\eta^+_{i,j}(\Delta t)$ is given by the system specification, it has to be derived for successor tasks $\tau_{i,j}$ with $j > 1$. The classical CPA is capable of deriving these event models for all successor tasks from the original CPA input model as defined in Section 2. Thus CPA explores here the most favorable and the most unfavorable behavior of the global worst case combination.

5.3 Computation of Minimum and Maximum Typical Event Models

The minimum and maximum typical event model $\eta^{-,(t)}_{i,j}(\Delta t)$, $\eta^{+,(t)}_{i,j}(\Delta t)$ have also to be computed for the successor tasks $\tau_{i,j}$ with $j > 1$. Our claim is that CPA can also be used for this purpose, given that in the input model the worst case bounds $\eta^-_{i,1}(\Delta t)$, $\eta^+_{i,1}(\Delta t)$ are replaced by the typical event models $\eta^{-,(t)}_{i,1}(\Delta t)$, $\eta^{+,(t)}_{i,1}(\Delta t)$. In other words, CPA is now applied for the best case and worst case scenario where all header tasks see only typical events (global typical combination). CPA, which is agnostic of event types, computes the converged minimum and maximum event models for all stream tasks. We assume in this paper that all typical events that are injected at the head of a stream keep their typical nature while propagating through the system. Knowing that only typical events have served for stream activation, we can interpret the CPA-derived event models as typical and have thus $\eta^{-,(t)}_{i,j}(\Delta t)$ and $\eta^{+,(t)}_{i,j}(\Delta t)$ for all stream tasks.

5.4 Computation of Minimum and Maximum Overload Event Models

Finally, our intention is to obtain the minimum and maximum overload event models for each successor task $\tau_{i,j}$ with $j > 1$. We begin by describing how an arbitrary event flow $e_{i,j}(t)$ can be decomposed in a typical event flow $e^{(t)}_{i,j}(t)$ and an overload event flow $e^{(o)}_{i,j}(t)$. In this context, we use the concept of a sliding window function which returns a maximum event model for a specific event flow.

▶ **Definition 21** (Sliding window function). A sliding window function f_{slw} takes a specific event flow $e_{i,j}(t)$ of task $\tau_{i,j}$ defined on $0 \leq t \leq T$ as an input, and returns a maximum event model for $e_{i,j}(t)$, denoted as $\eta^+_{e_{i,j},T}(\Delta t)$ for any interval size $0 \leq \Delta t \leq T$. This maximum event model $\eta^+_{e_{i,j},T}(\Delta t)$ is derived by passing a window of size Δt over the event flow $e_{i,j}(t)$ of length T and noting down the maximum number events contained in any position of the window Δt such that

$$\eta^+_{e_{i,j},T}(\Delta t) = \max_{t_1,t_2\,:\,0\leq t_1\leq t_2\leq T\wedge t_2-t_1=\Delta t} \{e_{i,j}(t_2) - e_{i,j}(t_1)\}\,.$$

▶ **Theorem 22** (Decomposition of an event flow). *Let $e_{i,j}(t)$ be an arbitrary event flow of length T belonging to task $\tau_{i,j}$. Known bounds for the activation frequency of task $\tau_{i,j}$ are i.a. $\eta^+_{e_{i,j},t}(\Delta t)$ for all (sub)lengths of the event flow with $0 \leq t \leq T$ and the maximum typical event model $\eta^{+,(t)}_{i,j}(\Delta t)$. A valid decomposition of $e_{i,j}(t)$ in a typical and overload event flow is given by*

$$e^{(o)}_{i,j}(t) = \max_{0\leq\Delta t\leq t}\left\{0,\ \eta^+_{e_{i,j},t}(\Delta t) - \eta^{+,(t)}_{i,j}(\Delta t)\right\} \qquad e^{(t)}_{i,j}(t) = e_{i,j}(t) - e^{(o)}_{i,j}(t). \tag{12}$$

Proof. The event flow $e_{i,j}(t)$ cannot contain more than $\eta^{+,(t)}_{i,j}(\Delta t)$ typical events in the observed interval $[0, t)$ by Def. 11, where $\Delta t = t - 0$. All events that occur additionally to the maximum number of typical events $\eta^{+,(t)}_{i,j}(\Delta t)$ in $[0, t)$ are a potential source of overload in the system and can therefore be safely interpreted as overload events.

To determine the number overload events in $e_{i,j}(t)$, we (1) apply the sliding window function to $e_{i,j}(t)$ within $[0, t)$ which results in $\eta^{+}_{e_{i,j},t}(\Delta t)$, and then (2) compare pointwise $\eta^{+}_{e_{i,j},t}(\Delta t)$ with $\eta^{+,(t)}_{i,j}(\Delta t)$. Pointwise comparison is done chronologically by increasing continuously the size of Δt with $0 \le \Delta t \le t$. The largest nonnegative difference $\max\limits_{0\le\Delta t\le t}\left\{0,\ \eta^{+}_{e_{i,j},t}(\Delta t) - \eta^{+,(t)}_{i,j}(\Delta t)\right\}$, is the number of overload events in $e_{i,j}(t)$.

Why is it not sufficient to compute $\max\left\{0, \eta^{+}_{e_{i,j},t}(\Delta t) - \eta^{+,(t)}_{i,j}(\Delta t)\right\}$ for $\Delta t = t$? Let $\Delta t'$ be the first interval, where the maximum budget of typical events is exceeded by the event flow such that $\eta^{+}_{e_{i,j},t}(\Delta t') - \eta^{+,(t)}_{i,j}(\Delta t') > 0$. This information should not be contradicted by a later smaller value of overload events derived at $\Delta t'' > \Delta t'$. This, however, may happen due to the cumulative representation of event arrival within Δt by event models, where information on the alignment of events gets lost with increasing interval size. The alignment information is however important to distinguish overload from typical events. The formulation $e^{(o)}_{i,j}(t) = \max\limits_{0\le\Delta t^*\le t}\left\{0,\ \eta^{+}_{e_{i,j},t}(\Delta t^*) - \eta^{+,(t)}_{i,j}(\Delta t^*)\right\}$ preserves the information on the maximum number of overload events once gained at Δt^*. Also, $e^{(o)}_{i,j}(t)$ is a wide-sense increasing function which accumulates the number of occurred overload events over time, and therefore satisfies Def. 10 of an event flow. Furthermore, we have $e^{(t)}_{i,j}(t) = e_{i,j}(t) - e^{(o)}_{i,j}(t)$ since an event in an event flow can either be overload or typical. ◀

In the following Theorem 23, we state how to compute a maximum overload event model. We would like to note that the minimum overload event model is the zero function $\eta^{+,(o)}_{i,j}(\Delta t) = 0$ since overload events can be completely absent cf. global typical combination.

▶ **Theorem 23** (Obtaining an overload event model). *A maximum overload event model is*

$$\eta^{+,(o)}_{i,j}(\Delta t) = f_{slw}\left(\max_{0\le\Delta t^*\le\Delta t}\left\{\eta^{+}_{i,j}(\Delta t^*) - \eta^{+,(t)}_{i,j}(\Delta t^*)\right\}\right)$$

where f_{slw} is a sliding window function.

Proof. An upper bound for all event flow-specific maximum event models $\eta^{+}_{e_{i,j},T}(\Delta t)$ of task $\tau_{i,j}$ is the maximum event model $\eta^{+}_{i,j}(\Delta t)$ by Def. 2. Thus we have

$$\max_{0\le\Delta t^*\le t}\left\{0,\ \eta^{+}_{e_{i,j},t}(\Delta t^*) - \eta^{+,(t)}_{i,j}(\Delta t^*)\right\} \le \max_{0\le\Delta t^*\le\Delta t}\left\{\eta^{+}_{i,j}(\Delta t^*) - \eta^{+,(t)}_{i,j}(\Delta t^*)\right\}.$$

In other words, the overload event flow $\tilde{e}^{(o)}_{i,j}(t) = \max\limits_{0\le\Delta t^*\le t}\left\{\eta^{+}_{i,j}(\Delta t^*) - \eta^{+,(t)}_{i,j}(\Delta t^*)\right\}$ is always larger than any other arbitrary overload event flow $e^{(o)}_{i,j}(t)$. To derive from the largest overload event flow $\tilde{e}^{(o)}_{i,j}(t)$ the corresponding maximum overload event model, we apply once again the sliding window function such that $\tilde{e}^{(o)}_{i,j}(t_2) - \tilde{e}^{(o)}_{i,j}(t_1) \le \eta^{+,(o)}_{i,j}(t_2 - t_1) = f_{slw}\left(\tilde{e}^{(o)}_{i,j}(t_2 - t_1)\right)$. The computation of the overload event model $\eta^{+,(o)}_{i,j}(\Delta t)$ is illustrated in Figure 6. ◀

Calculating a maximum overload event model according to Theorem 23 requires a high computational effort since the sliding window approach has to be applied to the infinitely long event flow $\tilde{e}^{(o)}_{i,j}(t) = \max\limits_{0\le\Delta t^*\le t}\left\{\eta^{+}_{i,j}(\Delta t^*) - \eta^{+,(t)}_{i,j}(\Delta t^*)\right\}$. Fortunately most event flows have a repetitive behavior or can be approximated by repetitive functions, so that the effort to derive overload event models is significantly reduced. In the following, we discuss special and practically relevant cases for the computation of overload event models.

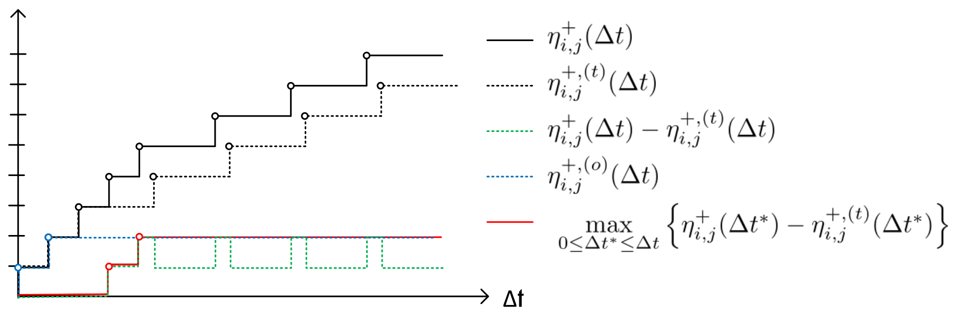


Figure 6 Computing a maximum overload event model.

▶ **Case 1** (Zero typical event model). *In this trivial but important case, the task $\tau_{i,j}$ has a zero typical event model $\eta^{+,(t)}_{i,j}(\Delta t) = 0$. Obviously, we have $\eta^{+,(o)}_{i,j}(\Delta t) = \eta^+_{i,j}(\Delta t)$. This case is relevant for header tasks, which have the character of a sporadic interferer.*

▶ **Case 2** (Zero overload event model). *In a second trivial but important case, the maximum and maximum typical event model of task $\tau_{i,j}$ are identical such that $\eta^+_{i,j}(\Delta t) = \eta^{+,(t)}_{i,j}(\Delta t)$. Consequently, we have a zero overload event model $\eta^{+,(o)}_{i,j}(\Delta t) = 0$. Header tasks with a periodic activation have often this behavior.*

▶ **Case 3** (Repetitive overload event flow). *If the overload event flow $\tilde{e}^{(o)}_{i,j}(t)$ is T-repetitive possibly with an offset (cf. Lemma 7), then applying the sliding window algorithm can be restricted to the interval $[0, 2T)$ to construct the maximum overload event model. In the following Theorem 24, we show that a T-repetitive overload event flow is obtained if the event model $\eta^+_{i,j}(\Delta t)$ and the typical event model $\eta^{+,(t)}_{i,j}(\Delta t)$ are both T-repetitive extensions (which can be achieved by appropriate output model computation described Section 3.1.2).*

▶ **Theorem 24** (Repetitive overload event flow). *If the event model $\eta^+(\Delta t)$ and the typical event model $\eta^{+,(t)}(\Delta t)$ are both T-repetitive extensions, then the resulting overload event flow $\tilde{e}^{(o)}_{i,j}(t)$ is likewise T-repetitive, such that*

$$\tilde{e}^{(o)}_{i,j}(t) = \max_{0\leq\Delta t^*\leq t}\{\left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot \left(\eta^+(T) - \eta^{+,(t)}(T)\right) \\ + \eta^+(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor T) - \eta^{+,(t)}(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor T)\}.$$

Proof.

$$\max_{0\leq\Delta t^*\leq\Delta t}\{\eta^+(\Delta t^*) - \eta^{+,(t)}(\Delta t^*)\} = \max_{0\leq\Delta t^*\leq\Delta t}\{\left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot \eta^+(T) + \eta^+(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot T) \\ - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot \eta^{+,(t)}(T) - \eta^{+,(t)}(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot T)\} = \max_{0\leq\Delta t^*\leq\Delta t}\{\left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot \\ (\eta^+(T) - \eta^{+,(t)}(T)) + \eta^+(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor T) - \eta^{+,(t)}(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor T)\} \\ \overset{\eta^{diff}(\Delta t)=\eta^+(\Delta t)-\eta^{+,(t)}(\Delta t)}{=} \max_{0\leq\Delta t^*\leq\Delta t}\{\left\lfloor\frac{\Delta t^*}{T}\right\rfloor \cdot \eta^{diff}(T) + \eta^{diff}(\Delta t^* - \left\lfloor\frac{\Delta t^*}{T}\right\rfloor T)\}$$ ◀

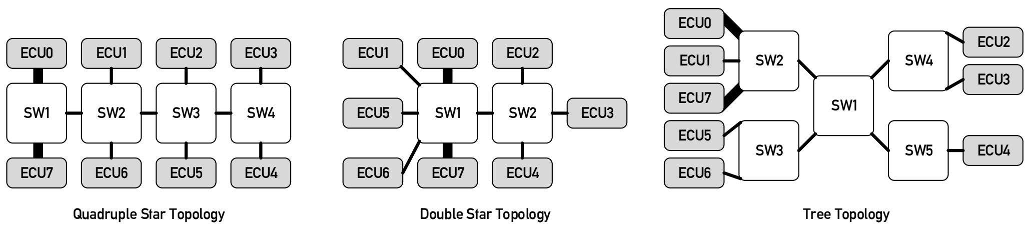


Figure 7 Network topologies. Thin lines represent links at 100 Mbit/s, while thick lines represent links at 1 Gbit/s. A maximum wire length of 10 m is assumed, which translates to a maximum wire propagation delay of 33 ns.

6 Experiments

The presented experiments focus on computing end-to-end (m, k)-guarantees for traffic streams in realistic network settings, while exploring how a varying amount of overload impacts the timing behavior of the investigated system.

6.1 System Generation

The case study presented in Thiele et al. [20] provides characteristics of future automotive backbone networks by Daimler. Based on this data, we have randomly generated a set of automotive switched Ethernet networks with mapped traffic streams. Firstly, let us present the data used from the case study. Figure 7 illustrates three possible network topologies. The topologies vary in the number of switches (SWs) which interconnect 8 electronic control units (ECUs). Links operate at 100 Mbit/s, only ECU0 and ECU7 are equipped with 1Gbit/s links due to high load. Stream characteristics are described statistically by [20], they are summarized in Table 1a. There are 50 periodic control streams of highest priority and 4 periodic camera streams of lower priority. Control streams have relatively small payloads and rather long periods, while camera streams have large payloads and shorter periods. Some of the streams are unicast, others are multicast or broadcast. A periodically sent Ethernet frame is mapped to exactly one stream. Information on the frame payload as well as on periods is given by [20] only in form of minimum and maximum values, averages, and quartiles for the purpose of data anonymization. In case of camera traffic, the number of streams is too small for quantifying quartiles. IPv4/UDP is used at the network/transport layer, which adds 28 bytes of protocol overhead (not shown in Table 1a). Furthermore, the communication matrix in Table 1b is given by [20] indicating the number of control and camera streams sent between a tuple of nodes. We use a parser to translate the network described in terms of topology and streams into a CPA/TWCA system model as defined in Sections 2 and 4.1.

Secondly, we describe the random generation of systems which conform to the presented properties. The generation process is designed to produce a configurable number of systems and consists of several runs. A single generation run first creates the set of 54 streams with their respective source and destination ECUs, and then the streams are mapped to each of the three topologies. A run thus creates 3 systems at once. However, this set of 3 systems is discarded if at least one is not schedulable to enable meaningful comparisons between the different topologies.

- *Generation of control streams.* Periods and payloads of control streams are only described by statistic figures. Therefore, we used fitting to find distributions which come closest the indicated average and quartiles. For the periods, we opted for a Weibull distribution

Table 1 Traffic properties as given in Thiele et al. [20].

	CONTROL	CAMERA
STREAMS		
# total	50	4
# unicast	26	3
# 2-cast	13	1
# 3-cast	4	0
# 4-cast	1	0
# broad-cast	6	0
FRAME PAYLOAD IN BYTES		
[min, max]	$[1, 250]$ B	[875, 1400] B
average	54 B	1231 B
quartiles	$q_{0.25} = 8$ B $q_{0.50} = 25$ B $q_{0.75} = 74$ B	
PERIOD		
[min, max]	$[5ms, 1s]$	$[100us$ $1ms]$
average	$182ms$	$440us$
quartiles	$q_{0.25} = 10ms$ $q_{0.50} = 40ms$ $q_{0.75} = 175ms$	

(a) Stream characteristics.

Src/Dst	ECU0	ECU1	ECU2	ECU3	ECU4	ECU5	ECU6	ECU7
ECU 0		1			1 1		10	2 2
ECU 1	1							1 1
ECU 2					5			
ECU 3					1			
ECU 4	1	2	3	3		1	1	1
ECU 5							3	2
ECU 6	10	6	4	3	4	3		10
ECU 7	5	2	2	2	4 1	3	8	

(b) Communication matrix indicating the number of control streams (black number, 1st entry) and camera streams (blue number, 2nd entry) between a pair of ECUs.

with the parameters $shape = 0.54$ and $scale = 88.09$. For the payload, an exponential distribution with $\lambda = 0.02$ was used.

- *Generation of camera streams.* The few, i.e. 4, camera streams $s_{cam,i}$ are assigned the same payloads and periods in each system generation run: $s_{cam,0} \mapsto (100\mu s, 875B)$; $s_{cam,1} \mapsto (1ms, 1400B)$; $s_{cam,2} \mapsto (330\mu s, 1325B)$; $s_{cam,3} \mapsto (330\mu s, 1325B)$.
- *Generation of stream sources & destinations and topology mapping.* The given communication matrix defines constraints on pairs of source-destination ECUs and on the number of streams sent between them. Stream sources & destinations are generated randomly respecting these constraints. The traffic is then mapped to each of the 3 topologies, creating 3 different systems with identical streams.
- *Schedulability test.* For control streams, local deadlines are set to the stream period and the end-to-end deadline is the sum of the local deadlines. For camera streams, we choose arbitrarily an end-to-end deadline of $2ms$ ($s_{cam,0}$, $s_{cam,2}$, $s_{cam,3}$) or $4ms$ ($s_{cam,1}$), such that – without any overload in the system – worst case stream latencys (WCSLs) of camera streams are already close to their end-to-end deadlines.[3] Local camera deadlines

[3] The worst case stream latency (WCSL) for a unicast stream is computed by summing the WCRTs of tasks included in the stream. For multi- or broadcast streams, the WCSLs are computed separately for each path from the source to a destination.

are derived by uniform distribution of the end-to-end deadline. Based on these timing constraints, the generated systems are filtered such that they are all schedulable as mentioned above.

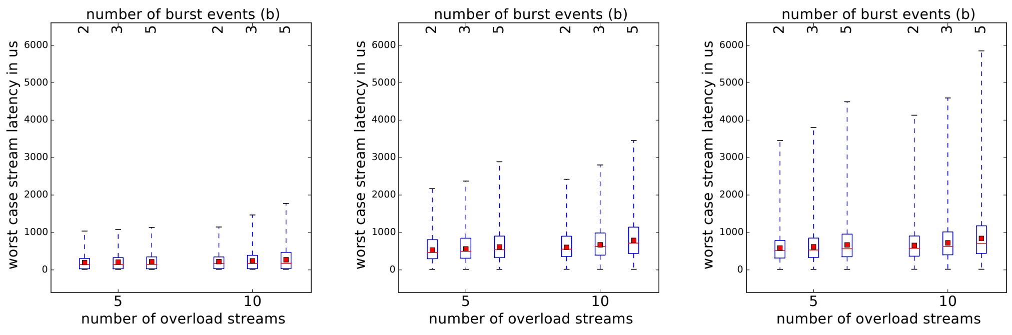
After a generation run, we dispose of a set of 3 systems in which no overload is present. We then add sporadic control streams to each system as transient overload. We see this as a realistic extension of the system description, representing event-triggered communication. A sporadic control stream s' is a duplicate of a randomly chosen control stream s from the original stream set but with modified activation behavior. The typical activation behavior of s' is zero, while the nonzero overload activation behavior is modeled as sporadically bursty [16]: A burst of b events with a minimum distance T_{in} is repeated after an outer period T_{out} such that $\eta^{+,(o)}(\Delta t) = \left\lfloor \frac{\Delta t}{T_{out}} \right\rfloor \cdot b + \min\left\{ \left\lceil \frac{\Delta t - \left\lfloor \frac{\Delta t}{T_{out}} \right\rfloor \cdot T_{out}}{T_{in}} \right\rceil, b \right\}$. While the burst length b is used as a variable parameter in the experiments, fixed parameters are $T_{in} = 100\mu s$ and T_{out} is 10-times the period of the original stream s.

6.2 Experimental Results

In the experiments, we investigate the impact of overload on the timing behavior of the generated systems. For each presented overload configuration, we randomly generated *50* systems of the same topology. We first present worst case stream latencys (WCSLs), and then discuss the DMMs computed for streams. The results in this section are presented in box plots as, for instance, in Figure 8a. This is done to summarize results (WCSLs or DMMs) over all streams from a set of similar systems. A single box plot indicates the average (red square) and the quartiles $q_{0.25}$, $q_{0.50}$, $q_{0.75}$ of the results. The 1st and 3rd quartiles $q_{0.25}$ and $q_{0.75}$ are the top and the bottom of the blue framed box, while the red band inside the box is the 2nd quartile (median). The whiskers indicate results outside the quartiles.

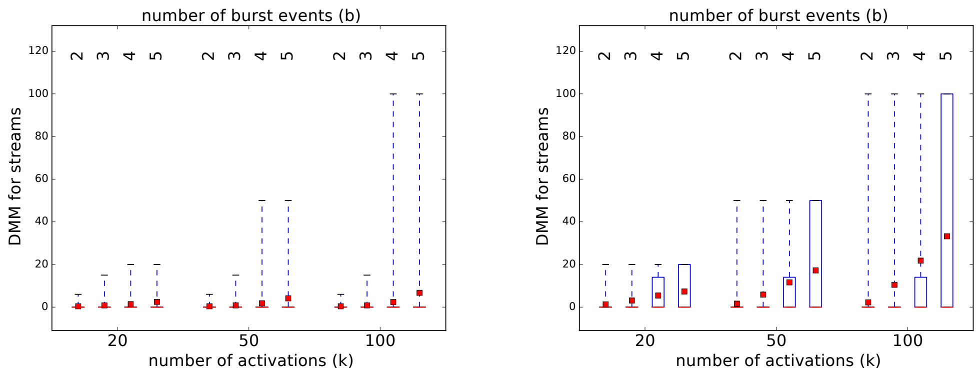
Worst Case Stream Latencies. The WCSLs depend both on the system characteristics as well as on the amount of introduced overload. Figure 8 shows that the double star topology has the shortest WCSLs, compared to to the tree topology with intermediate WCSLs and the quadruple star topology with even higher WCSLs. This behavior is due to the varying number and extent of contention points in the different topologies. Moreover, Figure 8 confirms the intuition that WCSLs increase with the amount of overload in the system, which is controlled by the number of overload streams in the system and the number of burst events b of each overload stream.

Deadline Miss Models of Streams. While the control streams satisfy their end-to-end deadlines even in the presence of overload, camera streams suffer from occasional deadline misses in particular in case of the quadruple star topology. A deadline miss in the context of a camera stream can be interpreted as a frame loss which impacts then video quality. We therefore focus on the DMMs of the camera streams. Figure 9 illustrates the DMMs for all camera streams of generated systems with quadruple star topology. Overload is varied by the number of overload streams and the burst length. We compute the DMM of a unicast stream as the sum of the task DMMs included in the stream. In the case that one or more local deadlines are violated but the global deadline is satisfied, the stream DMM is set to zero. Multicast and broadcast streams are decomposed into unicast streams in order to compute the DMMs according to the above rule. Figure 9a indicates DMMs for camera streams in the presence of 5 sporadic overload streams, while Figure 9b shows DMMs for an increased number of 10 sporadic overload streams. Table 2 lists the nonzero DMMs



(a) Double star topology. **(b)** Tree topology. **(c)** Quadruple star topology.

Figure 8 Worst case latencies of control and camera streams under varying topologies and overload. Each single box plot is based on the streams of 50 randomly generated systems with the indicated properties (num. of bursts, num. of overload streams).



(a) 5 overload streams. **(b)** 10 overload streams.

Figure 9 DMMs for camera streams under varying overload for the quadruple star topology. *Results evaluate camera streams in 50 systems. For a multicast camera stream with n destinations, there are n end-to-end DMMs computed.*

results for $k = 100$ to get a more detailed impression of the individual weakly-hard real-time guarantees. The number of deadline misses grows as expected with the number of overload streams. Furthermore, the m-k-ratio is improving for growing k.

- For 5 overload streams many camera streams are schedulable for any burst length. Few systems have camera streams that are not schedulable. Among these systems with late camera streams, most of them have a very acceptable (m, k) behavior – in particular for $b \in \{2, 3\}$.
- For 10 overload streams more camera streams experience occasional deadline misses. For $b \in \{2, 3\}$, the maximum number of deadline misses m in k executions is acceptable for many camera streams depending on system requirements. For $b \geq 4$ many of the investigated systems are clearly overloaded.

A note on run times: On a PC with an Intel i5-4210M processor at 2.6 GHz and 8GB RAM, the analysis of a single system is in the order of 15-30 seconds.

Table 2 Details on nonzero DMM results for camera streams for $k = 100$.

bursts	nonzero $\mathbf{dmm}(\mathbf{100})$ results with number of occurrence n in brackets (n)
5 overload streams	
$b = 2$	**2**(6), **3**(5), **4**(15), **6**(2)
$b = 3$	**2**(6), **3**(5), **4**(22), **5**(1), **6**(1), **7**(6), **9**(1), **13**(1), **15**(1)
$b = 4$	**2**(2), **3**(3), **4**(19), **5**(4), **6**(1), **7**(1), **8**(2), **9**(1), **10**(3), **11**(2), **12**(1), **13**(1), **14**(1), **16**(1), **18**(1), **30**(1), **100**(3)
10 overload streams	
$b = 2$	**2**(2), **3**(2), **4**(31), **5**(1), **6**(6), **11**(2), **12**(1), **14**(1), **16**(1), **19**(1), **100**(3)
$b = 3$	**4**(15), **5**(5), **7**(2), **8**(1), **9**(1), **10**(2), **12**(5), **14**(2), **15**(4), **16**(2), **100**(23)
$b = 4$	**4**(2), **5**(1), **8**(1), **9**(3), **10**(1), **13**(2), **14**(10), **16**(1), **20**(2), **21**(2), **30**(1), **100**(51)

7 Related Work

The seminal paper by Bernat et al. [1] has presented the principles of weakly-hard real-time systems. It summarizes existing work in a similar direction, introduces (m,k)-constraints, and derives (m,k)-guarantees for periodic task sets with known offsets under fixed priority scheduling. More powerful verification techniques for weakly-hard real-time systems have been subsequently developed. In particular, Quinton et al. [14] has introduced a method called TWCA, which can handle more comprehensive system models covering, e.g., arbitrary activation event models. The initial work [14] has been extended and refined in a sequence of publications; the latest analysis version is presented in [21]. A new and recent development is the verification technique for weakly-hard real-time systems presented by Sun et al. [19]. The work by Sun et al. [19] has only a limited focus on systems with fully periodic tasks with unknown offsets under fixed priority scheduling, but it has a higher accuracy than TWCA since it provides exact results. However, all of the verification techniques are restricted to systems with a single service-providing resource. In this paper, we lift this restriction by integrating TWCA as local analysis technique in the context of the CPA framework [11]. CPA is an established compositional analysis framework, which uses for each component a dedicated scheduling analysis and specifies the coupling of the component-based results. The advantage of using a compositional analysis framework is that large and heterogeneous systems can be analyzed. The choice of the combination (TWCA, CPA) is due to the similarities in the system models and interface definitions, which reduces the number of compatibility issues.

8 Conclusion

In this paper, we presented TypicalCPA which is the first verification method for weakly-hard real-time systems with *multiple* resources and we evaluated it in a network context with traffic streams. Previous verification techniques providing weakly-hard real-time guarantees have aimed at systems with only a single service-providing resource. The method builds on (1) CPA, a compositional performance verification framework for hard real-time guarantees, and (2) TWCA, an analysis method which derives weakly-hard real-time guarantees for systems with a single resource. CPA allows to use different local scheduling analysis techniques for each component in the investigated system, and defines a coupling mechanism between the results provided by each component analysis. We have interpreted TWCA as such a local scheduling analysis technique, but we had to extend (1) elements of TWCA as well as (2) the existing coupling mechanism to achieve compatibility of both CPA and TWCA. In particular, the computation and propagation of typical and overload event models between tasks on different resources has been introduced. In an industrial case study, focusing on automotive switched Ethernet networks, we demonstrated the applicability of TypicalCPA to realistic problems. In the future, we intend to work on improved accuracy of our results.

References

1 Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.

2 Rainer Blind and Frank Allgöwer. Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pages 7510–7515. IEEE, 2015.

3 Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

4 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional performance analysis in python with pycpa. *Proc. of WATERS*, 2012.

5 Jonas Diemer, Philip Axer, Daniel Thiele, and Johannes Schlatow. pyCPA. URL: `https://pycpa.readthedocs.io/en/latest/index.html#`.

6 Jonas Diemer, Jonas Rox, and Rolf Ernst. Modeling of ethernet avb networks for worst-case timing analysis. *IFAC Proceedings Volumes*, 45(2):848–853, 2012.

7 Jonas Diemer, Daniel Thiele, and Rolf Ernst. Formal worst-case timing analysis of ethernet topologies with strict-priority and avb switching. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 1–10. IEEE, 2012.

8 Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 53–62. IEEE, 2014.

9 Mongi Ben Gaid, Daniel Simon, and Olivier Sename. A design methodology for weakly-hard real-time control. *IFAC Proceedings Volumes*, 41(2):10258–10264, 2008.

10 Zain AH Hammadeh, Sophie Quinton, and Rolf Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *Proceedings of the 14th International Conference on Embedded Software*, page 10. ACM, 2014.

11 Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis–the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

12 Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

13 John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209. IEEE, 1990.

14 Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 515–520. EDA Consortium, 2012.

15 Parameswaran Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

16 Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, TU Braunschweig, IDA, 2005.

17 Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 185–190. ACM, 2008.

18 Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Intern. Conference on HW/SW Codesign and System Synthesis. Proceedings*, pages 185–190, New York, 2008. ACM.

19 Youcheng Sun and Marco Di Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):171, 2017.

20 Daniel Thiele, Philip Axer, Rolf Ernst, and Jan R Seyler. Improving formal timing analysis of switched ethernet by exploiting traffic stream correlations. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, page 15. ACM, 2014.

21 Wenbo Xu, Zain AH Hammadeh, Alexander Kröller, Rolf Ernst, and Sophie Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 247–256. IEEE, 2015.

Quantifying the Resiliency of Fail-Operational Real-Time Networked Control Systems

Arpan Gujarati
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
arpanbg@mpi-sws.org

Mitra Nasri[1]
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
mitra@mpi-sws.org

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

In time-sensitive, safety-critical systems that must be fail-operational, active replication is commonly used to mitigate transient faults that arise due to electromagnetic interference (EMI). However, designing an effective and well-performing active replication scheme is challenging since replication conflicts with the size, weight, power, and cost constraints of embedded applications. To enable a systematic and rigorous exploration of the resulting tradeoffs, we present an analysis to quantify the resiliency of fail-operational networked control systems against EMI-induced memory corruption, host crashes, and retransmission delays. Since control systems are typically robust to a few failed iterations, *e.g.*, one missed actuation does not crash an inverted pendulum, traditional solutions based on hard real-time assumptions are often too pessimistic. Our analysis reduces this pessimism by modeling a control system's inherent robustness as an (m, k)-firm specification. A case study with an active suspension workload indicates that the analytical bounds closely predict the failure rate estimates obtained through simulation, thereby enabling a meaningful design-space exploration, and also demonstrates the utility of the analysis in identifying non-trivial and non-obvious reliability tradeoffs.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases probabilistic analysis, reliability analysis, networked control systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.16

Related Version An extended version of this paper is available as a technical report [25], `http://www.mpi-sws.org/tr/2018-005.pdf`.

1 Introduction

Networked control systems (NCSs) – where sensors, controllers, and actuators belonging to one or more control loops are connected by a *shared* network – are widely deployed in contemporary cyber-physical systems as they offer many practical advantages over dedicated wiring solutions, not the least of which are cost and weight savings [26].

[1] Mitra Nasri is supported by a post-doctoral fellowship awarded by the Alexander von Humboldt Foundation.

Like other embedded systems, NCSs are susceptible to both internal and external sources of *electromagnetic interference* (EMI), *e.g.*, spark plugs, TV towers, *etc.* [47]. In fact, the likelihood of soft errors due to EMI across a fleet of devices should not be underestimated. For example, Mancuso [41] observed that, assuming one soft error per bit in a 1 MB SRAM every 10^{12} hours of operation, and a worldwide population of 0.5 billion cars with an average daily operation time of 5%, about 5,000 vehicles per day are affected by a soft error.

Since unmitigated soft errors can result in potentially catastrophic system failures, EMI-induced error scenarios are anticipated in the design of safety-critical systems, and commonly mitigated by means of either active or passive replication. In the context of high-frequency control applications specifically, passive replication, *i.e.*, the use of hot/cold standbys, is insufficient if the failure detection and view-change latencies exceed the control frequency. System engineers thus devise active replication (or static redundancy) schemes to ensure that safety-critical NCSs are *fail-operational* (*e.g.*, see [22, 15, 29]).

However, coming up with a good active replication scheme is no easy task. Engineers face many questions, such as which components, if made more or less resilient (*e.g.*, by adding an extra replica, or shielding), will most impact the overall reliability? Alternatively, which components could be replaced with cheaper consumer-grade parts with the least effect on system reliability? Would dual modular redundancy suffice if the control logic is robust to, say, 10% message loss or would triple modular redundancy be needed? In general, such questions (and many more like them) do not have obvious answers, and particularly not if size, weight, and power (SWaP) as well as cost constraints must be taken into account, too.

The challenge is further exacerbated by the fact that commercially-used controllers are typically safeguarded against disturbances and noise using appropriate limiting or clamping mechanisms, and most well-designed control systems are inherently robust to a few failed iterations, *e.g.*, one missed actuation does not crash an inverted pendulum. That is, requiring that *all* control loop iterations must be correct and timely – *i.e.*, completely unaffected by soft errors – forces excessively pessimistic answers relative to the "true" needs of the workload, and consequently results in under-utilized, cost-inefficient systems. Thus, to appropriately dimension a fail-operational real-time NCS, a robustness-aware reliability analysis is required.

In this paper, we present a sound reliability analysis that evaluates a given configuration of an actively replicated NCS and quantifies its resiliency to EMI-induced transient errors, including message omission errors due to host crashes, incorrect computation errors due to memory corruption, and deadline violations due to retransmission delays. The objective is to provide system engineers with a sound method to evaluate (*i.e.*, safely bound) the reliability of an active replication scheme (*i.e.*, for a given number of replicas for each task in the NCS) assuming peak failure rates are known from empirical measurements and/or environmental modeling. We consider NCSs that are networked using a broadcast medium such as CAN (or Ethernet with a reliable broadcast primitive implemented on top) and evaluate them at the granularity of message exchanges between the distributed components.

Unlike traditional solutions based on hard real-time assumptions, our analysis leverages the robustness of well-designed control systems: since robust control loops tolerate a limited number of transient failures (which result in degraded control performance, but not an unrecoverable plant state), we characterize control loops with (m, k)-firm specifications, where out of every k consecutive control loop iterations, at least m must be "correct and timely" [27]. Blind and Allgöwer [9] have shown that the (m, k)-firm model is strictly stronger than the classical *asymptotic* requirement for control robustness (*e.g.*, as recently studied by Saha *et al.* [52]), which mandates that, as the number of control loop iterations approaches infinity, the failure rate should not exceed a given threshold. We thus use this model to bound the *failures in time* (FIT) of an NCS, *i.e.*, the expected number of *control failures* in one billion operating hours, where control failure denotes a violation of the (m, k)-firm constraint.

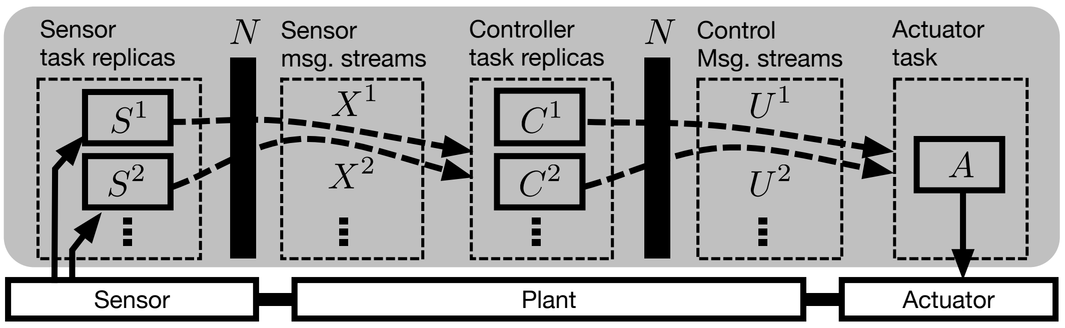


Figure 1 An FT-SISO control loop. Solid boxes denote hosts. Each dashed box denotes a task replica set or a set of message streams transmitted by a task replica set. Dashed arrows denote message streams broadcasted over the shared network N, *e.g.*, X^1 and X^2 are received by all tasks in C.

The proposed analysis consists of three steps. Given a model of a fault-tolerant single-input single-output (FT-SISO) control loop with active replication of its critical tasks (§2), the program-visible effects of EMI are first classified as crashes (resulting in message omissions), memory corruption (resulting in incorrect messages), and message retransmissions (resulting in deadline violations), and each of these errors is modeled probabilistically (§3). Second, an intermediate analysis (§4) then relates the probability of individual message errors to that of a *failed iteration* of a control loop, *i.e.*, where the controlled plant is not actuated as expected in an error-free iteration.[2] Finally, a reliability analysis upper-bounds the FIT of an NCS, which may consist of one or more FT-SISO control loops, as a function of the control loops' respective (m, k)-firm specifications.

We have evaluated the proposed analysis with a case study exploring replication options for a CAN-based active suspension workload (§5). Our results show that analysis and simulation results closely track each other when configuration parameters are varied. We also demonstrate how the analysis can help in identifying non-obvious reliability tradeoffs, and identify the underlying timing analysis of the CAN bus [16] as the single greatest individual source of pessimism in our analysis due to its reliance on a *critical instant* that occurs only rarely.

2 System Model

We consider an FT-SISO networked control loop L deployed on hosts $H = \{H_1, H_2, \ldots\}$ connected by a broadcast medium N, which is shared with other traffic as well, *e.g.*, other control loops, the clock synchronization protocol, *etc.* A block diagram is shown in Fig. 1.

The sensor task replicas $S = \{S^1, S^2, \ldots\}$ periodically generate sensor output and broadcast it over N. As a convention, we let superscripts denote replica IDs. We let X^i denote the message stream carrying the sensor values of the i^{th} replica of the sensor task, and let $X = \{X^1, X^2, \ldots\}$ denote the set of all such message streams.

The controller task replicas $C = \{C^1, C^2, \ldots\}$, upon periodic activation, read the latest received sensor messages, compute a new control command for the plant, update their local states (*e.g.*, in a PID controller, the integrator), and broadcast the control command. They are assigned appropriate offsets to ensure that, in an error-free execution, the sensor messages are available before any controller task replicas are activated. The message streams carrying control commands are denoted $U = \{U^1, U^2, \ldots\}$.

[2] Note the difference between a *failed iteration* of a control loop and *control failure*. A failed iteration is simply a deviation from an ideal, error-free scenario. Multiple failed iterations may lead to control failure if they violate the control loop's (m, k)-firm specification.

Algorithm 1 Voting procedure before the i^{th} activation of any controller task. The voting procedure for the actuator task is defined similarly by replacing the input set X_i with U_i.

```
1: procedure PeriodicControllerTaskActivation
2:     $Latest_i \leftarrow \emptyset$                                  ▷ start voting protocol
3:     for all $X_i^k \in X_i$ do
4:         if $X_i^k$ not received by its deadline then
5:             continue                                          ▷ also accounts for omissions
6:         $Latest_i \leftarrow Latest_i \cup X_i^k$
7:     if $Latest_i = \emptyset$ then return                               ▷ omit output
8:     $result_i \leftarrow SimpleMajority(Latest_i)$           ▷ break ties based on message IDs
9:     ...                                                       ▷ main logic of the task starts
```

The actuator task A is directly connected to the plant. Upon periodic activation, it reads the latest received control commands and actuates the plant accordingly. Like the controller tasks, A is also assigned an appropriate offset to ensure that, in an error-free execution, all control commands are received before its activation. Unlike the sensor and controller tasks, the actuator task A is not replicated since it requires special hardware in the plant actuator to handle redundant inputs [29]. We revisit this issue in §7.

All tasks and messages in the control loop have a period of T time units. The i^{th} runtime activations or jobs of sensor task replicas in $S = \{S^1, S^2, \ldots\}$ and controller task replicas in $C = \{C^1, C^2, \ldots\}$ are denoted $S_i = \{S_i^1, S_i^2, \ldots\}$ and $C_i = \{C_i^1, C_i^2, \ldots\}$, respectively; and the i^{th} job of actuator task A is denoted A_i. Similarly, the i^{th} messages in sensor message streams $X = \{X^1, X^2, \ldots\}$ and controller message streams $U = \{U^1, U^2, \ldots\}$ are denoted $X_i = \{X_i^1, X_i^2, \ldots\}$ and $U_i = \{U_i^1, U_i^2, \ldots\}$, respectively.

Finally, we let $\mathcal{U}_i$ denote the actuator command applied to the physical plant in the i^{th} iteration, *i.e.*, output of job A_i, and let $\mathcal{U} = \{\mathcal{U}_1, \mathcal{U}_2, \ldots\}$ denote the ordered set of such commands applied to the physical plant across all iterations.

Assumptions. We assume that tasks resolve redundant inputs at the start of every iteration through voting (Algorithm 1). We let $V_i = \{V_i^1, V_i^2, \ldots\}$ denote the set of voter instances that resolve the redundant inputs for controller jobs $C_i = \{C_i^1, C_i^2, \ldots\}$, respectively, and let V_i^A denote the voter instance that resolves the redundant inputs for the actuator job A_i. Since all inputs are available before the task is activated in an error-free scenario, message streams that are delayed or omitted due to transmission or crash errors are ignored during voting (Line 5 of Algorithm 1). In the worst case, if no input is available on time to the voter due to errors, the task's activation is skipped, *i.e.*, the task's output for that iteration is omitted (Line 7). While computing the simple majority (Line 8), any ties in quorum size are broken deterministically using message IDs.

We (pessimistically) assume that corrupted message replicas are identical because it is a worst-case scenario w.r.t. the voting protocol. In particular, if the number of corrupted messages exceeds the number of correct messages, then assuming identically corrupted messages implies that the voting outcome is corrupted, while in the case of non-identically corrupted messages there is a high likelihood that correct messages still form the largest quorum. In practice though, whether or not corrupted messages are likely to be identical is highly system- and application-specific. Random EMI normally does not cause identically corrupted patterns and many systems use end-to-end checksums; the likelihood of identically corrupted messages is thus small. In contrast, if the application payload is of boolean type or encoded using only a few bits, the likelihood of identically corrupted messages is non-negligible.

Furthermore, we assume that NCS hosts are synchronized using a clock synchronization protocol (such as the Precision Time Protocol [1]), and that task and message offsets have been chosen to account for the maximum clock synchronization error. Without this assumption, it is much more challenging to ensure replica determinism (*e.g.*, simply assigning appropriate offsets to tasks and messages is insufficient) [48]. We also require that all NCS tasks are deterministic. Thus, given identical inputs and identical states, any two sensor (controller) task replicas produce identical sensor (control) messages, unless one is affected by memory corruption.

3 Fault Model

To lay the foundation for our analysis, we first give a precise fault model.

We model the EMI-induced *raw transient faults*, *i.e.*, bit-flips on the network and in host memory, as random events following a Poisson distribution. Let $\mathcal{P}(x, \delta, \lambda)$ denote the *probability mass function* of the Poisson distribution, *i.e.*, the probability that x independent events occur in an interval of length δ when the arrival rate is λ. Let τ and λ_i denote the peak rate of raw transient faults affecting the network and each host $H_i \in H$, respectively. We define the probability that x raw transient faults affect the network (respectively, host H_i) in any interval of length δ as $\mathcal{P}(x, \delta, \tau)$ (respectively, $\mathcal{P}(x, \delta, \lambda_i)$).

In practice, the peak fault rates are empirically determined with measurements or derived from environmental modeling assuming worst-possible operating conditions, and typically include safety margins as deemed appropriate by reliability engineers or domain experts. As a result, a Poisson process is a good *approximation* of the worst-case scenario, as previously discussed by Broster *et al.* [13]. For instance, in the case of network faults, τ is likely to exceed any transient actual fault rate τ_{actual} experienced in practice, which also varies over time and/or based on a system's current surroundings. Thus, as per the Poisson model, while the *actual* probability that the network experiences at least one transient fault in any interval of length δ is given by $\sum_{x>0} \mathcal{P}(x, \delta, \tau_{actual})$, we upper-bound this probability in our analysis by $\sum_{x>0} \mathcal{P}(x, \delta, \tau)$. That is, if $\tau > \tau_{actual}$, then $\sum_{x>0} \mathcal{P}(x, \delta, \tau) > \sum_{x>0} \mathcal{P}(x, \delta, \tau_{actual})$.[3]

Raw transient faults may manifest as program-visible retransmission, crash, and incorrect computation errors [8, 6], which are also modeled probabilistically, as described below.

Networking protocols incorporate explicit mechanisms to mitigate the effects of transient faults on the wire, *e.g.*, error detection and correction in CAN [44]. Thus, we assume that network message corruptions are always detected, but may result in retransmission errors which may eventually lead to deadline violations. As in [12], we make the simplifying (but safe) assumption that every transient fault on the network causes a retransmission. Thus, we define the retransmission rate as τ, and the probability that x retransmissions occur in any interval of length δ as $\mathcal{P}(x, \delta, \tau)$. Given this, an upper bound on the probability that a message misses its deadline can be derived using prior work [13, 56].

In this work, we assume that an upper bound on the worst-case deadline-miss probability of any message instance belonging to any sensor message stream X^x or any control message stream U_x is known and denote this bound as $B(X^x)$ or $B(U^x)$, respectively.

Crash errors occur if the system suffers an EMI-induced corruption that causes an exception to be raised and the system to be rebooted, or that induces an unbounded hang that causes the system's watchdog timer to trigger a reboot, *e.g.*, see [43]. A crashed system remains unavailable for some time while it reboots and thus causes an interval in which

[3] This basic fact can be proved by representing the *cumulative density function* of the Poisson distribution in the form of an *upper incomplete gamma function* [5].

messages are continuously omitted. We assume that the recovery interval on each host H_i is upper-bounded by R_i, which we assume also includes any delays that arise due to the need to resynchronize any application state after a crash.

Prior studies have shown that a large fraction of transient faults have no negative effects [60, 7, 3]. We thus assume a *derating factor* that accounts for masked transient faults, which can be determined empirically [42]. Let f_i denote the derating factor for crash errors on host H_i; the peak rate of crash errors on host H_i is then given by $\rho_i = f_i \lambda_i$. Using the peak crash error rate, we model crash errors like raw transient faults as random events following a Poisson distribution. Thus, we define the probability that x crash errors occur on host H_i in any interval of length δ as $\mathcal{P}(x, \delta, \rho_i)$.

Incorrect computation errors may occur if a message is corrupted before transmission (during preparation), before the network controller computes a checksum for subsequent error detection. Like crash errors, assuming a host-specific *derating factor* f'_i for incorrect computation errors, the average error rate on host H_i is given by $\kappa_i = f'_i \lambda_i$ and the probability that x errors occur in any interval of length δ is given by $\mathcal{P}(x, \delta, \kappa_i)$.[4] Our notion of incorrect computation errors does not refer to software bugs or Byzantine errors.

We refer to the interval during which a message is at risk of corruption as its *exposure interval.* For *stateful* tasks such as a PID controller, the message computation relies on both the current input and the application state, and the latter could be affected by *latent faults* (*i.e.*, state corruptions that have not yet been detected). Thus, the exposure interval of a message depends on the mechanisms in place to tolerate (or avoid) latent faults.

If the hardware platform uses *Error-Correcting Code* (ECC) memory and processors with *lockstep* execution (common in safety-critical systems), then the built-in protections suppress latent faults, and it suffices to consider the *scheduling window* of a message (*i.e.*, the duration from the message's creation to its deadline) as its exposure interval. If no such architectural support is available, then any relevant state can be protected with a *data integrity checker* task that periodically verifies the checksums of all relevant data structures (and that reboots the system in the case of any mismatch). The exposure interval of a message then includes its scheduling window and (in the worst case) an entire period of the data integrity checker.

We assume that the worst-case exposure interval for each message in X^x, U^y, and $\mathcal{U}$ is known in advance and denote it using $E(X^x)$, $E(U^y)$, and $E(\mathcal{U})$, respectively.

Assumptions. Based on the stochastic nature of physical EMI processes, we consider EMI-induced transient faults, and hence basic message errors, to be independent. We do however account explicitly for correlated errors that arise from the system architecture, *e.g.*, deterministic replicas will produce the same wrong output if given the same wrong input.

We also implicitly account for correlated surges in error rates across all components since we analyze peak rates for all components. For example, if a UAV with an FT-SISO control loop is flying through a strong radar beam, all replicas of the control loop simultaneously experience increased rates of EMI. The proposed analysis is able to handle this correlation because the derived upper bound on the failure rate is monotonic in all fault rates and applied assuming *peak* fault rates, which in turn are determined such that they exceed the fault rates expected in practice, especially during such high interference scenarios.

[4] The choice of Poisson distribution for modeling both crash and incorrect computation errors is reasonable since real-time tasks are repeated, short workloads; thus, any generated message is equally likely to be affected by an error, and a host is equally likely to be crashed during any iteration of the task (see [37] for a mathematical basis for this argument).

While evaluating the EMI-induced errors discussed above, we assume that other system components are reliable, even though the NCS subsystem being analyzed may directly depend on them, *e.g.*, the power sources, the physical sensors and the actuators, the controlled physical plant, or the clock synchronization mechanism. This assumption does not imply that the proposed analysis is not useful if a dependent component fails, rather it provides a FIT rate for one subsystem, which can then be composed with the FITs of other dependent, dependee, or unrelated subsystems, *e.g.*, using a fault tree analysis. This is a common way of decomposing the reliability analysis of the whole system into manageable components.

We also assume that the network protocol guarantees atomic broadcast, *i.e.*, messages are received consistently by either all hosts, or none. While Byzantine error scenarios violate this assumption, *e.g.*, [39], they occur with such low likelihood that they are best modeled as a separate, additive failure source and accounted for using a separate FIT analysis.

Finally, recall from §2 that tasks are assigned appropriate offsets to ensure sequentiality, *e.g.*, to ensure that sensor values are always available (in an error-free execution) before any control task replica is activated. In this work, we assume that processor scheduling on each host is statically checked and thus task offsets are correctly enforced. Alternatively, processor scheduling delays due to transient faults could be explicitly taken into account as an additional source of failed control loop iterations, *e.g.*, when upper-bounding the probability of a message omission (see Definition 1 in §4).

4 Probabilistic Analysis

We analyze the probability that the n^{th} iteration of the control loop fails, for any n.

As mentioned in §2, due to clock synchronization and the atomic broadcast assumption, message replicas are identical in an error-free scenario, *i.e.*, the messages in X_n carry identical sensor values and the messages in U_n carry identical control commands. However, due to incorrect computation errors, one or more messages in X_n may be corrupted. If the voters V_n choose a corrupted sensor value, then all messages U_n carrying the control commands are also corrupted. Messages in X_n could also be delayed or omitted due to transmission and crash failures, in which case the voters V_n work with fewer inputs. But if all the messages in X_n are either delayed or omitted, the controller jobs C_n have no inputs to work with, hence the messages U_n are not prepared. Similarly, the controller to actuator information flow may also be affected by errors, resulting in A_n's output $\mathcal{U}_n$ being corrupted or omitted. These dependencies are illustrated using an example in Fig. 2.

Based on this intuition, we next bound the probability that the final output $\mathcal{U}_n$ is corrupted or omitted, in a bottom-up fashion and in small steps of a few lemmas each. We use $P(\cdot)$ to denote *exact* probabilities and $Q(\cdot)$ to denote upper bounds on the exact probabilities.

In particular, we first define the analysis as a function of the following exact (but unknown) probabilities for each message m:

▶ **Definition 1.** $P(m\ \textit{omitted})$ denotes the exact probability of an omission. $P(m\ \textit{delayed})$ denotes the exact probability of a deadline violation. $P(m\ \textit{corrupted})$ denotes the exact probability of an incorrect computation.

In addition, since the effect of message corruption on Algorithm 1's output also depends on the application-specific message payload, the analysis initially also assumes the following exact (but unknown) probability.

▶ **Definition 2.** $P(\textit{Majority incorrect} \mid \mathcal{I}, \mathcal{C})$ denotes the exact probability that, given a set of incorrect inputs $\mathcal{I}$ and correct inputs $\mathcal{C}$, the *SimpleMajority*$(\mathcal{I} \cup \mathcal{C})$ procedure in Algorithm 1 (Line 8) outputs an incorrect value.

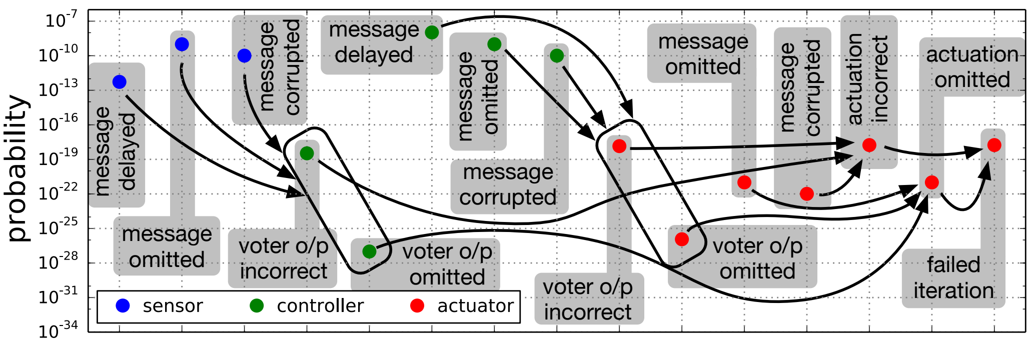


Figure 2 Error probabilities at different stages of a CAN-based wheel control loop (see §5 for details). Arrows denote dependencies among error probabilities of the different control loop stages. The error rates (per ms) are $\tau = 10^{-4}$ for the CAN bus, $\rho_i = 10^{-12}$ and $\kappa_i = 10^{-12}$ for each H_i hosting sensor and controller tasks, and $\rho_a = 10^{-24}$ and $\kappa_a = 10^{-24}$ for the actuator task's host H_a.

In each step of the analysis, we ensure that the derived probability is either independent of, or increasing in, these exact error probabilities. Thus, when instantiating the analysis using upper bounds on the exact probabilities, we implicitly guarantee that the *derived* iteration failure probability upper-bounds the *actual* iteration failure probability. Due to space constraints, we do not give a proof of monotonicity in this paper. We revisit the issue at relevant places where we explicitly add some pessimism to the analysis to ensure monotonicity.

Step 1. Analyzing the correctness of V_n^y's output. We evaluate the probability that a controller voter instance V_n^y outputs an incorrect value because of corrupted inputs.

Recall from §2 that X_n denotes the set of all sensor message replicas that are inputs to V_n^y. Let $\mathcal{T}_n = \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle$ denote a 5-tuple constrained by the following definition.

▶ **Definition 3.** $\mathcal{T}_n = \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle$ is *valid* if $\mathcal{O}_n$, $\mathcal{D}_n$, $\mathcal{I}_n$, $\mathcal{C}_n$, and $\mathcal{Z}_n$ *partition* set X_n: messages in $\mathcal{O}_n$ are omitted; messages in $\mathcal{D}_n$ are not omitted, but delayed due to retransmissions; messages in $\mathcal{I}_n$ are neither omitted nor delayed, but are incorrectly computed; messages in $\mathcal{C}_n$ are neither omitted, delayed, nor incorrectly computed; and messages in $\mathcal{Z}_n$ *may* be omitted, delayed, or corrupted.

In general, $\mathcal{Z}_n$ denotes the messages whose *fate is undecided*, or in other words, each message $X_n^y \in \mathcal{Z}_n$ may still be omitted with probability $P(X_n^y \text{ } \textit{omitted})$, delayed with probability $P(X_n^y \text{ } \textit{delayed})$, and incorrectly computed with probability $P(X_n^y \text{ } \textit{corrupted})$. Thus, if message $X_n^y \in X_n$ is guaranteed to be omitted due to host crashes, then $X_n^y \in \mathcal{O}_n$. Similarly, if X_n^y is guaranteed to be transmitted on time and without being incorrectly computed due to host corruptions, then $X_n^y \in \mathcal{C}_n$.

Based on Definitions 1–3, we use the following recursive expression to compute the probability that V_n^y outputs an incorrect value because the majority of its inputs is corrupted.

$$P\left(\begin{array}{c} V_n^y \text{ output} \\ \text{incorrect} \end{array} \middle| \begin{array}{c} \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \\ \mathcal{C}_n, \mathcal{Z}_n \rangle \end{array} \right) = \begin{cases} P(\textit{Majority incorrect} \mid \mathcal{I}_n, \mathcal{C}_n) & \mathcal{Z}_n = \emptyset \\ \Gamma(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle) & \mathcal{Z}_n \neq \emptyset \end{cases} \quad (1)$$

where $\Gamma(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle) =$

$$\left(\begin{array}{l} P(X_n^s \text{ omitted}) \\ \times P(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n \cup \{X_n^s\}, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\} \rangle) \end{array} \right) +$$
$$\left(\begin{array}{l} (1 - P(X_n^s \text{ omitted})) \times P(X_n^s \text{ delayed}) \\ \times P(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n \cup \{X_n^s\}, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\} \rangle) \end{array} \right) +$$

$$\left(\begin{array}{l}(1-P(X_n^s \text{ omitted})) \times (1-P(X_n^s \text{ delayed})) \times P(X_n^s \text{ corrupted}) \\ \times P(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n \cup \{X_n^s\}, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right) + \\ \left(\begin{array}{l}(1-P(X_n^s \text{ omitted})) \times (1-P(X_n^s \text{ delayed})) \times (1-P(X_n^s \text{ corrupted})) \\ \times P(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n \cup \{X_n^s\}, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right)$$

and X_n^s denotes the message with the smallest ID in $\mathcal{Z}_n$ (if $\mathcal{Z}_n \neq \emptyset$).

In each step of the recursion, a single message $X_n^s \in \mathcal{Z}_n$ is either **(i)** omitted with probability $P(X_n^y\ \textit{omitted})$ and inserted into set $\mathcal{O}_n$; **(ii)** not omitted but delayed with probability $(1 - P(X_n^y\ \textit{omitted})) \cdot P(X_n^y\ \textit{delayed})$ and inserted into set $\mathcal{D}_n$; **(iii)** transmitted on time, *i.e.*, neither omitted nor delayed, but is incorrectly computed with probability $(1 - P(X_n^y\ \textit{omitted})) \cdot (1 - P(X_n^y\ \textit{delayed})) \cdot P(X_n^y\ \textit{corrupted})$ and inserted into set $\mathcal{I}_n$; or **(iv)** transmitted on time and without any corruptions with probability $(1 - P(X_n^y\ \textit{omitted})) \cdot (1 - P(X_n^y\ \textit{delayed})) \cdot (1 - P(X_n^y\ \textit{corrupted}))$, and thus inserted into set $\mathcal{C}_n$. The recursion terminates when all cases have been exhaustively enumerated, *i.e.*, when $\mathcal{Z}_n = \emptyset$ and $\mathcal{O}_n \cup \mathcal{D}_n \cup \mathcal{I}_n \cup \mathcal{C}_n = X_n$.

Therefore, $P(V_n^y\ \textit{output incorrect} \mid \langle \emptyset, \emptyset, \emptyset, \emptyset, X_n\rangle)$, as defined in Eq. 1, computes the exact probability that controller voter instance V_n^y outputs an incorrect value.

However, Eq. 1 is not monotonically increasing in the omission and delay probabilities, as required. Its monotonicity in $P(X_n^s\ \textit{omitted})$ and $P(X_n^s\ \textit{delayed})$ depends on $P(X_n^s\ \textit{corrupted})$. This is because the overall failure probability could be reduced by simply delaying or omitting a message, if that message is likely to be incorrectly computed and thus has the potential to tilt the voting outcome in favor of an incorrect quorum.

To remove this dependency on $P(X_n^s\ \textit{corrupted})$, we replace $\Gamma(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n\rangle)$ in Eq. 1 with a slightly pessimistic term $\Gamma_\pi(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n\rangle)$ (notice the fifth term in the definition of $\Gamma_\pi(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n\rangle)$), and define an upper bound (stated below) on the probability that controller voter instance V_n^y outputs an incorrect value.[5]

$$Q\left(\begin{array}{c} V_n^y \text{ output} \\ \text{incorrect}\end{array} \middle| \begin{array}{c}\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \\ \mathcal{C}_n, \mathcal{Z}_n\rangle\end{array}\right) = \begin{cases} P(\textit{Majority incorrect} \mid \mathcal{I}_n, \mathcal{C}_n) & \mathcal{Z}_n = \emptyset \\ \Gamma_\pi(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n\rangle) & \mathcal{Z}_n \neq \emptyset \end{cases} \tag{2}$$

where $\Gamma_\pi(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n\rangle) =$

$$\left(\begin{array}{l} \left(\begin{array}{l} P(X_n^s \text{ omitted}) \\ \times Q(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n \cup \{X_n^s\}, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right) + \\ \left(\begin{array}{l} (1-P(X_n^s \text{ omitted})) \times P(X_n^s \text{ delayed}) \\ \times Q(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n \cup \{X_n^s\}, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right) + \\ \left(\begin{array}{l} (1-P(X_n^s \text{ omitted})) \times (1-P(X_n^s \text{ delayed})) \times P(X_n^s \text{ corrupted}) \\ \times Q(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n \cup \{X_n^s\}, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right) + \\ \left(\begin{array}{l} (1-P(X_n^s \text{ omitted})) \times (1-P(X_n^s \text{ delayed})) \times (1-P(X_n^s \text{ corrupted})) \\ \times Q(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n \cup \{X_n^s\}, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right) + \\ \left(\begin{array}{l} \left(\begin{array}{l} (1-P(X_n^s \text{ omitted})) \times P(X_n^s \text{ delayed}) \times P(X_n^s \text{ corrupted}) + \\ P(X_n^s \text{ omitted}) \times P(X_n^s \text{ corrupted})\end{array}\right) \\ \times Q(V_n^y \text{ output incorrect} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n \cup \{X_n^s\}, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\}\rangle)\end{array}\right) + \end{array}\right)$$

and X_n^s denotes the message with the smallest ID in $\mathcal{Z}_n$ (if $\mathcal{Z}_n \neq \emptyset$).

$Q(V_n^y\ \textit{output incorrect} \mid \langle \emptyset, \emptyset, \emptyset, \emptyset, X_n\rangle)$ thus yields an upper bound on the probability that voter instance V_n^y outputs an incorrect value. For convenience, we let $Q(V_n^y\ \textit{output incorrect})$ $= Q(V_n^y\ \textit{output incorrect} \mid \langle \emptyset, \emptyset, \emptyset, \emptyset, X_n\rangle)$ in the following.

[5] See the appendix in the extended version of the paper [25] for a proof of monotonicity of Eq. 2.

Step 2. Analyzing whether V_n^y omits its output. We evaluate the probability that a controller voter instance V_n^y omits its output because all its inputs were either delayed or omitted, *i.e.*, the special case in Algorithm 1 (Line 7). Once again, we state a recursive expression to compute the probability, similar to the one used in Step 1.

$$P\left(\begin{array}{c} V_n^y \text{ output} \\ \text{omitted} \end{array} \middle| \begin{array}{c} \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \\ \mathcal{C}_n, \mathcal{Z}_n \end{array} \right) = \begin{cases} \Lambda\Big(\ \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle\ \Big) & \mathcal{Z}_n \neq \emptyset \\ 1 & \mathcal{I}_n \cup \mathcal{C}_n = \emptyset \\ 0 & \mathcal{I}_n \cup \mathcal{C}_n \neq \emptyset \end{cases} \tag{3}$$

where $\Lambda(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle) =$

$$\left(\begin{array}{l} \left(\begin{array}{l} P(X_n^s \text{ omitted}) \\ \times P(V_n^y \text{ output omitted} \mid \langle \mathcal{O}_n \cup \{X_n^s\}, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\} \rangle) \end{array} \right) + \\ \left(\begin{array}{l} (1 - P(X_n^s \text{ omitted})) \times P(X_n^s \text{ delayed}) \\ \times P(V_n^y \text{ output omitted} \mid \langle \mathcal{O}_n, \mathcal{D}_n \cup \{X_n^s\}, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\} \rangle) \end{array} \right) + \\ \left(\begin{array}{l} (1 - P(X_n^s \text{ omitted})) \times (1 - P(X_n^s \text{ delayed})) \times P(X_n^s \text{ corrupted}) \\ \times P(V_n^y \text{ output omitted} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n \cup \{X_n^s\}, \mathcal{C}_n, \mathcal{Z}_n \setminus \{X_n^s\} \rangle) \end{array} \right) + \\ \left(\begin{array}{l} (1 - P(X_n^s \text{ omitted})) \times (1 - P(X_n^s \text{ delayed})) \times (1 - P(X_n^s \text{ corrupted})) \\ \times P(V_n^y \text{ output omitted} \mid \langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n \cup \{X_n^s\}, \mathcal{Z}_n \setminus \{X_n^s\} \rangle) \end{array} \right) \end{array} \right)$$

and X_n^s denotes the message with the smallest ID in $\mathcal{Z}_n$ (if $\mathcal{Z}_n \neq \emptyset$).

$P(V_n^y$ *output omitted* $\mid \langle \emptyset, \emptyset, \emptyset, \emptyset, \mathcal{X}_n \rangle)$ thus yields the exact probability that voter instance V_n^y omits its output. Note that Eq. 3 does not depend on the correctness of V_n^y's inputs, but only on the timeliness of its inputs, unlike the simple majority procedure in Eq. 1. Hence, Eq. 3's monotonicity in $P(X_n^s$ *omitted*) and $P(X_n^s$ *delayed*) does not depend on $P(X_n^s$ *corrupted*), unlike Eq. 1. As a result, the use of a pessimistic term such as $\Gamma_\pi(\langle \mathcal{O}_n, \mathcal{D}_n, \mathcal{I}_n, \mathcal{C}_n, \mathcal{Z}_n \rangle)$ in Eq. 2 is not required in this case.

For convenience, we let $P(V_n^y$ *output omitted*$) = P(V_n^y$ *output omitted* $\mid \langle \emptyset, \emptyset, \emptyset, \emptyset, \mathcal{X}_n \rangle)$.

Step 3: Analyzing the actuator voter instance V_n^A. We bound the probability that V_n^A outputs an incorrect value, because the majority of its inputs is corrupted, or that it does not choose anything, because all its inputs are either omitted or delayed.

Since all controller voter instances V_n operate on the same input values, if a correct voter instance V_n^y outputs an incorrect value because of wrong inputs, it implies that all correct voter instances in V_n output incorrect values. In such a scenario, the actuator voter V_n^A is guaranteed to get only incorrect control messages, since all of the control messages will be prepared using the corrupted sensor values.

A similar property holds for the controller voter output omission. Proper deadline and offset assignment guarantees that, in an error-free scenario, messages in X_n are transmitted before the voter instances in V_n are activated. Thus, each voter instance can decide locally whether a message was received past its deadline (in which case it is discarded, recall Algorithm 1). As a result, if a controller voter instance V_n^y does not choose any value because all its inputs are delayed or omitted, then all controller voter instances in V_n do not choose any values, either. Thus, no output is generated by the controller task replicas and the actuator voter omits its output, too, which results in a skipped actuation.

Let $Q(V_n^A$ *output incorrect*) denote an upper bound on the probability that voter instance V_n^A's outputs an incorrect value, conditioned on the assumption that the sensor inputs of the controller voter instances V_n did not result in a corrupted output. Similarly, let $P(V_n^A$ *output omitted*) denote the probability that voter instance V_n^A's output is omitted, conditioned on the assumption that the sensor inputs of the controller voter instances V_n did not result in an omitted output. Both $Q(V_n^A \textit{ output incorrect})$ and $P(V_n^A \textit{ output omitted})$ can be derived using the recursive procedures discussed in Steps 1 and 2, respectively, by replacing the set of voter inputs X_n with U_n (recall from §2 that U_n denotes the set of all inputs to V_n^A). The case that the sensor inputs of the controller voter instances V_n result in a corrupted or omitted output is accounted for in Step 5.

Step 4: Analyzing the final output $\mathcal{U}_n$. We first bound the probability that the actuation during the n^{th} control loop iteration is incorrect (Lemma 4), followed by the probability that it is omitted (Lemma 5), and finally the joint probability of both events (Lemma 6). For brevity, we let $\phi_{1} = Q(V_n^y \textit{ output incorrect})$, $\phi_{2a} = Q(V_n^A \textit{ output incorrect})$, $\phi_{2b} = P(\mathcal{U}_n \textit{ corrupted})$, $\omega_{1} = P(V_n^y \textit{ output omitted})$, $\omega_{2a} = P(V_n^A \textit{ output omitted})$, and $\omega_{2b} = P(\mathcal{U}_n \textit{ omitted})$.

▶ **Lemma 4.** *The probability that the actuation during the n^{th} control loop iteration is incorrect is at most $\phi_1(1 + \phi_{2a}\phi_{2b}) + \phi_{2a} + \phi_{2b}$.*

Proof. We consider two cases based on whether the sensor inputs to any voter instance V_n^y results in corruption of the controller voter outputs (case 1) or not (case 2). The probability that case 1 occurs is $\phi_{case1} = P(V_n^y \textit{ output incorrect})$. For this case, since the sensor inputs to voter instance V_n^y results in corruption of its output, voter instances in all controller tasks choose an incorrect output. Thus, all control commands transmitted were incorrect, thus it is guaranteed that the actuation during the n^{th} control loop iteration is incorrect. Thus, the conditional probability in this case is $\phi_{cond1} = 1$.

The probability that case 2 occurs is $\phi_{case2} = 1 - \phi_{case1}$. For this case, the conditional probability that the actuation during the n^{th} control loop iteration is incorrect depends on two sources: **(a)** voter instance V_n^A's output can be incorrect, and **(b)** A's host can be affected by incorrect computation errors. The probability for case (a) is $\phi_{case2a} = P(V_n^A \textit{ output incorrect})$. The probability for case (b) is $\phi_{case2b} = P(\mathcal{U}_n \textit{ corrupted})$. Cases (a) and (b) are independent: (a) occurs because inputs to V_n^A were corrupted due to incorrect computation errors on the controller tasks' hosts, whereas (b) occurs due to incorrect computation errors on the actuator task's host. Thus, using theorem $P(A_1 \cup A_2) = P(A_1) + P(A_2) - P(A_1) \cdot P(A_2)$ for independent events A_1 and A_2, the conditional probability for case 2 is $\phi_{cond2} = \phi_{case2a} + \phi_{case2b} - \phi_{case2a}\,\phi_{case2b}$.

By the law of total probability, the probability that the actuation during the n^{th} control loop iteration is incorrect is given by $\phi_{case1}\,\phi_{cond1} + \phi_{case2}\,\phi_{cond2}$. Upon expanding ϕ_{cond1}, ϕ_{cond2}, and ϕ_{case2}, and then rearranging the resulting expression w.r.t. ϕ_{case1}, we get

$$\phi_{case1}\,\phi_{cond1} + \phi_{case2}\,\phi_{cond2} = \left(\begin{array}{l} \phi_{case1} \times (1 - \phi_{case2a} - \phi_{case2b} + \phi_{case2a} \cdot \phi_{case2b}) \\ + \phi_{case2a} + \phi_{case2b} - \phi_{case2a} \cdot \phi_{case2b} \end{array} \right).$$

Further, upon dropping any negative terms for monotonicity, and since $\phi_{case1} \leq \phi_1$, $\phi_{case2a} \leq \phi_{2a}$, and $\phi_{case2b} = \phi_{2b}$, we have the following upper bound:

$$\phi_{case1}\,\phi_{cond1} + \phi_{case2}\,\phi_{cond2} \leq \phi_1 \times (1 + \phi_{2a} \cdot \phi_{2b}) + \phi_{2a} + \phi_{2b}.$$ ◀

▶ **Lemma 5.** *The probability that the actuation during the n^{th} control loop iteration is delayed or omitted is at most $\omega_1(1 + \omega_{2a}\omega_{2b}) + \omega_{2a} + \omega_{2b}$.*

The proof of Lemma 5 is analogous to that of Lemma 4 and is thus omitted. In Lemma 6, we compose the probabilities derived in Lemmas 4 and 5 to derive the probability that the n^{th} control loop iteration fails, *i.e.*, that the actuation during this iteration is either incorrect or delayed (or omitted). We do not assume that the probabilities derived in Lemmas 4 and 5 are independent, since it is possible that an omitted control message tilted the majority in favor of the correct quorum, thereby reducing the probability that the actuation is incorrect.

▶ **Lemma 6.** *The probability that the n^{th} control loop iteration fails is at most*

$$Q\left(n^{th} \textit{ control loop iteration fails}\right) = \left(\begin{array}{c} \phi_1(1+\phi_{2a}\phi_{2b}) + \phi_{2a} + \phi_{2b} + \\ \omega_1(1+\omega_{2a}\omega_{2b}) + \omega_{2a} + \omega_{2b} \end{array}\right). \tag{4}$$

Proof. Follows from Lemmas 4 and 5. ◀

In summary, Steps 1–4 account for all direct and indirect dependencies between the individual message error events and the final actuation of the controlled plant, and the derived $Q\left(n^{th} \textit{ control loop iteration fails}\right)$ automates propagation of the failure probability along this dependency tree. Although the analysis has exponential time complexity in the number of sensor message streams $|X_n|$ and the number of controller message streams $|U_n|$ (due to the branching recursions in Eqs. 2 and 3), since the number of replicas of any task is likely small, *i.e.*, typically under ten, the analysis can be quickly performed.

Upper-bounding the failure probability. Since exact message error probabilities are impossible to obtain, we instantiate the above analysis with upper bounds on the exact probabilities. The analysis is monotonically increasing in the message error probabilities, and thus remains sound despite the use of these upper bounds. We next define upper bounds on the message error probabilities for any sensor message X_n^y. The bounds for any control message U_n^y and actuator task's output message $\mathcal{U}_n$ are analogously defined.

The probability that any sensor message X_n^y is delayed beyond its deadline is bounded by $P(X_n^y \textit{ delayed}) \leq B(X^y)$ (as defined in §3). Let the host on which X_n^y's sender task is deployed be denoted H_a. Regarding message omission, suppose X_n^y is expected to be scheduled for transmission at the earliest by time t and at the latest by time $t+J$ (where J denotes the maximum release jitter of the message). Since R_a is the maximum time to recover from a crash error on host H_a, if there is at least one crash error during the interval $[t-R_a,\ t+J)$, X_n^y's arrival may be skipped. Thus, $P(X_n^y \textit{ omitted}) \leq \sum_{x>0} \mathcal{P}(x,\ R_a+J,\ \rho_a)$. Regarding message corruptions due to incorrect computation errors, recall from §2 that the exposure interval for sensor message X_n^y is upper-bounded by $E(X^y)$. Thus, X_n^y may be corrupted if there is at least one incorrect computation error in this interval. Thus, $P(X_n^y \textit{ corrupted}) \leq \sum_{x>0} \mathcal{P}(x,\ E(X^y),\ \kappa_a)$.

The probability $P(\textit{SimpleMajority incorrect} \mid \mathcal{I},\ \mathcal{C})$ is upper-bounded by making the worst-case assumption that incorrect inputs in $\mathcal{I}$ are identically faulty. Recall from Definition 2 that $\mathcal{C}$ and $\mathcal{I}$ denote the sets of correct and incorrect inputs, respectively, to the *SimpleMajority*$(\mathcal{I} \cup \mathcal{C})$ procedure in Algorithm 1. Assuming $n_c = |\mathcal{C}|$, $n_i = |\mathcal{I}|$, and that $s_0 \in \mathcal{C} \cup \mathcal{I}$ denotes the message in $\mathcal{C} \cup \mathcal{I}$ with the smallest ID, we obtain the following bound.

$$Q\left(\begin{array}{c} \textit{SimpleMajority} \\ \text{incorrect} \end{array} \middle|\, \mathcal{I},\ \mathcal{C}\right) = \begin{cases} 1 & (n_i > n_c) \vee (n_i = n_c \neq 0 \wedge s_0 \in \mathcal{I}) \\ 0 & n_i = n_c \neq 0 \wedge s_0 \in \mathcal{C} \\ 0 & n_i < n_c \vee n_i = n_c = 0 \end{cases} \tag{5}$$

▶ **Lemma 7.** *Eq. 5 upper-bounds the probability that procedure SimpleMajority$(\mathcal{I} \cup \mathcal{C})$'s output in Algorithm 1 (Line 8) is incorrect.*

Proof. If $n_i > n_c$, the largest-sized quorum belongs to incorrect messages, and Algorithm 1's output is incorrect with probability 1. If $n_i = n_c \neq 0$, there are two largest-sized quorums. If message s_0 with the smallest ID is incorrect ($s_0 \in \mathcal{I}$), Algorithm 1 chooses an incorrect output with probability 1. Otherwise ($s_0 \in \mathcal{C}$), it chooses an incorrect output with probability 0. If $n_i < n_c$, the largest-sized quorum belongs to correct messages, and Algorithm 1's output is correct, *i.e.*, incorrect with probability 0. If $n_i = n_c = 0$, the voter has received no inputs, so the probability of choosing an incorrect output is 0. ◀

The IID property. Since each of the upper bounds defined above is independent of n, Eq. 4 can be iteratively unfolded until it consists only of terms that are independent of n. The bound is thus identical for any control loop iteration. In addition, the upper bounds are derived under worst-case assumptions with respect to interference from other messages on the network [13, 16]; and failure of the n^{th} control loop iteration, defined as a deviation from an error-free execution of that iteration, is independent of whether past iterations encountered any failures or not. Thus, the bounds obtained using Eq. 4 for any two iterations n_1 and n_2 are mutually independent as well. As a result, when $Q(n^{th}$ *control loop iteration fails*$)$, which is monotonic in the error rates, is instantiated with the aforementioned upper bounds on the error rates, it satisfies the IID property with respect to n.

FIT analysis. We use the probability of a failed control loop iteration, *i.e.*, the result of Lemma 6, to derive the NCS's FIT rate. First, we derive a lower bound on the mean time to failure (MTTF) of the control loop. Recall from §1 that a control failure occurs if the control loop violates its (m, k) specification. We model this problem in the form of a well-studied *a-within-consecutive-b-out-of-c:F* system model [32], and leverage existing results [54] (which depend on the IID property of the iteration failure probability) on the reliability analysis of this system model to safely lower-bound the MTTF. Given an MTTF lower bound $MTTF_{LB}$ in hours, the FIT rate is computed as $10^9/MTTF_{LB}$ [57]. The full derivation and evaluation of the FIT analysis is available online [24].

5 Evaluation

The objective of the evaluation is threefold. First, in order to understand the accuracy of our approach, we compare the proposed analysis with simulations (§5.1). Second, we demonstrate the ability of our analysis to reveal and quantify non-obvious differences in the reliability of workloads with different (m, k) requirements and subject to error rates (§5.2). And third, we illustrate the utility of our analysis in a design-space exploration context by comparing FITs of different replication schemes (§5.3).

To implement the analysis, we extended the SchedCAT [10] library to support our system model for CAN-based NCSs, and implemented the proposed analysis on top. All computations related to the analysis were carried out at a precision of 200 decimal places using the *mpmath* Python library for arbitrary precision arithmetic [31]. As the underlying timing analysis of the network, we used Broster *et al.*'s probabilistic response-time analysis for CAN [12]. We also implemented a simulation of a CAN-based NCS that mimics the system model described in §2 along with CAN's network transmission protocol (see [44] for a detailed description).

We use an active suspension workload for our experiments since it plays an important role in ensuring the stability of a vehicle, and since robustness of such control systems under faults has been thoroughly investigated in the past. For example, Li in his thesis [36] discusses related work in the context of actuator delays and faults, and proposes a fault-tolerant controller design for guaranteeing asymptotic stability. We base our experiments on the CAN-based active suspension workload studied by Anta and Tabuada [4], since it nicely matches our SISO NCS model. However, while Anta and Tabuada assume hard constraints and vary the control loop periodicity for improved bandwidth allocation, our objective is to explore the reliability of the control loop when assigned different (m, k)-firm configurations (synthetically chosen in this paper) and for different fault parameters.

The workload consists of four control loops (L_1, L_2, L_3, and L_4) corresponding to the control of four wheels (W_1, W_2, W_3, and W_4) with magnetic suspensions (period $1.75\,ms$), two hard real-time messages that report the current in the power line cable (period $4\,ms$) and the internal temperature of the coils (period $10\,ms$). In addition, we assumed the presence of a clock synchronization message with a period of $50\,ms$ [21] and a soft real-time message responsible for logging with a period of $100\,ms$. The logging messages carried payloads of eight bytes each, the control loop messages carried payloads of three bytes each, and the remaining messages carried one-byte payloads. Considering a bus rate of $1\,mbit/s$, the workload resulted in a total bus utilization of 40%. The clock synchronization message stream had the highest priority, followed by the current and temperature monitoring message streams, the control message streams, and last, the logging message stream.

The recovery time from a crash was set to $R_h = 1\,s$ for each host $H_h \in H$, and the exposure interval of each message stream was set to ten times its period to reflect the possibility of latent errors. The error rates and the (m, k) specifications used in each experiment are mentioned in the corresponding graphs. All error rates in the following are reported as the mean number of errors per *ms*. For context, Ferreira *et al.* [20] and Rufino *et al.* [51] reported peak transmission error rates range from 10^{-4} in aggressive environments to 10^{-10} in lab conditions, and as per Hazucha and Svensson [28], a 4 Mbit SRAM chip has a fault rate of approximately 10^{-12}. The error rates used in the following experiments have similar orders of magnitudes. Since the actuator task is not replicated, its host was assumed to be heavily shielded and thus assigned negligibly low crash and incorrect computation error rates.

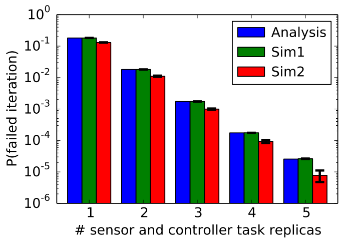
5.1 Experiment 1: Simulation vs. Analysis

To assess the accuracy of the proposed analysis, we compared the analytically-derived iteration-failure probability bound (§4) with an estimate of the mean iteration failure probability obtained through simulation. The pessimism incurred by the FIT analysis was already evaluated in prior work [24] and found to be acceptably small, and is not considered here.

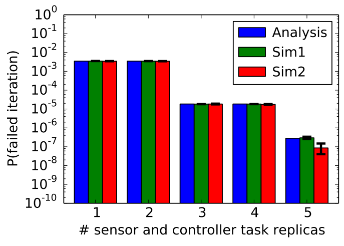
Recall from §4 that: **(1)** the analysis first upper-bounds the control loop iteration failure probability as a monotonic function of the exact message error probabilities; and **(2)** since it is impossible to determine the exact message error probabilities, a safe upper bound on the iteration failure probability is then obtained by instantiating the monotonic function from (1) with upper bounds on the exact message error probabilities (derived using the Poisson fault model in §3). To separately evaluate the pessimism incurred in steps (1) and (2), we used two different simulator versions `Sim-v1` and `Sim-v2` in this experiment.

In the simple version (`Sim-v1`), for each sensor message (and similarly for each control message), the message error probabilities were known to the simulator. Thus, each time any message is activated, the simulator draws a number uniformly at random from the range $[0, 1]$, compares it with the respective message error probabilities to decide whether the message is affected by that error type, and if the message is affected, simulates the corresponding error scenario. Thus, `Sim-v1` does not actually simulate Poisson processes, nor does it simulate the CAN protocol, but it helps to isolate the pessimism incurred in step (1).

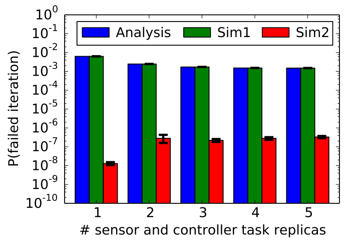
`Sim-v2` is more complex than `Sim-v1`, and simulates the entire NCS along with the CAN transmission protocol. Separate Poisson processes are used to generate the respective fault events on each host and on the network. These fault events may manifest as message errors if they coincide with the message's lifetime, *e.g.*, as an incorrect computation error if they coincide with the message's exposure interval and a retransmission error if they coincide with the message's network transmission interval. `Sim-v2` evaluates the pessimism incurred when upper-bounding the message error probabilities as a function of the raw transient fault rates using the Poisson model, *e.g.*, when using the Poisson-based CAN timing analysis [13]



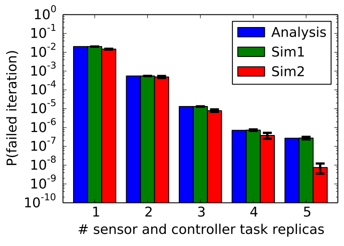
(a) $\tau = 3 \times 10^{-20}$, $\rho_i = 10^{-4}$, $\kappa_i = 10^{-20}$



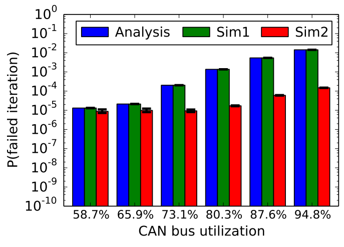
(b) $\tau = 3 \times 10^{-20}$, $\rho_i = 10^{-20}$, $\kappa_i = 10^{-4}$



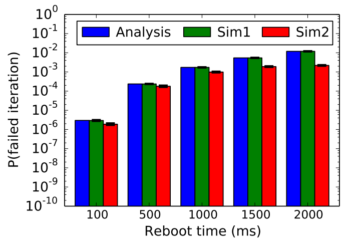
(c) $\tau = 3$, $\rho_i = 10^{-20}$, $\kappa_i = 10^{-20}$



(d) $\tau = 3 \times 10^{-1}$, $\rho_i = 10^{-5}$, $\kappa_i = 10^{-5}$



(e) $\tau = 3 \times 10^{-1}$, $\rho_i = 10^{-5}$, $\kappa_i = 10^{-5}$



(f) $\tau = 3 \times 10^{-20}$, $\rho_i = 10^{-4}$, $\kappa_i = 10^{-20}$

Figure 3 Comparing the iteration failure probability bound derived from the analysis with the estimates derived from simulation versions `Sim-v1` and `Sim-v2`. The vertical errors bars along with the simulation estimates denote 99% confidence intervals. Insets **(a)**, **(b)**, **(c)**, and **(d)** illustrate the variation in the iteration failure probability when the number of sensor and controller task replicas of L_1 are increased from one to five, for different sets of error rates. Insets **(e)** and **(f)** illustrate the impact of increasing CAN bus utilization and reboot times, respectively, for three replicas.

to determine bounds on deadline violation probabilities. It also evaluates whether this pessimism significantly impacts the overall iteration failure probability bound.

Both `Sim-v1` and `Sim-v2` make the worst-case assumption that any two faulty message copies are identical, as in the analysis.

We compared the analysis, `Sim-v1`, and `Sim-v2` for four different sets of error rates and replication factors. We used higher error rates for this experiment than can be realistically expected (and much higher than those used in the later experiments) as otherwise the simulations would be extremely time-consuming. To understand the effects of individual error types, we first compared three scenarios in which respectively only one of the three error types was assigned a significant rate, *i.e.*, $\rho_i = 10^{-4}$, $\kappa_i = 10^{-4}$, and $\tau = 3$, respectively, whereas the others were assigned negligible values, *i.e.* $\approx 10^{-20}$. Additionally, we evaluated a fourth scenario where all three error types have significant rates, *i.e.*, $\rho_i = 10^{-5}$, $\kappa_i = 10^{-5}$, and $\tau = 3 \times 10^{-1}$. Finally, to understand the effects of bus utilization and reboot time on the analysis, we compared the analysis, `Sim-v1`, and `Sim-v2` for different CAN bus utilizations (by assuming increased message payload sizes) and for different reboot times ($100\,ms$-$2000\,ms$), with a replication factor of three. The results are shown in Figs. 3a–3f.

Several trends can be clearly seen. First, in *all* evaluated scenarios, the analysis results always track `Sim-v1` extremely closely, which indicates that any pessimism introduced in step (2) to ensure monotonicity of the model with respect to the error rates is negligible. The results shown in Figs. 3a, 3b, and 3d further show that the analysis tracks `Sim-v2` quite closely, too, provided that the underlying CAN timing analysis is not the bottleneck (*i.e.*, if message delays are not the dominant source of failures). Specifically, we observe that the full analysis, including step (1), results in less than an order of magnitude difference between the predicted and observed failure probabilities if crash or incorrect computation errors are non-negligible. This confirms the overall accuracy of the approach for the intended use cases: the proposed analysis closely tracks and soundly bounds the actual iteration failure probabilities in the presence of crashes, retransmissions, and message corruptions.

However, as is evident from Figs. 3a, 3b, and 3d, there exist cases where the analysis diverges significantly from `Sim-v2`. The common factor in these scenarios is that the underlying CAN analysis is the dominating factor. Most prominently, this is visible in Fig. 3c, which focuses exclusively on transmission faults: while the analytical failure bound is initially large and then decreases gradually with increasing replication factor, the observed failure probability is several orders of magnitude smaller than the analytical bound and actually indicates the opposite trend – the analysis is not at all a good predictor of actual failure rates in this scenario. Fig. 3e indicates that the gap between `Sim-v2` and the analysis increases with CAN bus utilization. And even in Fig. 3a, when the replication factor is increased to five (resulting in high network contention), `Sim-v2` begins to deviate from the analysis.

We attribute the pessimism caused by the timing analysis to the fact that not every message instance experiences worst-case interference during transmission (*i.e.*, not every message is released at a *critical instant*), and consequently, the derived deadline violation probability is extremely pessimistic for most message instances.

We conclude that the pessimism incurred by the current CAN timing analysis is significant, however, this has a measurable effect only in cases where the network becomes the dominant reliability bottleneck. As we will show with the next experiment, this is rather unlikely in the case of realistic fault rates (in contrast to the extremely high rates assumed in this experiment for the sake of simulation speed, which exaggerate the impact of the CAN analysis).

Finally, Fig. 3f indicates that the pessimism incurred by step (1) also increases with the reboot time, which is also an exaggerated trend due to the extremely high rate of crash failures in this scenario (*i.e.*, $\rho_i = 10^{-4}$ per millisecond, which means a reboot is expected every 10 seconds on average). As a result, with increasing reboot times, it becomes more likely that a crash fault affects an already-crashed host while it is rebooting – which "masks" in part the effects of the prior crash, which our analysis does not exploit. For more realistic crash rates, the effect is negligible, and even in this exaggerated setup, the analysis stays within an order of magnitude of the observed failure rate (note the y-axis scale in Fig. 3f).

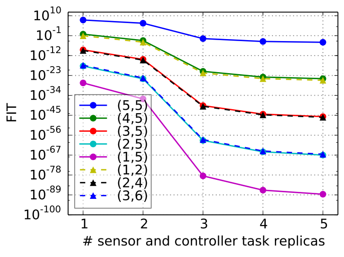
In summary, we conclude from the overall small gap to the `Sim-v2` baseline that the incurred pessimism is not significant in cases where the crash and incorrect computation error rates are non-negligible, and where network congestion is not the sole dominating bottleneck.

5.2 Experiment 2: FIT for Different Parameters

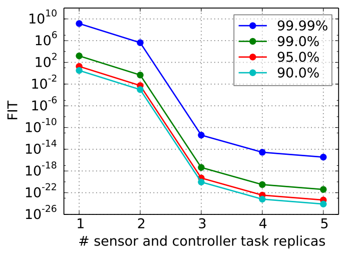
To evaluate the impact of different replication factors, (m, k) requirements, and environmental conditions, we next evaluated control loop L_1's FIT while varying the number of sensor and controller task replicas. Figs. 4a–4d present the results.

In Fig. 4a, m and k were varied as follows: $1 \leq m \leq 5$, and $k = 5$ or $k = 2m$; and in Fig. 4b, m/k is 90%, 95%, 99%, or 100% (while minimizing m and k).

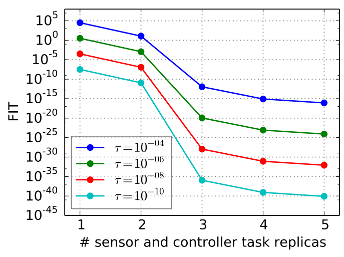
A hard specification, *i.e.*, where $m = k$, yields a much higher FIT rate compared to all other specifications with $m < k$, even the ones with $m/k \geq 0.9$, which highlights the need for a robustness-aware reliability analysis. For example, in Fig. 4b, if the desired reliability threshold is 10 FIT, a hard real-time analysis (*i.e.*, requiring a 100% iteration success rate) requires the use of three replicas, whereas if a 90% success rate is sufficient, then our analysis indicates that no replication is required. Fig. 4a shows that increasing both m and k while



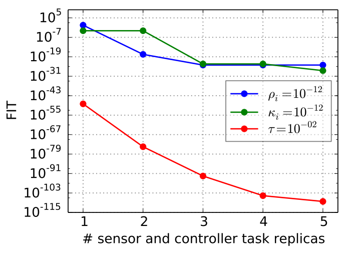
(a) $\tau = 10^{-4}$, $\rho_i = 10^{-12}$, $\kappa_i = 10^{-12}$



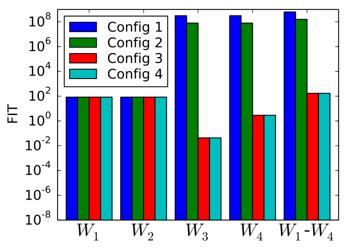
(b) $\tau = 10^{-4}$, $\rho_i = 10^{-12}$, $\kappa_i = 10^{-12}$



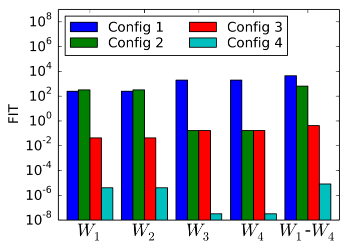
(c) $m = 9$, $k = 10$, $\rho_i = \kappa_i = \tau \times 10^{-6}$



(d) $m = 9$ and $k = 10$



(e) $\tau = 10^{-4}$, $\rho_i = 10^{-8}$, $\kappa_i = 10^{-12}$



(f) $\tau = 10^{-4}$, $\rho_i = 10^{-8}$, $\kappa_i = 10^{-12}$

Figure 4 (a, b) Parameters m and k are varied. **(c, d)** Failure rates τ, κ_i, and ρ_i are varied. **(e, f)** Replication factors of the different control loops are varied.

keeping m/k constant reduces the FIT rate, which shows that an asymptotic specification that relies only on the ratio m/k (and where k can hence be chosen to be arbitrarily large) can be easily supported by our analysis. Interestingly, different (m, k) specifications can result in very similar FIT rates, *e.g.*, the curves of $(3, 5)$ and $(2, 4)$ in Fig. 4a overlap.

Next, we varied the transmission error rate τ across 10^{-4}, 10^{-6}, 10^{-8}, and 10^{-10}. The crash and incorrect computation error rates were set to $\rho_i = \kappa_i = \tau \times 10^{-6}$. The results are illustrated in Fig. 4c. As expected, the FIT rates decrease as the error rates are lowered. The FIT rates also decrease with increasing replication, but this decrease is significant only up to three replicas. Confirming earlier observations [23], active replication in real-time systems results in diminishing returns or becomes counterproductive after some point, as it reduces the slack available for fault-induced retransmissions and results in increased FIT rates. In general, graphs such as these can help engineers identify the maximum reliability that they can extract out of a system by increasing its replication factor, or conversely, the minimum number of replicas needed to achieve a desired level of reliability.

To better understand the effects of individual error types, we computed FIT values for three different scenarios. In each scenario, only one of the three error types was assigned a significant rate, *i.e.*, $\rho_i = 10^{-12}$, $\kappa_i = 10^{-12}$, and $\tau = 10^{-2}$, respectively, whereas the others were assigned negligible values, *i.e.*, 10^{-48}. As apparent in Fig. 4d, the FIT rates are higher for crash and incorrect computation errors, but very low for transmission errors, despite a relatively high retransmission rate and even at high utilization (five replicas). This indicates the relative importance of tolerating host errors, at least when hard timeliness is not required, and also puts in perspective the pessimism observed in §5.1 – while the CAN analysis is the single biggest source of pessimism, its overall contribution to the overall failure probability is relatively minor for realistic retransmission rates.

Fig. 4d also shows that active replication helps tolerate incorrect computation errors only if the number of replicas is odd (*i.e.*, the curve for $\kappa_i = 10^{-12}$ does not improve when going from 1 to 2, or 3 to 4, replicas), in contrast to crash errors, which become less relevant already with the first added replica (until a point of diminishing returns is reached at 3 replicas). This is expected, *e.g.*, with two replicas, the voting algorithm is unable to distinguish between a correct and an incorrect input, which is significant if the messages are frequently corrupted.

In summary, Experiment 2 demonstrates that the analysis allows engineers to quantify and explore an NCS's reliability under various environmental conditions (*i.e.*, for varying peak error rates) and for different levels of robustness (by varying (m, k) specifications).

5.3 Experiment 3: FIT for Different Replication Schemes

To demonstrate that the analysis is useful for identifying reliability bottlenecks with respect to resource constraints, and for identifying opportunities to significantly increase a system's reliability at modest costs, we conducted a case study in which we analyzed different replication schemes of the workload. Our objective was to identify a replication scheme with a FIT rate under 10. That is, if such an active suspension workload is deployed in, say, 100 million cars, then as per Mancuso's calculations (discussed in §1), no more than about one vehicle per day will experience a failure in its active suspension NCS.

We considered the following error rates: $\tau = 10^{-4}$, $\rho_i = 10^{-8}$, and $\kappa_i = 10^{-12}$. To model practical design constraints, we assumed that the rear wheels W_1 and W_2 were close to many electromechanical parts, and assigned the hosts of the respective sensor tasks an order of magnitude higher crash and incorrect computation error rates. The different configurations are summarized in Table 1 and the their FIT bounds are illustrated in Figs. 4e and 4f.

Given a period of $1.75\,ms$ and an (m, k)-firm specification of (9, 10) for each control loop, the bound on the total FIT rate without any replication is greater than 10^{10}. Can we find a replication scheme with a FIT rate under 10 and with as few replicas as possible?

To answer the question, we conducted an exhaustive search over all possible replication schemes, varying the replication factor of each task from one to five, ignoring any scheme that did not result in a schedulable system. While we do not report the results of this exhaustive search due to space constraints, we observed that all feasible replication schemes can be partitioned into a few groups, where each group corresponds to schemes that result in FIT rate bounds of roughly the same order of magnitude. Thus, for each group, we report only the scheme with the minimum number of replicas, as given by Configurations 1–4 in Table 1 and Fig. 4e (configurations 5-8 and the corresponding Fig. 4f are discussed below).

Unfortunately, none of the feasible replication schemes yields a FIT rate under 10. Configuration 1 contains two copies of the sensor and controller tasks for L_1 and L_2, which helps reduce their respective FIT rate to under 10^2, but the system's total FIT rate still remains high ($\approx 10^8$) owing to L_3 and L_4's high individual FIT rates. Adding an extra replica of the sensor task for L_3 and L_4 (Configuration 2) does not help reduce this difference, but adding an extra copy of both sensor and controller tasks for L_3 and L_4 (Configuration 3) reduces the total FIT to around 10^2. In fact, while L_3 and L_4 are the bottleneck in Configuration 1 and Configuration 2, the bottleneck in Configuration 3 is L_1 and L_2. At this point, it seems that adding another pair of replicas for the rear wheel sensors (Configuration 4) to tolerate the relatively higher fault rates might be sufficient to bring down the total FIT rate under 10. However, this does not yield any significant benefit, and since we have maxed out the bus utilization, we cannot add any more replicas. This shows that with the current set of parameters, we cannot guarantee a FIT of under 10, which would have been difficult to realize without the proposed analysis.

Can we instead relax the parameters of the control loops at the cost of slightly affecting their *instantaneous quality-of-control* [4]? For example, does **(i)** a shorter period of $1.25\,ms$

Table 1 Different replication schemes. Parameters xS and yC denote that x and y replicas were provisioned for the sensor and the controller task of the respective wheel control loops.

Config.	Wheel 1	Wheel 2	Wheel 3	Wheel 4	Period	(m, k)	Util.
1	$2S$, $2C$	$2S$, $2C$	$1S$, $1C$	$1S$, $1C$	1.75 ms	(9, 10)	59%
2	$2S$, $2C$	$2S$, $2C$	$2S$, $1C$	$2S$, $1C$	1.75 ms	(9, 10)	68%
3	$2S$, $2C$	$2S$, $2C$	$2S$, $2C$	$2S$, $2C$	1.75 ms	(9, 10)	77%
4	$4S$, $2C$	$4S$, $2C$	$2S$, $2C$	$2S$, $2C$	1.75 ms	(9, 10)	96%
5	$2S$, $1C$	$2S$, $1C$	$1S$, $1C$	$1S$, $1C$	1.25 ms	(3, 5)	68%
6	$2S$, $2C$	$2S$, $2C$	$2S$, $2C$	$2S$, $2C$	2.50 ms	(19, 20)	55%
7	$3S$, $2C$	$3S$, $2C$	$2S$, $2C$	$2S$, $2C$	2.50 ms	(19, 20)	61%
8	$3S$, $3C$	$3S$, $3C$	$3S$, $3C$	$3S$, $3C$	2.50 ms	(19, 20)	81%

with a relaxed (m, k)-firm specification of (3, 5), or alternatively, **(ii)** a relaxed period of 2.5 ms with a stricter (m, k)-firm specification of $(19, 20)$ allow designing the system with the desired levels of reliability, *i.e.*, with a FIT rate of 10 or less? To answer this question, we once again exhaustively generated FIT bounds for all schedulable replication schemes and report four representative cases here (see Configurations 5–8 in Table 1 and Fig. 4f).

For case (i), the best possible FIT bound ($\approx 10^3$) is obtained when two copies of the L_1 and L_2 sensor tasks are provisioned (Configuration 5). While we could add a few more replicas to Configuration 5 without saturating the bus, this does not help to reduce the FIT bound any further. Case (ii), however, allows us to add many more replicas (Configurations 6–8) because of the relaxed period, yielding much better FIT bounds despite the stricter (m, k)-firm specification. In particular, Configuration 7 yields a total FIT bound under 1 and Configuration 8 yields a total FIT bound of around 10^{-5}. Thus, while case (i) is not a useful alternative, case (ii) shows clear reliability benefits. In fact, the substantial FIT reduction in case (ii) makes it a worthwhile tradeoff, despite the slightly degraded control quality [4], whereas case (i) would give up control quality for no appreciable gain in reliability.

In general, this case study highlights the importance of quantifying system reliability for design-space exploration and for identifying and strengthening the weakest link of a system (*e.g.*, in this study, L_3 and L_4 in Configurations 1–2, and L_1 and L_2 in Configuration 3), and that the proposed analysis is an effective aid in this process.

6 Related Work

The (m, k)-firm model was first studied in the context of real-time streams and control applications [27, 50]. Since then, many analyses and system designs have been proposed for applications with (m, k)-firm specifications, mainly focussing on their temporal aspects (*e.g.*, see [11]). We use (m, k)-firm specifications to model control system robustness, where the specification is a function of control loop iteration failures in both the time and value domains.

With regard to real-time networks, several reliability analyses have been proposed to date, particularly of the CAN bus under EMI-induced retransmission errors, *e.g.*, [59, 49, 45, 12, 16]. For example, Punnekkat *et al.* [49] and Broster *et al.* [12] proposed analyses to bound the response time of CAN messages assuming a sporadic and a Poisson model of EMI, respectively, and recently, Sebastian *et al.* [53] proposed the use of hidden Markov models in this context. Our prior work [23] proposed an analysis to bound the probability of successful transmission of a single logical message stream over CAN assuming host failures. In this work, like some of the prior work, we use the Poisson model of EMI, but unlike all aforementioned analyses, we evaluate the reliability of an end-to-end NCS system model.

Related work in the NCS domain has focussed on evaluating the criteria for control stability and performance, *i.e.*, to what extent a control system can deviate from ideal operating conditions without jeopardizing its functionality in the wake of various network failures, *e.g.*, [14, 38, 46, 30]. In contrast, we abstract the control specifics and solve the related, but orthogonal problem of determining *when* and *how frequently* such robustness criteria are not met, *i.e.*, how likely it is that an inherently robust control system deviates from ideal operating conditions to such a degree that its controlled plant may enter an unrecoverable state.

In related work targeting overall system reliability, Dugan and Van Buren [18] evaluated the reliability of a specific system, namely a fly-by-wire systems with passive replication (hot standbys), using Markov models to evaluate the state transition probabilities when a system component fails, and fault trees to evaluate the reliability of each of the individual states. It is possible to extend our analysis for systems with passive replication in a similar manner. Sinha [55] proposed a reliability analysis of a fail-operational brake-by-wire system networked with CAN and FlexRay buses. Sinha's approach differs substantially from ours since it is not defined at the granularity of individual messages. In contrast to these works that focus on specific systems, our analysis targets broadcast-based NCSs in general.

An alternative approach for quantifying the reliability of NCSs faced with transient component failures is the use of probabilistic model checkers such as PRISM [34] and Storm [17]. This approach has been adopted in a number of works for reliability analysis of simple networked systems [33, 19, 58, 2, 40, 35]. While such model-checking approaches are very general, their weak spot is generally the question of scalability. In contrast, our analysis is specific to the presented NCS model, but in return does not suffer from state-space explosion issues. We plan to carry out a comparison of the two approaches in future work.

7 Conclusion

We have proposed the first analysis to safely bound the FIT rate of CAN-based SISO NCSs that employ active replication to mitigate transient errors. Our analysis accounts for failures in both the time and value domains, and exposes the inherent robustness of NCSs in the form of (m, k)-firm constraints.

There is plenty of scope for future work, especially on more complex system models. To tolerate failures in the actuator task, it could be replicated like the sensor and controller tasks. Assuming that the physical actuator has some mechanism for redundancy suppression (*e.g.*, a hardware voter), such a system can be analyzed similarly to the presented analysis.

A fault-tolerant multi-input single-output (FT-MISO) control loop can be analyzed by modifying Steps 1 and 2 in §4 to account for all replicas of the distinct sensor tasks in the system. In contrast, a fault-tolerant multi-input multi-output system can be analyzed as multiple independent FT-MISO systems, if an (m, k)-firm specification is given for each actuator.

For adaptive systems that allow dynamic reconfiguration of task replication factors based on runtime monitoring of the error rates, our analysis can be used to evaluate the reliability of different system modes. Similarly, in systems using passive replication or subject to permanent failures, our analysis yields a FIT rate for each state in the system's lifetime, *i.e.*, given a set of alive/dead replicas for that state, as in [18].

References

1 IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, July 2008. `doi:10.1109/IEEESTD.2008.4579760`.

2 Masakazu Adachi, Yiannis Papadopoulos, Septavera Sharvia, David Parker, and Tetsuya Tohdo. An approach to optimization of fault tolerant architectures using hip-hops. *Software: Practice and Experience*, 41(11):1303–1327, 2011.

3 Zaid Al-Ars and Ad J van de Goor. Transient faults in DRAMs: Concept, analysis and impact on tests. In *International Workshop on Memory Technology, Design and Testing*, pages 59–64. IEEE, 2001.

4 Adolfo Anta and Paulo Tabuada. On the benefits of relaxing the periodicity assumption for networked control systems over CAN. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 3–12. IEEE, 2009.

5 Robert B Ash. *Basic Probability Theory*. Courier Corporation, 2012.

6 Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

7 Ali Bakhoda, Seyed Ghassem Miremadi, and Hamid R Zarandi. Investigation of transient effects on fpga-based embedded systems. In *Proceedings of the 2nd International Conference on Embedded Software and Systems*, pages 6–pp. IEEE, 2005.

8 Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, 1993.

9 Rainer Blind and Frank Allgöwer. Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *Proceedings of the 54th Annual Conference on Decision and Control*, pages 7510–7515. IEEE, 2015.

10 Björn B Brandenburg. The schedulability test collection and toolkit, 2017. Available at `https://people.mpi-sws.org/~bbb/projects/schedcat`.

11 Ian Broster, Guillem Bernat, and Alan Burns. Weakly hard real-time constraints on controller area network. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 134–141. IEEE, 2002.

12 Ian Broster, Alan Burns, and Guillermo Rodriguez-Navas. Probabilistic analysis of CAN with faults. In *Proceedings of the 23rd Real-Time Systems Symposium*, pages 269–278. IEEE, 2002.

13 Ian Broster, Alan Burns, and Guillermo Rodriguez-Navas. Timing analysis of real-time communication under electromagnetic interference. *Real-Time Systems*, 30(1-2):55–81, 2005.

14 Ahmet Cetinkaya, Hideaki Ishii, and Tomohisa Hayakawa. Networked control under random and malicious packet losses. *Transactions on Automatic Control*, 62(5):2434–2449, 2017.

15 Cristian Ionut Chihaia. *Active Fault-Tolerance in Wireless Networked Control Systems*. PhD thesis, Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften» Elektrotechnik und Informationstechnik» Automatisierungstechnik und komplexe Systeme, 2010.

16 Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

17 Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 592–600, 2017. `doi:10.1007/978-3-319-63390-9_31`.

18 Joanne Bechta Dugan and Randy Van Buren. Reliability evaluation of fly-by-wire computer systems. *Journal of Systems and software*, 25(1):109–120, 1994.

19 Jonas Elmqvist and Simin Nadjm-Tehrani. Formal support for quantitative analysis of residual risks in safety-critical systems. In *Proceedings of the 11th High Assurance Systems Engineering Symposium*, pages 154–164. IEEE, 2008.

20 Joaquim Ferreira, Arnaldo Oliveira, Pedro Fonseca, and José Fonseca. An experiment to assess bit error rate in CAN. In *Proceedings of the 3rd International Workshop of Real-Time Networks*, pages 15–18, 2004.

21 Martin Gergeleit and Hermann Streich. Implementing a distributed high-resolution real-time clock using the CAN-bus. In *Proceedings of the 1st International CAN Conference*, volume 94, 1994.

22 Alain Girault, Hamoudi Kalla, and Yves Sorel. An active replication scheme that tolerates failures in distributed embedded real-time systems. In *Design Methods and Applications for Distributed Embedded Systems*, pages 83–92. Springer, 2004.

23 Arpan Gujarati and Björn B Brandenburg. When is CAN the weakest link? A bound on failures-in-time in CAN-based real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 249–260. IEEE, 2015.

24 Arpan Gujarati, Mitra Nasri, and Björn B Brandenburg. Lower-bounding the MTTF for systems with (m,k) constraints and IID iteration failure probabilities. Technical Report MPI-SWS-2018-004, Max Planck Institute for Software Systems, Germany, 2018. URL: `http://www.mpi-sws.org/tr/2018-004.pdf`.

25 Arpan Gujarati, Mitra Nasri, and Björn B Brandenburg. Quantifying the resiliency of fail-operational real-time networked control systems. Technical Report MPI-SWS-2018-005, Max Planck Institute for Software Systems, Germany, 2018. URL: `http://www.mpi-sws.org/tr/2018-005.pdf`.

26 Rachana A Gupta and Mo-Yuen Chow. Overview of networked control systems. In *Networked Control Systems*, pages 1–23. Springer, 2008.

27 Moncef Hamdaoui and Parameswaran Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995.

28 Peter Hazucha and Christer Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science*, 47(6):2586–2594, 2000.

29 Rolf Isermann, Ralf Schwarz, and Stefan Stolzl. Fault-tolerant drive-by-wire systems. *IEEE Control Systems*, 22(5):64–81, 2002.

30 Ning Jia, Ye-Qiong Song, and Rui-Zhong Lin. Analysis of networked control system with packet drops governed by (m, k)-firm constraint. In *Fieldbus Systems and Their Applications 2005*, pages 63–70. Elsevier, 2006.

31 Fredrik Johansson. mpmath - Python library for arbitrary-precision floating-point arithmetic, 2017. Available at `http://mpmath.org/`.

32 Way Kuo and Ming J Zuo. *Optimal Reliability Modeling: Principles and Applications.* John Wiley & Sons, 2003.

33 Marta Kwiatkowska, Gethin Norman, and David Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427–1434, 2007.

34 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

35 Florian Leitner-Fischer. *Causality Checking of Safety-Critical Software and Systems.* PhD thesis, University of Konstanz, Germany, 2015. URL: `http://kops.uni-konstanz.de/handle/123456789/30778`.

36 Hongyi Li. *Robust Control Design for Vehicle Active Suspension Systems with Uncertainty*. PhD thesis, University of Portsmouth, Portsmouth, 2012.

37 Xiaodong Li, Sarita V Adve, Pradip Bose, and Jude A Rivers. Architecture-level soft error analysis: Examining the limits of common assumptions. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 266–275. IEEE, 2007.

38 Feng-Li Lian, James Moyne, and Dawn Tilbury. Analysis and modeling of networked control systems: MIMO case with multiple time delays. In *Proceedings of the American Control Conference*, volume 6, pages 4306–4312. IEEE, 2001.

39 George MA Lima and Alan Burns. A consensus protocol for CAN-based systems. In *Proceedings of the 24th Real-Time Systems Symposium*, pages 420–429. IEEE, 2003.

40 Yu Lu. *Probabilistic Verification of Satellite Systems for Mission Critical Applications*. PhD thesis, University of Glasgow, 2016.

41 Renato Mancuso. *Next-Generation Safety-Critical Systems on Multi-Core COTS Platforms*. PhD thesis, University of Illinois at Urbana-Champaign, 2017. Available at `http://hdl.handle.net/2142/97399`.

42 Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 29–40. IEEE, 2003.

43 Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, and Ravishankar K Iyer. An architectural framework for detecting process hangs/crashes. In *Proceedings of the European Dependable Computing Conference*, pages 103–121. Springer, 2005.

44 Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Springer, 2012.

45 Nicolas Navet, Y-Q Song, and Françoise Simonot. Worst-case deadline failure probability in real-time applications distributed over Controller Area Network. *Journal of Systems Architecture*, 2000.

46 Johan Nilsson. *Real-Time Control Systems with Delays*. PhD thesis, Lund Institute of Technology Lund, Sweden, 1998.

47 John Noto, Gary Fenical, and Colin Tong. Automotive EMI shielding–controlling automotive electronic emissions and susceptibility with proper EMI suppression methods. URL: `https://www.lairdtech.com/sites/default/files/public/solutions/Laird-EMI-WP-Automotive-EMI-Shielding-040114.pdf`.

48 Stefan Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, volume 345. Springer Science & Business Media, 2007.

49 Sasikumar Punnekkat, Hans Hansson, and Christer Norstrom. Response time analysis under errors for CAN. In *Proceedings of the 6th Real-Time Technology and Applications Symposium*, pages 258–265. IEEE, 2000.

50 Parameswaran Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

51 Jose Rufino, Paulo Verissimo, Guilherme Arroz, Carlos Almeida, and Luis Rodrigues. Fault-tolerant broadcasts in CAN. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pages 150–159. IEEE, 1998.

52 Indranil Saha, Sanjoy Baruah, and Rupak Majumdar. Dynamic scheduling for networked control systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 98–107. ACM, 2015.

53 Maurice Sebastian, Philip Axer, and Rolf Ernst. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *Proceedings of the*

17th Pacific Rim International Symposium on Dependable Computing, pages 79–88. IEEE, 2011.

54 M. Sfakianakis, S. Kounias, and A. Hillaris. Reliability of a consecutive k-out-of-r-from-n:F system. *Transactions on Reliability*, 41(3):442–447, 1992.

55 Purnendu Sinha. Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives. *Reliability Engineering & System Safety*, 96(10):1349–1359, 2011.

56 Fedor Smirnov, Michael Glaß, Felix Reimann, and Jürgen Teich. Formal reliability analysis of switched ethernet automotive networks under transient transmission errors. In *Proceedings of the 53nd Design Automation Conference*, pages 1–6. IEEE, 2016.

57 Susan Stanley. MTBF, MTTR, MTTF & FIT explanation of terms. URL: `http://imcnetworks.com/wp-content/uploads/2014/12/MTBF-MTTR-MTTF-FIT.pdf`.

58 Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Augmenting formal development of control systems with quantitative reliability assessment. In *Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems*, pages 61–70. ACM, 2010.

59 Ken Tindell and Alan Burns. Guaranteeing message latencies on Control Area Network (CAN). In *Proceedings of the 1st International CAN Conference*, 1994.

60 Nicholas J Wang, Justin Quek, Todd M Rafacz, and Sanjay J Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–70. IEEE, 2004.

Camera Networks Dimensioning and Scheduling with Quasi Worst-Case Transmission Time

Viktor Edpalm
Axis Communications, Lund, Sweden
viktor.edpalm@axis.com

Alexandre Martins
Axis Communications and Department of Automatic Control, Lund University, Lund, Sweden
alexandre.martins@axis.com

Karl-Erik Årzén
Department of Automatic Control, Lund University, Sweden
karlerik@control.lth.se

Martina Maggio
Department of Automatic Control, Lund University, Sweden
martina@control.lth.se
https://orcid.org/0000-0002-1143-1127

Abstract

This paper describes a method to compute frame size estimates to be used in quasi Worst-Case Transmission Times (qWCTT) for cameras that transmit frames over IP-based communication networks. The precise determination of qWCTT allows us to model the network access scheduling problem as a multiframe problem and to re-use theoretical results for network scheduling. The paper presents a set of experiments, conducted in an industrial testbed, that validate the qWCTT estimation. We believe that a more precise estimation will lead to savings for network infrastructure and to better network utilization.

2012 ACM Subject Classification Computer systems organization → Embedded systems, Computer systems organization → Real-time systems, Networks → Network components

Keywords and phrases worst-case transmission time, H.264, bandwidth estimation, video compression, network access scheduling, multiframe model, camera network

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.17

Funding This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Alexandre Martins, Karl-Erik Årzén, and Martina Maggio are members of the LCCC Linnaeus Center and the ELLIIT Excellence Center at Lund University.

1 Introduction

In the modern interconnected world, multiple devices share access to networking resources, such as transmission bandwidth. For some of these devices — e.g., video surveillance cameras connected to a monitoring station [21] — access to networking resources is often more critical than access to computing resources [27–29]. Scheduling network access is therefore crucial for the satisfaction of real-time requirements [1, 18, 25, 26], like the timely transmission of surveillance videos from different cameras [22].

A typical video surveillance system comprises of multiple cameras disseminated over an area. These cameras continuously record a specific scene, it being an office space, a parking lot, a road, or any other alternative. The recorded scenes are of course different from one another, but their characteristics do not evolve significantly over time. A camera that is installed outdoor in a parking lot will record similar scenes, mostly involving cars and people, in different light conditions. At the same time, a camera that is pointing to a highway lane will (most likely) record either an empty road, or the passage of cars. A common challenge in the video surveillance industry is to tailor the entire infrastructure of the surveillance system to achieve a certain level of quality, while keeping the cost as limited as possible. Today, the video industry is mainly focused on using IP cameras, which stream videos that are compressed using the H.264 standard. In order to tailor the infrastructure, one must be able to anticipate how much data each camera in the system is expected to produce, given its unique set of internal characteristics and settings — e.g., position, placement, surrounding environment, etc. Such an estimate can be conservative, assuming that video frames are not compressed. Currently, conservative techniques are adopted for practical applications [1, 18, 22, 29]. However, conservativeness greatly increases infrastructure cost and limits the network usage. Non-conservative estimates have the potential of reducing the operational cost of video-surveillance systems. The challenge explored in this paper is therefore the estimation of the amount of data produced by each camera in the surveillance system.

We motivate our investigation by drawing a parallel between network scheduling for video surveillance camera systems and CPU scheduling. Using the periodic task model, a set $\mathcal{T} = \{\tau_1, \ldots, \tau_p\}$ of p tasks execute on a given hardware platform. Each $\tau_i = \{E_i, P_i, D_i\}$ is activated at precise time instants, determined by the period P_i, and must meet a given deadline D_i. For scheduling policies to be effective at ensuring the satisfaction of deadline constraints in complex systems, schedulers use information about the Worst-Case Execution Time (WCET) E_i of a task τ_i on the given hardware.

Similarly, a set $\mathcal{C} = \{c_1, \ldots, c_p\}$ of p surveillance cameras transmits video streams to a monitoring station. Each camera c_i has a given frame rate f_i, denoting the number of frames that the camera captures in a second. The frame rate has a direct implication on the transmission requirements of the camera, its inverse $1/f_i$ being equal to the activation period. For simplicity, we can assume that the deadline to transmit the currently captured frame is equal to the period. Hence, in this setting, reusing well-known CPU scheduling algorithms for network access depends on determining the Worst-Case Transmission Time (WCTT) for video frames. From the theoretical perspective, the task set model is not as simple as a set of periodic tasks, and can be described using a multiframe model [16], as will be shown in the following. Also, video encoders are very complex and the frame size depends heavily on the encoded scene. We therefore cannot compute precise upper bounds — e.g., using static analysis or formal methods — that guarantee that the given size is never exceeded. We therefore limit ourselves to the computation of quasi Worst-Case Transmission Times (qWCTT). We have experimentally verified that our estimate of the upper bound is valid in most cases and we have not encountered any case in which a frame exceeding our estimated upper bound is not a result of software bugs.

This paper contributes to the state of the art of real-time systems (and real-time surveillance video streaming) by:

- Determining a combination of measurable parameters that can accurately predict the expected H.264 frame sizes;
- Computing reasonable estimates of upper bounds for the qWCTT of frames of different

Table 1 Nomenclature: Acronyms.

Acronym	Brief Explanation
GOP	**Group of Pictures:** Set of one I-frame and multiple P- and B-frames. The number also represents the amount of frames between two consecutive I-frames
HDR	**High Dynamic Range:** Technique used to enhance video, that typically allows frames to include more details and be sharper
IDR	**Instantaneous Decoding Refresh:** I-frame that imposes a refresh, i.e., following frames must not need any information from frames prior to the IDR I-frame
QP	**Quantization Parameter:** Compression parameter defined in the H.264 standard, higher numbers indicate more information loss
SAO	**Size of Average Object:** Reflects the expected distance to an object in an image, determined by factors like the zoom level, field of view, and lens type, as well as placement of the camera
WCET	**Worst-Case Execution Time:** Upper bound on the time it takes for a task to execute on a given hardware platform
WCTT	**Worst-Case Transmission Time:** Indicates the maximum time it takes to transmit a frame of the video using the available network bandwidth
qWCTT	**quasi Worst-Case Transmission Time:** Indicates a non-exact upper bound for the transmission time of a frame using the available network bandwidth

types over a network, casting the problem of scheduling switched Ethernet network access into a multiframe non-preemptive scheduling problem;

- Conducting a thorough experimental campaign to validate our findings and the given models, providing parameters for different camera models and circumstances and allowing researchers to use the derived models to verify real-time properties on the network transmission time.

From the industrial perspective, the relevance of this paper is in enabling infrastructure tailoring for a video surveillance system and selecting quantities like the total required network bandwidth to guarantee a given video stream quality.

In the following, we review the H.264 standard and terminology in Section 2. Section 3 then discusses our models; enumerating the parameters, explaining how to measure them when needed, and showing the equations used to determine the frame sizes. Section 4 presents related efforts and Section 5 shows experimental results obtained with 6 different cameras in a laboratory environment and 24 different real-life surveillance scenarios. We finally conclude the paper in Section 6.

2 Background on Video Encoding

This section provides a brief overview of H.264, also called MPEG-4 part 10 AVC, which currently is the *de facto* standard for video encoding and decoding[1]. Table 1 presents a recap of the acronyms used in the paper.

[1] The first official version H.264 version was approved in March 2003 [17, 30] and has since evolved over time. The standard now includes more features and modes, the latest version being approved in April 2017 [11]. The MPEG LA organization administers most of the licenses for patents applying to this standard.

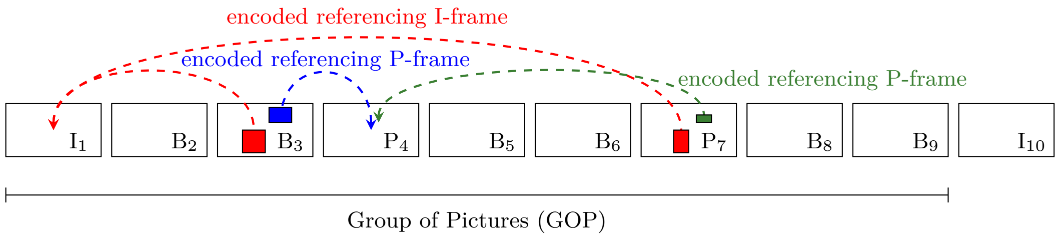

encoded referencing I-frame
encoded referencing P-frame
encoded referencing P-frame
I1
B2
B3
P4
B5
B6
P7
B8
B9
I10
Group of Pictures (GOP)

Figure 1 H.264 frame sequence: I-frames, P-frames, B-frames, and Group of Pictures.

H.264 is a video compression standard that defines how a video should be decoded. The implementation of the encoding is left to the manufacturer's discretion. The standard describes a *block based hybric codec*, i.e., a video is decomposed in blocks of data for encoding. To allow for video compression, H.264 uses motion-compensated encoding, i.e., it describes a frame by referencing parts of other frames, thus capturing the motion of objects across different frames [30]. A stream encoded with H.264 contains a sequence of frames, these frames are not necessarily encoded following the display order or time they were captured. Based on the frame encoding, it is possible to distinguish between three different types of frames: Intra frames (I-frames), Predicted frames (P-frames), and Bi-directional predicted frames (B-frames).

- I-frames are (usually[2]) self-contained. An I-frame contains the full image and does not need additional information in the decoding process. In terms of encoding, these are fast and easier to encode, as all the information should be present in the resulting frame and no extra buffer containing other frames are necessary. In terms of size, on the contrary, these are the most space-consuming type of frames.
- P-frames are encoded using information contained in the current frame and in previous ones (up to the last self-contained I-frame). In the encoding of a P-frame, part of the image can be encoded using references to previous ones with extra information to reproduce the difference, instead of repeating the information. This allows the encoder to compress the frame, reducing its size, at the cost of additional computation and buffering.
- B-frames are encoded using both information from previous frames and information from following frames. In a B-frame, the encoder can introduce references to frames that come next, in display order, with respect to the current one being encoded. B-frames require the most computational capacity for the encoding, but are usually the lightest in terms of space consumption.

Figure 1 shows a sequence of 10 frames. The first nine frames in the example denote a Group of Pictures (GOP). A GOP consists of an I-frame followed by a sequence of B-frames and P-frames. The I-frame can be marked as an Instantaneous Decoding Refresh (IDR), meaning that the following frames do not need information from frames prior to that one in the sequence. If all the I-frames are marked as IDR points, the decoding of each GOP is independent, otherwise it is not. The sequence of frame types is determined and fixed by a high-level controller before the frame encoding starts.

[2] If an I-frame is marked as an Instantaneous Decoding Refresh (IDR), its encoding is self-contained. In most cases, this is true, but there are certain conditions in which this does not hold. Since we are interested in estimating the upper bounds, we can safely assume that the upper bound of an I-frame is self-contained.

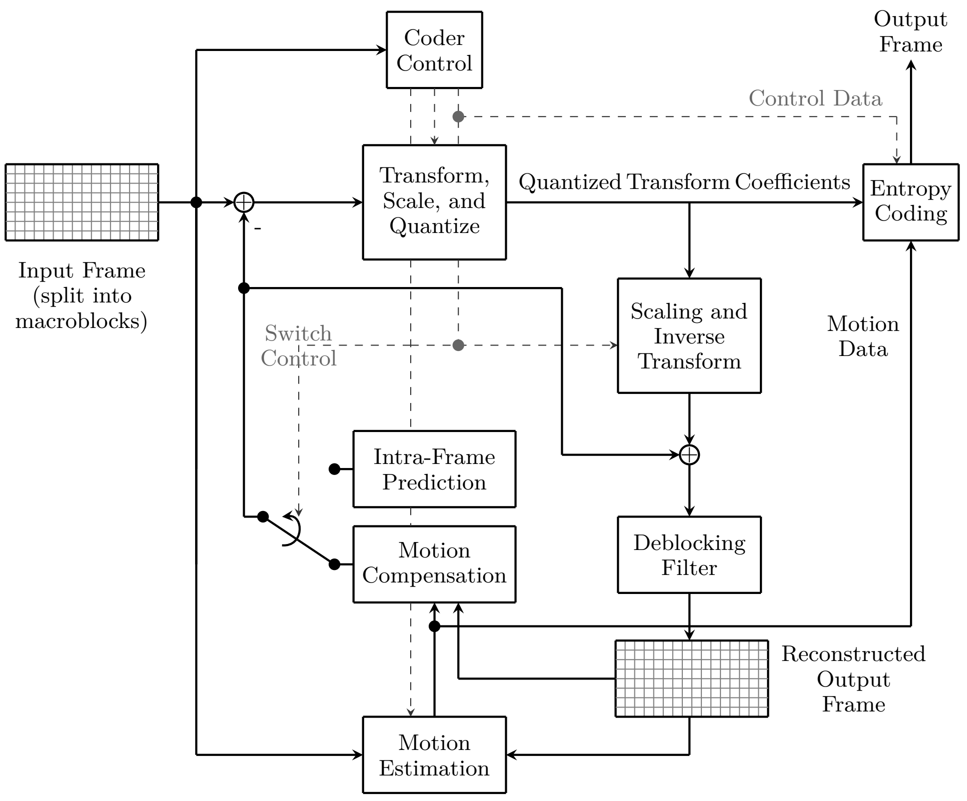

Coder Control
Output Frame
Control Data
Transform, Scale, and Quantize
Quantized Transform Coefficients
Entropy Coding
Input Frame (split into macroblocks)
-
Switch Control
Scaling and Inverse Transform
Motion Data
Intra-Frame Prediction
Motion Compensation
Deblocking Filter
Reconstructed Output Frame
Motion Estimation

Figure 2 Basic coding structure of a H.264 frame.

For the sequence shown in Figure 1, the first and the last frame are encoded as I-frames. The fourth and the seventh are encoded as P-frames. The remaining ones are encoded as B-frames. The red arrows in the Figure indicate areas of the third and seventh frames – respectively a B-frame and a P-frame – that are encoded as references to the previous I-frame. The blue arrow shows an area of the third B-frame that is encoded as reference to the following P-frame. The green arrow shows an area of the seventh P-frame that is encoded as a reference to the previous P-frame. These arrows are only examples and do not represent the full set of references of the encoding.

The given "areas" are composed of *macroblocks*. To be more precise, a generic H.264 frame is split into multiple 16×16 squares of pixels, each of them being a macroblock. Macroblocks are encoded/decoded separately from one another, and can be split into sub-blocks down to a block size of 4×4 pixels. Macroblocks are also assigned a type from the set {I, P, B}. I frames can contain only I-blocks. P-frames can contain both P-block and I-blocks. B-frames can contain all types of blocks.

Figure 2 shows an overview of the encoding process. The input frame is divided into macroblocks, each of them is passed to a Coder Control and to a Motion Estimation function. The Motion Estimation function uses some previously encoded and buffered frames, the number of them being determined by the Coder Control. These previous frames are used to choose if the current block should be encoded:

- as a new block, containing the full information (Intra-Frame Prediction, I-block),
- by referring to a previously encoded block in the same frame, containing a positional vector and the residual information (Intra-Frame Prediction, I-block),

- by referring to a block in a previous frame, containing a positional vector, the frame reference, and the residual information (Motion Compensation, P-block), or
- by referring to block in a previous or future frame, containing a positional vector, the frame reference, and the residual information (Motion Compensation, B-block).

The Motion Estimation function determines the cost for the four choices and selects the most appropriate one for the current macroblock.

The residual information is then Transformed, Scaled, and Quantized according to a Quantization Parameter (QP) to reduce its size. This is the only step where there is information loss and the higher the QP value, the higher the loss of information. The scaling, inverse transform and the deblocking filter allow the encoder to reconstruct the output frame and buffer it for future encoding. The entropy coding function uses lossless statistical compression to produce the final output frame.

3 Frame Size Estimation

The aim of this paper is to estimate an upper bound for the size of encoded video frames, to aid a potential external network manager towards a better scheduling of network capacity. The largest improvement is given when information-rich frames (I-frames) are treated separately from frames that can contain references to previous and future frames (P- and B-frames). The small difference in size of P- and B-frames and the similarity in the methods used for their construction justify the use of the same upper bound estimate for the two frame types. We therefore devise two models: an Intra Frame model for I-frames and an Inter Frame model for P- and B-frames. In Section 3.1 we explain what are the implication for network access scheduling. In Section 3.2 we describe the model we use for the estimation of the upper bound of the size of I-frames. In Section 3.3 we describe how to derive upper bounds estimates for P-frames and B-frames. In the following, we use $\propto$ to indicate proportionality.

3.1 Scheduling implications

Assume it is possible to compute an upper bound estimate for the size of I-frames, denoted with I^* and an upper bound estimate for the size of P- and B-frames, denoted with P^*. Knowing the network speed $\mathcal{N}$, e.g., 100 Mbps, one can then translate these bounds into knowledge of the WCTT for the two types of frames in the network. The GOP parameter specifies how many "dynamic" (P- and B-) frames there are in between two "static" (I-) frames.

In fact, when a set $\mathcal{C} = \{c_1, \ldots, c_p\}$ of p surveillance cameras share the same network, one can say that the i-th camera behaves according to the multiframe task model [16]. The camera has a vector of execution times $[E^0, E^1, \ldots E^{\text{GOP}-1}]$ and a single period and deadline, equal to the inverse of the frame rate $1/f_i$. E^0 is then equal to the upper bound estimate on the transmission time of the I-frame $I^*/\mathcal{N}$ and all the other execution times $[E^1, \ldots E^{\text{GOP}-1}]$ are equal to the upper bound estimates on the transmission time of the P-frame, i.e., $P^*/\mathcal{N}$. This allows us to reuse theoretical results developed for the specific model [5, 10, 15, 32] or for its generalizations [4, 7, 9, 14, 19, 24, 31]. In particular, once we have determined the WCTTs for the different frame types, we can use the analysis on non-preemptive scheduling of multiframe tasks [3, 6] to determine schedulability properties for a set of video-surveillance cameras communicating over switched Ethernet [2].

As video encoders are very complex software elements, we cannot really compute an upper bound with static analysis or formal methods, that would guarantee that the size will never exceed the one predicted. However, we can compute an approximation (estimate) of such upper bound, that is proven conservative in most cases. We believe that the very few circumstances in which the size of frames exceeds the computed values are due to problems and bugs of the execution of video-surveillance software. Therefore, we refer to $I^*/\mathcal{N}$ and $P^*/\mathcal{N}$ using the term quasi Worst-Case Transmission Times (qWCTT).

3.2 Intra Frame Model – I-frames

To determine the upper bound estimate I^* for the size of I-frames, we isolate the principal components that influence the amount of information included in the frame. Many acronyms and symbols are defined in the rest of the section. Table 2 contains a summary of the terms and constants that are needed for the estimation. The second column of the table contains a letter explaining how the value is obtained: `[C]` for computed, `[K]` for known, `[M]` for measured. Section 3.4 contains details on how to measure the `[M]`-parameters given a scene and a camera model.

Three different components influence the size of the frame: (i) the resolution of the video r, (ii) the compression level I_c, (iii) the actual camera and scene parameters I_a. There are many alternatives to write an expression of how each of these factors influences the size of the resulting frame. We decided to express I_c and I_a as scaling factors with respect to the resolution of the frame, therefore writing I^* as the product of the three terms,

$$I^* = \{r \cdot I_c \cdot I_a\}\,. \tag{1}$$

We now provide details for each of these terms separately.

- **Resolution r.** The frame resolution r is the number of pixels in the frame. Its value is equal to the product of the height h and the width w of the frame, $r = w \cdot h$. The resolution is linked to the number of macroblocks in the frame, therefore it influences its size directly.
- **Compression level I_c.** We denote with I_c the influence of the compression, $I^* \propto I_c$. The compression level QP determines the loss of information in each macroblock. From the H.264 standard, we infer that "an increase of 1 in QP corresponds to an increase of the quantization step size by approximately 12%" [30] (an increase of 6 means an increase of the quantization step size by a factor of 2).
  In order to properly capture this relationship, we define a reference QP, denoted with $\mathrm{QP}^{\mathrm{ref}}$, and express I_c as a function of the difference between the current value and the reference value, $\Delta\mathrm{QP} = \mathrm{QP} - \mathrm{QP}^{\mathrm{ref}}$. We select $\mathrm{QP}^{\mathrm{ref}} = 28$ as the baseline. This choice is arbitrary, but represents a commonly used value, and does not affect the generality of the approach. $\Delta\mathrm{QP}$ is used to scale the frame sizes between two compression levels, according to the relationship $I_c = 2^{-\frac{\Delta\mathrm{QP}}{6}}$. The expression in Equation (1) thus becomes

$$I^* = \{r \cdot I_c \cdot I_a\} = \left\{r \cdot 2^{-\frac{\Delta\mathrm{QP}}{6}} \cdot I_a\right\}. \tag{2}$$

- **Actual camera and scene parameters I_a.** The last component that influences the size of an I-frame includes a mix of camera and scene parameters, that we denote with I_a for "actual". I_a includes two different terms, $I_a = I_d + n_{c,\ell}$. The first one, I_d, is related to how many details the scene has and how well the camera is able to retain that information. The second one, $n_{c,\ell}$ is related to the amount of noise generated in the camera. I^* then becomes

$$I^* = \{r \cdot I_c \cdot I_a\} = \left\{r \cdot 2^{-\frac{\Delta\mathrm{QP}}{6}} \cdot I_a\right\} = \left\{r \cdot 2^{-\frac{\Delta\mathrm{QP}}{6}} \cdot (I_d + n_{c,\ell})\right\}. \tag{3}$$

The detail influence I_d, captures how the scene details and their perception at the camera level affect the size of the frame. These can be separated into two categories: (i) scene-dependent parameters (each camera reacts differently to them, but they are a property of the scene), (ii) camera-dependent parameters. Parameters in the first category should be measured, while parameters in the second category are either measured or known, e.g., available from the camera manufacturer.

In the first category, we include the scene illumination ℓ, the scene detail level d_s, and the nature parameter n. In the second category, we include the camera detail level d_c, the enhancing factor e induced by features like High Dynamic Range (HDR), and the Size of the Average Object (SAO) in the scene, which depends for example on the zoom level enforced by the camera. The resulting I_d is the product of all these factors. In fact, the factors are known or measured as the relative difference that they produce in the I-frame size.

The frame size is greatly influenced by the illumination of the surroundings ℓ, given that more light allows the camera to capture the scene better while the absence of light hides details in the image. The value of ℓ represents the ratio between the current illumination level and a reference one, it is is measured in a controlled environment with predetermined light levels. The result of the measurement is a value $\ell \in \mathbb{R}^+ \,|\, 0.25 \leq \ell \leq 1$. We consider three different light levels: low, medium, and high. A low light scenario is a scene recorded at night time, without any major light sources. A medium illumination scene is a night time scenario, with some light source illuminating the scene. A high illumination scene is a daylight scene, or a well lit indoor environment such as an office or a store. The high illumination scenario used as basis for scaling the remaining ones. This means that each camera at high light level has $\ell = 1$, and values for middle and low level are scaling factor that decrease the size of the frame. Given a camera model, these values can be determined experimentally as described in Section 3.4.

Directly connected with the light factor, is the level of details in the scene d_s. The scene detail level represents how many details there are in a scene, and can be measured in the field based on the different scenes. The resulting value is a number $d_s \in \mathbb{R}^+ \,|\, 500 \leq d_s \leq 2000$ expressed in millibits per pixel. Section 3.4 describes how to conduct field measurements.

We have experimentally found that the detail influence is also highly correlated to the amount of nature in the scene—lawns, bushes, trees, and similar. These features increase the difficulty of the encoding process, forcing the encoder to include more details in the resulting image, especially in the presence of wind. A high level description of the scene (e.g., a road, a garden, an office) allows one to provide an estimate of the amount of nature present in the frames. The nature factor n is expressed as the portion of the scene that includes natural elements, $n \in \mathbb{R}^+ \,|\, 0 \leq n \leq 1$. It can be easily measured on the field by taking a frame and computing a rough estimate. Typically, indoor scenes have a nature factor $n = 0$, while forest scenes have a nature factor $n = 1$. Common values for an outdoor parking lot are between 0.5 and 1. The factor included in the computation is $(1 + n)$, as the presence of nature only adds complexity to the scene, compared to the baseline.

The camera properties should be taken into account when computing the detail influence. The factor d_c is used to scale the frame size taking into account factors like the sensor types, lenses properties, etc. The constant value d_c represents how well the camera captures the details in the scene and how sharp they are. A measure of d_c can be obtained with respect to a standard camera. The camera detail level d_c can be measured for a given camera as detailed in Section 3.4.

The dynamic range of the scene, together with the camera's ability of capturing it through various image enhancement techniques such as HDR is modelled using the enhancement factor, e. If one assumes that the different light ranges have the same bitrate characteristics and that the camera auto-exposure will select the range filling the most pixels then $e \in \mathbb{R}^+ \,|\, 1 \leq e \leq 2$.

Table 2 Terms and Constants used in the Estimation of the upper bound for the I-frame size.

Acronym or Symbol		**Brief Explanation**	**Range or Typical Values**
d_c	`[M]`	Camera detail level: camera specific constant that reflects the camera capacity to retain scene details	$d_c \in \mathbb{R}^+ \,\vert\, 0.1 \leq d_c \leq 10$
d_s	`[M]`	Scene detail level: indicates the total amount of details in the scene	$d_s \in \mathbb{R}^+ \,\vert\, 500 \leq d_s \leq 2000$
e	`[M]`	Enhancement factor, indicates the effectiveness of High Dynamic Range (HDR) or similar technology	$e \in \mathbb{R}^+ \,\vert\, 1 \leq e \leq 1.35$
h	`[K]`	Height of a frame in pixels	~200–4320
I^*	`[C]`	Upper bound on the size of I-frames	
I_a	`[C]`	Influence of camera and scene	
I_c	`[C]`	Influence of the compression level QP	
I_d	`[C]`	Influence of the detail level	
ℓ	`[M]`	Scene illumination: it indicates the luminance (amount of light) in the scene, lower values indicate less light	$\ell \in \mathbb{R}^+ \,\vert\, 0.25 \leq \ell \leq 1$
n	`[M]`	Nature factor: amount of nature (trees, bushes, etc) in the scene	$n \in \mathbb{R}^+ \,\vert\, 0 \leq n \leq 1$
$n_{c,\ell}$	`[M]`	Noise level: camera specific constant indicating the amount of noise in the camera, capturing characteristics like sensor size and type; lower values indicate indoor high light and higher values low-light environments	$n_{c,\ell} \in \mathbb{R}^+ \,\vert\, 1 \leq n_{c,\ell} \leq 500$
QP	`[K]`	Quantization Parameter: reflects the frame compression, higher numbers indicate more information loss	$\text{QP} \in \mathbb{N}^+ \,\vert\, 1 \leq \text{QP} \leq 51$
QP^{ref}	`[K]`	Reference value used in measurements for the Quantization Parameter QP	28
ΔQP	`[K]`	$\text{QP} - \text{QP}^{\text{ref}}$	
r	`[K]`	Frame resolution (number of pixels in the frame)	~64000–35389440
SAO	`[M]`	Size of Average Object: reflects the expected distance of an object in an image, determined by factors like the zoom level, field of view, and lens type, and placement of the camera	$\text{SAO} \in \mathbb{R}^+ \,\vert\, 0.5 \leq \text{SAO} \leq 1.5$
w	`[K]`	Width of a frame in pixels	~320–8192

There are two corner cases, 1 and 2. $e = 1$ describes a scene with no additional dynamic range to capture, such as an indoor scene or a foggy day scene. $e = 2$ describes a scene where half the the frame is low dynamic and the other half is high dynamic, such as an indoor scene with large windows. An average value for all real world scenarios lays in between the two. The cameras that we tested had on average a 35% larger I-frame size when HDR was enabled, inducing $e \in \mathbb{R}^+ \,|\, 1 \leq e \leq 1.35$.

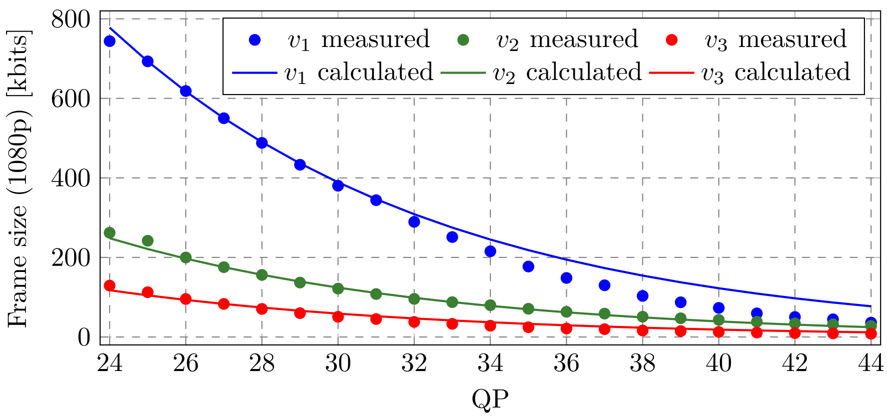


Figure 3 Measured I-frame sizes and calculated ones for different videos, varying QP.

Another important factor affecting the I-frame size via I_d is the size of typical objects and details in the scene, denoted with the term SAO. This parameter can be approximated based on a combination of the distance to the scene, the zoom level and the field of view. The effect of this is to reduce the I-frame size for scenes where the objects are large, since the amount of details in a typical object usually does not scale with resolution. Section 3.4 provides an explanation of how to estimate this parameter.

The last parameter that we need to include is $n_{c,\ell}$, which captures the influence of noise generated in the camera (which in the end influences the size of the I-frame). We assume that the camera is the only source of noise, but the parameter value varies with the amount of light ℓ. In fact, the amount of noise is in direct relation to the scene noise level. The more light there is, the more sensor saturation, the more photons the sensor receives, and the less noticeable the camera noise becomes. The noise level is heavily camera dependent, and related to both hardware (optics and sensor) and software (exposure strategies, noise filtering technologies, and image settings). Depending on the different light conditions ℓ, the noise level can be measured. Values are $n_{c,\ell} \in \mathbb{R}^+ \,|\, 1 \leq n_{c,\ell} \leq 500$. The procedure to measure $n_{c,\ell}$ is described in Section 3.4.

Considering all the contributions to the upper bound estimate I^*, and substituting I_d and $n_{c,\ell}$ in Equation (3), we can finally write

$$I^* = \dots = \left\{ r \cdot 2^{-\frac{\Delta\text{QP}}{6}} \cdot (\ell \cdot d_s \cdot (1+n) \cdot d_c \cdot e \cdot \text{SAO} + n_{c,\ell}) \right\}, \tag{4}$$

obtaining our desired expression for the I-frame size upper bound estimate.

Figure 3 illustrates the results that we obtain using Equation (4) with a default camera. The figure represents data obtained with three different 1080p videos: v_1, v_2, and v_3. The videos were encoded using different QP values in a standard setup where we know lighting conditions, detail level of both the scene and the camera, the size of objects, the enhancement features and the noise. We record I-frame sizes during the encoding with varying QP values, shown as dots in the Figure. The three lines represent the estimation obtained with Equation (4), which upper bounds the dot in almost every case.

3.3 Inter Frame Model – P-frames and B-frames

The same reasoning we used to estimate the upper bound of I-frames can be used to estimate the upper bound of the size of frames that can be encoded referencing macroblocks in other frames. The three components that provide contributions to the size of a P- and

Table 3 Additional Terms and Constants used in the upper bound for the P-frame size.

Acronym or Symbol		Brief Explanation	Range or Typical Values
$f_{\inf}$	[K]	Inferior frame rate limit	2
$f_{\sup}$	[K]	Superior frame rate limit	120
f_s^{ref}	[K]	Reference frame rate used for the motion level measurement	30
f_s	[K]	Number of frames per second in the video (saturated)	$f_s \in [f_{\inf}, f_{\sup}]$; ~20–40
P^*	[C]	Upper bound on the size of P-frames	
P_a	[C]	Influence of camera and scene	
P_c	[C]	Influence of the compression level QP	
P_d	[C]	Influence of the detail level	
P_m	[C]	Influence of motion	
μ_s	[C]	Motion level: fraction of the image that is expected to be moving	$\mu_s \in \mathbb{R}^+ \mid 0 \le \mu_s \le 1$
μ_x	[M]	Motion encoder efficiency: reflects the ability of efficiently encode moving object, an encoder with a large motion search window will have a low motion cost	$\mu_x \in \mathbb{R}^+ \mid 0 \le \mu_x \le 1$

B-frame are the same. We use P-frames as our basis, as we expect the encoder to be slightly more successful in encoding B-frames, therefore P-frames should represent an upper bound estimates for B-frames too. Table 3 summarizes the additional terms that are defined in this Section.

Using scaling factors with respect to the resolution (as we did for the I-frame), we define

$$P^* = \{r \cdot P_c \cdot P_a\}\,. \tag{5}$$

The first element contributing to the size of the frame is the resolution of the image r. The second and third components respectively are related to compression (P_c) and to the actual parameters of the camera and scene (P_a).

P-frames are highly correlated with neighboring frames, due to the compression algorithm. This makes the compression factor for P-frames larger than the one for I-frames and $P_c < I_c$. The relation between the compression parameter (QP) and frame size that we used for I-frames does not apply for P-frames due to this correlation. We introduce this by changing the compression term (the base 5 experimentally achieved through curve fitting):

$$P_c = 5^{-\frac{\Delta\mathrm{QP}}{6}}\,. \tag{6}$$

P_a can again be split into two parts, one part relative to the influence of the detail level P_d and the noise $n_{c,\ell}$, which is the same term used for the I-frames, $P_a = P_d + n_{c,\ell}$. The difference between I_d and P_d, on the contrary, lies in the motion detected in the image. The encoding algorithm tries to find motion, starting from the same macroblock position in buffered images. We therefore encode $P_d = P_m \cdot I_d$, defining P_m as a multiplicative gain that explains the effect of motion on the resulting frame size, refining Equation (5) into

$$P^* = \{r \cdot P_c \cdot P_a\} = \left\{r \cdot 5^{-\frac{\Delta\mathrm{QP}}{6}} \cdot (P_m \cdot I_d + n_{c,\ell})\right\}. \tag{7}$$

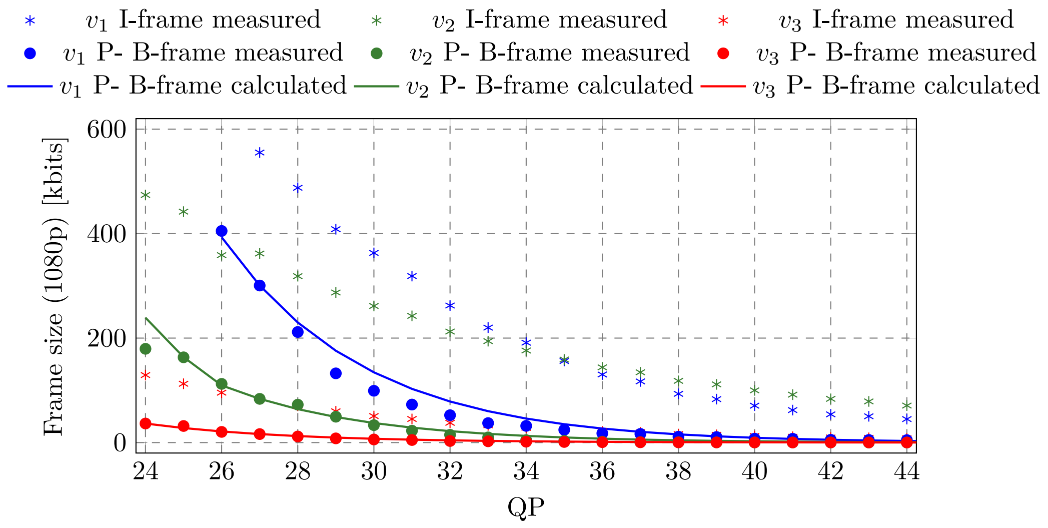


Figure 4 Measured P- and B-frame sizes and calculated ones for different videos, varying QP.

The influence of motion on the P-frame size P_m is affected by three factors: (i) the frame rate f_s, (ii) the scene motion level μ_s, and (iii) a camera motion cost, which reflects how well the H.264 encoder captures encoding of moving objects, which we call motion encoder efficiency μ_x.

- **Saturated frame rate f_s:** P_m is directly linked to the frame rate of the video: the lower the frame rate, the more difference there will be between consecutive frames, the larger the motion step will be and the more objects would have moved. This larger gap will translates into higher chances of a motion miss by the encoder, and leads to higher bandwidth consumption. At extremely high frequencies or extremely low frequencies, the frame rate effect saturates. We therefore impose thresholds on the frame rate, forcing it to belong to the interval $[f_{\text{inf}}, f_{\text{sup}}]$. We have experimentally determined good values for f_{inf} and f_{sup} and respectively set them to 2 and 120. Using experimental data, we have determined that P_m is proportional to the inverse square of the video frame rate $P_m \propto \sqrt{f_s}^{-1}$.

- **Scene motion level μ_s:** The motion level of a scene is a measurable quantity at a certain reference frame rate f_s^{ref}, in our case equal to 30. This means that $P_m \propto \mu_s \cdot \sqrt{f_s^{\text{ref}}}$. The motion level determines the portion of the image that has moved from one frame to the next. If accurately known, μ_s can be uniquely used and varied per frame. However, since the primary use case of our upper bound is to estimate the required bandwidth there is a strong added benefit in simplifying the analysis. For simplification, we only use a generic set of possible motion levels: high, medium, and low. For high motion scenes, μ_s is typically around 0.15. For medium motion scenes, its value is around 0.07, and for low motion scenes 0.01.

- **Motion encoder efficiency μ_x:** The motion encoder efficiency is a measurable quantity per camera. The camera encoding capabilities are often dependent on the encoder capabilities and efficiency. The motion encoder efficiency can be measured, as explained in Section 3.4.

Including all the terms specified above, one can write $P_m = \mu_s \cdot \mu_x \cdot \sqrt{f_s^{\text{ref}}/f_s}$, and therefore, substituting P_m in Equation (7), we obtain our upper bound estimate

$$P^* = \cdots = \left\{ r \cdot 5^{-\frac{\Delta\text{QP}}{6}} \cdot \left(\mu_s \cdot \mu_x \cdot \sqrt{f_s^{\text{ref}}/f_s} \cdot I_d + n_{c,\ell} \right) \right\}. \tag{8}$$

Figure 4 illustrates the results that we obtain using Equation (8) with a default camera with known parameters. The figure represents data obtained with three different 1080p videos, v_1, v_2, and v_3. The lines represent the estimation obtained with Equation (8), which upper bounds the measured values, plotted with dots. We also report the measured size of I-frames for the same videos with asterisks.

3.4 Model Calibration

As indicated above, different constants need to be measured for the various cameras and scenes, in order to be able to extract meaningful numbers for Equations (4) and (8). These characteristics can be grouped in different sets: (i) platform-related, (ii) camera-related, and (iii) scene-related.

- **Platform-related characteristics.** The motion encoder efficiency μ_x is related to the platform (mostly the encoder) that is being used. In principle, the scene is also important in this case, but a scene-independent approximation can be computed. For each encoder generation and brand, the estimation of μ_x is done by isolating the encoder, or an equivalent encoder model, with a series of predetermined video sequences, encoded using varying compression.
- **Camera-related characteristics.** The three characteristics that we need to measure among the camera-related ones are d_c, $n_{c,\ell}$, and e. They are respectively: (d_c) the ability of the camera to retain scene details, ($n_{c,\ell}$) the amount of noise that the camera generates in specific light conditions, and (e) the enhancement factor added by technology like HDR. These are constants that summarize many different physical elements like the sensor size and quality.
- **Scene-related characteristics.** Four scene-related characteristics should be measured: the scene level detail d_s, the amount of nature n, the Size of the Average Object in the scene, SAO, and the amount of light ℓ.

Measurements should be collected in a reproducible environment. In our case, we collected the data in a dedicated laboratory. The main idea is to be able to reproduce certain scene conditions. The environment must contain different levels of details. It should be possible to shoot videos of areas with few or no details, as well as others with many details. It should also be possible to control the amount of light, at least to reproduce three different light conditions — high, medium, and low. Finally, there should be some reproducible source of motion, e.g., a fan or a toy train. The position of the camera with respect to the scene should be fixed in advance and should be reproducible as well. Figure 5 shows the laboratory in which the tests to compute the above mentioned parameters were conducted. Most measurements are conducted using a reference camera, and then for a new camera some additional data is collected to compare the camera to the reference one.

To determine the parameters we follow a specific procedure, both for the reference camera and for the model that we are trying to profile: (i) we record (repeatable) scenes with no motion, motion, no details, details, in three different light levels; using the compression level QP^{ref}; (ii) we extract the frame sizes for all the I-frames and P-frames in the video; and (iii) we compute statistics for the videos, the average and maximum size of I- and P-frames.

Figure 5 Image laboratory used to determine characteristics related to the camera and the scene.

For the camera detail level d_c, we compute the average frame size (for all the set of recorded videos), including both I-frames and P-frames and compare them with the values obtained with the reference camera. Denoting with S_{avg} the average frame size of the camera under test and with $S^{\text{ref}}_{\text{avg}}$ the one of the reference camera, $d_c = S_{\text{avg}}/S^{\text{ref}}_{\text{avg}}$. We repeat the same considering only low light conditions, and compute $n_{c,\ell}$ exactly using the same formula. The value of e for a given camera is determined by computing the ratio of the average frame sizes with HDR activated and deactivated.

To compute scene-level measurements, there are two alternatives. The first one is to physically record videos from the location where the camera should be installed, and the second one is to film similar scenes multiple times, and re-use the average measured parameter for similar scene types. We denote with $S_{I,\text{avg}}$ the average size of I-frames measured in bits for these measurements. We also want to collect videos done with the zoom level set to 50% for this case. The average size of the I-frames for this zoom level is indicated with $S_{I,50\%,\text{avg}}$.

The reference camera is used to measure the scene detail level d_s. Using the set of videos recorded from similar or the same scene, d_s is computed as the average size of I-frames expressed in millibits per pixel, $d_s = S_{I,\text{avg}} \cdot 1000/r$. The scene illumination ℓ is measured by comparing the laboratory result with the scene results using the actual camera to be used. From the laboratory results, we take videos recorded in high illumination scenes and compute the average size of I-frames for these videos as $S_{I,\ell=1,\text{avg}}$. We then compute ℓ as $\ell = S_{I,\ell=1,\text{avg}}/S_{I,\text{avg}}$. The amount of nature n is computed by looking at how many pixels in a frame are covered by nature.

Finally, we need to measure the size of the average object SAO. SAO is determined as $S_{I,50\%,\text{avg}}/S_{I,\text{avg}}$. The SAO levels can be, for simplicity, divided into three levels: large, medium, and small. As a general rule of thumb, one can determine the SAO level for 1080p video such as: (i) Large SAO: Objects taking up more than 1% of the pixels. An example is a licence plate camera, commonly setup to capture mainly a car with sufficient margin around it. (ii) Medium SAO: Objects are between 1% and 0.01% of the pixels. This is the most common case. (iii) Small SAO: Objects are very small, less than 0.01% of the pixels. This is sufficient only for scene awareness, i.e. knowing what happened in the scene, but does not permit to identify objects.

4 Related Work

The ultimate goal of this paper is to enable scheduling of network bandwidth in a video-surveillance system, utilizing the available bandwidth as much as possible. This goal can be achieved in many different ways.

One alternative to better utilize network resources is to reduce the amount of sent information by exploiting better compression and enhanced encoding. A lot of research has been devoted to adapting video stream quality to fit network channels [1, 12, 13, 20, 21, 23]. For example, adaptive strategies have been developed for MJPEG encoding [1, 23], MPEG-2 [12], and MPEG-4 [13]. Another alternative offer variable network channels [22, 29]. In this work, we investigate estimation of the WCTT for frames over a network, which is related to these works, but takes a different route. The aim of this paper is to devise a reasonably accurate model to aid scheduling decisions, without introducing adaptation.

To the best of our knowledge, there are two known alternative methods to estimate the frame size, and in turn the expected video bandwidth needed for the video transmission. These methods are based on other encoding methods (respectively MJPEG and MPEG-4) and aim to provide an estimate of the expected frame sizes. To the best of our knowledge, we propose the first frame size estimation for MPEG-4 part 10 AVC (H.264).

We denote the MJPEG method with `LIN`. This method only considers the compression parameter (QP for H.264 videos), and scales the frame size linearly according to such a parameter that we name q_l. Given a maximum size, identified with the term s_{max}, the frame size $s(q_l)$ is computed as $s(q_l) = q_l \cdot s_{max}$. The parameter q_l indicates the quality of the encoding, and relates, as indicated previously, to the Quantization Parameter QP. The scale and logic used are different and in MJPEG $q_l \in [0.01, 1.0]$, 1 being the lowest compression and 0.1 the highest, therefore $q_l = 1.01 - (\text{QP}/51)$. In the case of a 1080p YCbCr color video with 8 bits per pixel, $s_{max} = 1920 \cdot 1080 \cdot 8 \cdot 3 = 49766400$ [bits per frame]. This model is used for example in [22, 23] to devise a control strategy to determine the quality to be applied given a target bandwidth consumption.

We call the MPEG-4 model `RQM`. This model is used in [1] and described in [8]. It uses curve fitting to determine the parameters of a rate-distortion curve, modeled with a Gaussian random variable. Denoting with α a constant accounting for overhead bits, with β a constant that varies with the resolution and amount of motion in the video, with q_r the compression level for MPEG-4 ($q_r \in [1, 31]$), and with γ a constant that varies depending on the frame type (paper [8] providing recommended bounds of $\gamma \in [0.5, 1]$ for I-frames and $\gamma \in [0.5, 1.5]$ for P-frames), the size of the frame can be written as $s(q_r) = \alpha + \beta \cdot 1/{q_r}^{\gamma}$.

Notice that neither `LIN`, nor `RQM` compute proper upper bounds. They rather compute estimates of the frame size. We therefore do not expect them to be suitable for upper bounding the size of frames and obtaining WCTTs.

5 Experimental Results

In this section we present our experimental evaluation. We conducted many tests with different cameras and in different scenarios to validate the upper bounds estimates computed with our technique. We present two different categories of tests. Section 5.1 shows the results obtained for a controlled environment and a repeatable video, comparing our estimation strategy with state-of-the-art techniques. Section 5.2 presents a stress-test where we report the aggregate results of a large experimental campaign.

To conduct a comprehensive evaluation, we used 6 different camera models, and deployed them in 24 real-life (surveillance) scenarios. We refer to the different camera models using

Table 4 Measured camera-related parameters.

Model	d_c	$n_{c,l}$ (high ℓ)	$n_{c,l}$ (medium ℓ)	$n_{c,l}$ (low ℓ)	μ_x
A	1.00	2.50	2.75	22.2	0.450
B	0.98	0.25	2.75	230.0	0.450
C	1.23	0.35	1.10	102.0	0.450
D	0.54	0.75	4.05	5.6	0.400
E	0.81	1.25	12.00	35.0	0.400
F	1.03	2.25	2.7	119.0	0.425

letters from A to F. Camera A was used as reference camera for the parameter estimation discussed in Section 3.4. To show the versatility of the model we use different parameters, resolutions, etc. Also, Camera C is a thermal camera. Table 4 contains the camera-related parameters that do not change with the scenario. Parameters that change with the scenario will be discussed in the corresponding sections.

5.1 Frame-by-Frame Evaluation

We present here a first validation experiment done with our reference Camera A. We recorded two videos of the same scene in our laboratory. The scene has a lot of details. The laboratory allows us to move the camera with predictable motion and control the amount of movement introduced in the image. Our aim is to show a frame-by-frame comparison between our frame size estimation and the state-of-the-art techniques discussed in Section 4.

The two videos differ in the amount of motion that is introduced[3]. A toy, present in the scene, allows us to introduce very limited but non-zero motion in both cases. In the first video, we also sharply changed the position of the camera. This simulates a fast movement for a video-surveillance camera. In the second video we kept the camera still, thus the only movement comes from the toy. The first video is characterized by a large amount of motion μ_s, while the second video has a very low μ_s.

The Camera A parameters for the two videos are: camera level detail $d_c = 1$, enhancement factor $e = 1.35$ (HDR), width $w = 1920\,$[pixels], height $h = 1080\,$[pixels], frame rate $f_s = 25\,$[frames per second], QP $= 29$, noise level $n_{c,\ell} = 2.5$, motion encoder efficiency $\mu_x = 0.45$, GOP $= 64$. The scene parameters are: no nature, $n = 0$, very good illumination, $\ell = 1$, scene detail $d_s = 780\,$[millibit per pixel], and size of the average object SAO $= 1$.

Figure 6 shows the results we obtained for the two videos. Each plot represents 200 frames of one video, the top one being the high-motion one and the bottom one being the low-motion case. The black bars represent the real frame sizes measured after the encoding. The circles represent the estimated upper bound on the frame sizes provided by the algorithm presented in this paper. The squares show the estimate produced by the `LIN` model, which does not take into account the difference between I, P, and B frames. Finally, the squares represent the estimate produced by the `RQM` model.

[3] The two videos are available online: `https://www.youtube.com/watch?v=614BbbhD56M` (high-motion), and `https://www.youtube.com/watch?v=q4j3L1Vr0ls` (low-motion). We have manipulated them to also visually show the motion vectors detected for both the original videos: `https://www.youtube.com/watch?v=5YrxlGhadsY` (high-motion), and `https://www.youtube.com/watch?v=cfr08CZQa-E` (low-motion)

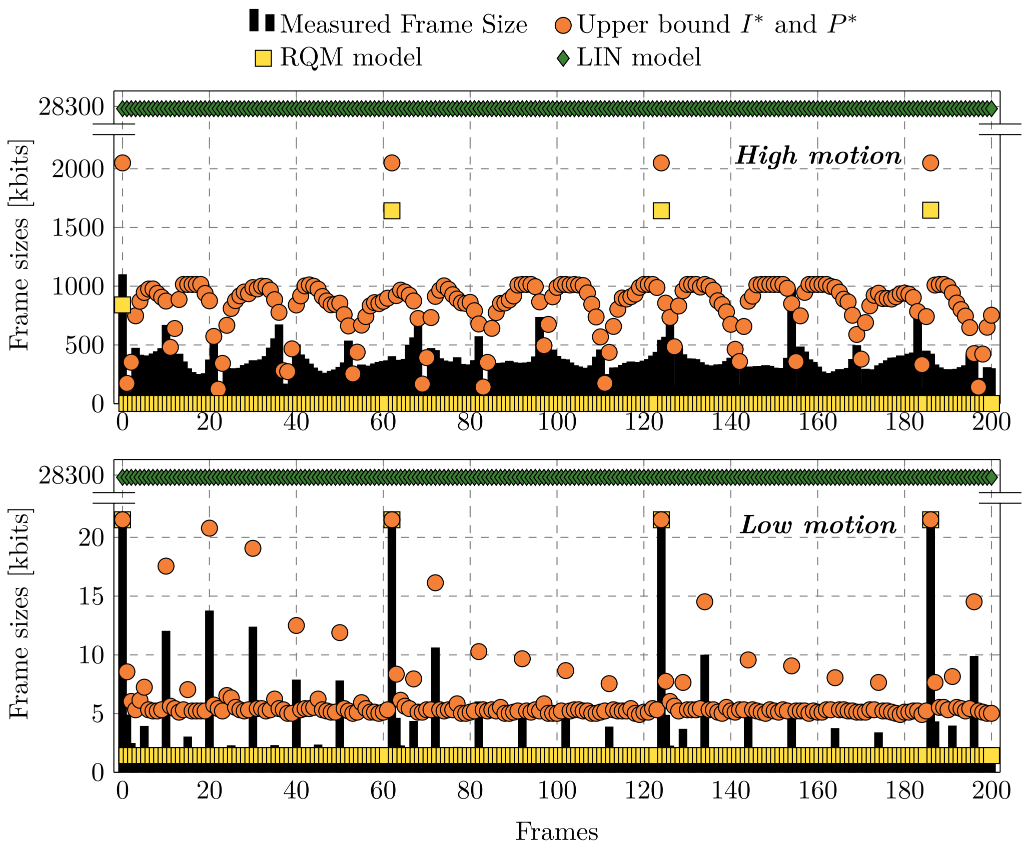


Figure 6 Results of the comparison experiment with the high- and low-motion video.

For the RQM model, we used to the low-motion video to tune the parameters α, β, and γ, as recommended in [1]. The tuning resulted in $\alpha = 0.55$ and $\beta = 1.7$. As γ changes depending on the frame type, we fit $\gamma_I = 0.5$ and $\gamma_P = 4$ separately. The RQM tuning resulted in average errors on I-frames and P-frames respectively of 1.80% and 1.38%, which indicate very good performance for the low motion video. The square points in the lower plot of Figure 6 are therefore *a posteriori* estimations, and are clearly a very good fit for the video, despite the presence of a few outliers. The RQM model neglects motion — i.e., the β parameter is not sufficient to take motion into account. In fact, when the parameters determined with the low-motion video are used for *a priori* estimating the size of the frames in the high-motion video, the estimate frame size greatly underestimates the real value. The RQM approximation is therefore not a good fit to upper bound the size of the frames.

On the contrary, the LIN model gives very conservative results for both the high- and low-motion video, as its only parameter is a translation of the encoding quality QP. These are too conservative to be used in any practical setting, since the estimates are roughly 30 times as large as the real values. The LIN approximation is therefore also not a good upper bound for the size of the frames.

In the case of our upper bound estimates I^* and P^*, the circles represent for both plots *a priori* estimates based on the parameters that we have selected and on a standard computation of the motion level μ_s based on the percentage of pixels that differ from one image to the next (which could be determined before the encoding step). Roughly, the computed upper bound estimates are twice as large as the real values. While this could be

Table 5 Parameters and results of the experiments conducted with 6 cameras in 24 real-life surveillance scenarios.

	f_s	**QP**	**GOP**	μ_s	ℓ	d_s	**SAO**	b_r	$\hat{b_r}$
1a (A)	25	28	62	$\approx 1\%$	1	780	1.00	1040	1275
1b (A)	25	28	62	$\approx 3\%$	1	780	1.00	1600	1806
1c (A)	25	28	62	$\approx 9\%$	1	780	1.00	3200	3398
1d (A)	12	32	32	$\approx 1\%$	1	780	1.00	544	600
1e (A)	12	32	32	$\approx 3\%$	1	780	1.00	720	723
1f (A)	12	32	32	$\approx 11\%$	1	780	1.00	1200	1219
2a (B)	15	28	62	$\approx \{2,3\}\%$	$\{1, 0.8\}$	810	1.00	794	991
2b (B)	15	28	62	$\approx 0\%$	1	710	0.45	78	208
2c (C)	15	28	62	$\approx 1\%$	0.8	820	0.45	243	287
2d (A)	15	28	62	$\approx \{3,5\}\%$	$\{1, 0.8\}$	990	0.45	669	765
2e (C)	15	28	62	$\approx 1\%$	1	810	1.00	513	761
2f (C)	15	28	62	$\approx 1\%$	$\{1, 0.8\}$	1400	1.00	333	490
2g (C)	15	28	62	$\approx 5\%$	1	920	0.45	409	456
2h (F)	15	28	62	$\approx 0\%$	0.8	710	0.45	45	96
2i (A)	15	28	62	$\approx 0\%$	$\{1, 0.5\}$	780	1.10	722	793
2j (F)	15	28	62	$\approx 4\%$	0.8	780	1.00	139	144
2k (A)	15	28	62	$\approx \{4,3\}\%$	$\{1, 0.5\}$	780	1.00	194	220
3a (A)	25	28	32	$\approx 21\%$	1	1200	1.00	10000	10051
3b (A)	25	28	32	$\approx 4\%$	1	1200	1.00	2800	3116
4a (C)	30	18	32	$\approx 6\%$	1	660	1.00	4215	4551
4b (C)	30	18	32	$\approx 2\%$	0.5	780	1.00	4966	5321
5 (D)	25	24	4	$\approx 2\%$	1	990	1.00	42500	46529
6 (E)	25	32	32	$\approx 4\%$	0.5	660	1.00	2837	2878
7 (A)	15	36	30	$\approx 20\%$	1	1050	1.00	620	681

reduced with a more conservative setup of parameters, we believe that there could be a risk of cases in which the real frame size exceeds the upper bound estimate. In the full length of the two videos (low-motion 751 frames, high-motion 376 frames) this never happens for the low-motion case, and happens five times for the high-motion case. Inspecting these five occurrences prompted us to suspect some capturing error or some encoding miss, possibly due to the sharp movement.

5.2 Stress test

The purpose of the stress test is to verify that we obtain a reasonably good estimate of the bandwidth consumed by cameras to transmit their frame streams to a base station. We deployed our cameras in real-life surveillance scenarios and collected video streams for a time up to five days. We then measured the expected bandwidth consumption using estimates of the parameters (e.g., instead of computing precisely the motion level μ_s, we guessed it based on the type of recorded scene). We compared the measure expected bandwidth with the real bandwidth requirements — the videos' bitrates. The characteristics of the tested scenarios and the obtained results are summarized in Table 5, where b_r represents the bitrate, and $\hat{b_r}$ its estimate.

The scene in scenarios 1a–1f is a highly illuminated parking lot, recorded with camera A ($e = 1.35, w = 1920, h = 1080$). Scenarios 2a–2k are videos from the surveillance system of a hotel complex. Camera B ($w = 1920, h = 1080$) in scenario 2a points at the reception entrance. In scenario 2b, Camera B ($w = 1920, h = 1080$) captures the emergency exit. Camera C ($w = 1920, h = 1080$) in scenario 2c films the control room. Camera A ($w = 1920, h = 1080$) in 2d is directed to the parking entrance. Camera C ($w = 1920, h = 1080$) in 2e films the reception. Camera C ($w = 1280, h = 720$) in 2f captures the corridor with shops. In 2g, Camera C ($w = 1280, h = 720$) is directed towards the elevator. Camera F in 2h films the staircase. Camera A ($w = 1280, h = 720$) in 2i streams a parking lot with nature $n = 0.5$. Camera F ($w = 704, h = 480$) in 2j and Camera A ($w = 704, h = 480$) in 2k film parking lots without nature. When the table contains two numbers for the motion level μ_s and for the light ℓ, this means that in the estimation the numbers are adjusted for day and night capture. The set includes first the day and then the night value. The value of e is set to 1 for 2b, 2c, 2e, 2f, 2h, 2j, which means HDR is turned off. In the other scenarios, HDR is turned on with a contribution of $e = 1.35$. The two instances of Camera A ($e = 1.35, w = 1920, h = 1080$) used in scenario 3a and 3b are placed in bridges on the highway and monitor car traffic. The two instances of Camera C ($e = 1$) of scenario 4a and 4b monitor a perimeter of a parking lot and the parking lot itself. In 4a the resolution is set to $w = 640, h = 480$, while in 4b the resolution is set to $w = 384, h = 288$. In scenario 5, Camera D ($e = 1, w = 3840, h = 2160$) streams a 4k video of the corner of a city street. Camera E ($e = 1.35, w = 3072, h = 1728$) in scenario 6 is filming a shipyard loading dock. Finally, in scenario 7 Camera A ($e = 1.35, w = 1280, h = 720$) is facing a city intersection.

Despite the high variety of scenes, the varying light conditions, the different cameras, and the different motion levels, the estimated bitrate $\hat{b_r}$ (upper bound estimate) is always higher than the measured bitrate b_r. In most cases, the two values are very similar to one another (see for example scenario 1e or 3a). In a few cases, like 2b and 2h, it is possible to see that the upper bound overestimates the video bitrate (respectively 2.65 and 2.13 times as large). However, we believe these numbers provide a reasonable upper bound estimate and permit to correctly dimension the network bandwidth, aiding scheduling decisions.

6 Conclusion and Future Work

In this paper we presented a practical contribution on how to derive upper bounds estimates for the size of video frames in a streaming system. We have discussed which characteristics influence the bandwidth requirements of different cameras, derived models for the upper bound estimates of the size of I-, P-, and B-frames. We have also systematized the knowledge on the involved quantities and parameters. We divided such quantities into parameters that are known, characteristics that are measurable, and values that are computable. We have then taken the measurable characteristics and discussed how to conduct field tests to obtain reasonable values for them, and — when possible — how to guess based on the environmental conditions. Some parameters can be more or less easily estimated online (motion, light level, noise level, scene type...). Estimating these parameters on the source could lead to a more accurate and less pessimistic short term prediction. More frame by frame tests as well as highly challenging scenarios will also be ran in order to enhance the model.

The derivation of reasonable upper bounds estimates for the WCTT allows us to precisely formulate the problem of allocating network bandwidth to a set of cameras in a switched Ethernet network environment and to reuse well-known scheduling results. We have shown with a thorough experimental campaign that our estimated upper bounds are more reliable, and closer to the real frame sizes than state-of-the-art estimation techniques.

A proper estimation of the frame sizes is the key to properly dimension network infrastructures for real-time video-surveillance systems. Our results demonstrated that we can dimension the network infrastructure, being able to accurately predict the bitrate consumption of video streams. Our findings have a significant industrial relevance, as they permit to reduce the infrastructure cost and allows us to reuse known scheduling results.

References

1 Luís Almeida, Paulo Pedreiras, Joaquim Ferreira, Mário Calha, José Alberto Fonseca, Ricardo Marau, Valter Silva, and Ernesto Martins. Online QoS adaptation with the flexible time-triggered (FTT) communication paradigm. In *Handbook of Real-Time and Embedded Systems*, 2007.

2 Björn Andersson. Schedulability analysis of generalized multiframe traffic on multihop-networks comprising software-implemented ethernet-switches. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008. `doi:10.1109/IPDPS.2008.4536565`.

3 Björn Andersson, Sagar Chaki, Dionisio de Niz, Brian Dougherty, Russel Kegley, and Jules White. Non-preemptive scheduling with history-dependent execution time. In *24th Euromicro Conference on Real-Time Systems*, pages 363–372, July 2012. `doi:10.1109/ECRTS.2012.38`.

4 Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999. `doi:10.1023/A:1008030427220`.

5 Sanjoy Baruah, Deji Chen, and Aloysius Mok. Static-priority scheduling of multiframe tasks. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 38–45, 1999. `doi:10.1109/EMRTS.1999.777448`.

6 Sanjoy K. Baruah and Samarjit Chakraborty. Schedulability analysis of non-preemptive recurring real-time tasks. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006. `doi:10.1109/IPDPS.2006.1639406`.

7 Samarjit Chakraborty and Lothar Thiele. A new task model for streaming applications and its schedulability analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 486–491, Washington, DC, USA, 2005. IEEE Computer Society. `doi:10.1109/DATE.2005.26`.

8 Wei Ding and Bede Liu. Rate control of mpeg video coding and recording by rate-quantization modeling. *IEEE transactions on circuits and systems for video technology*, 6(1):12–20, 1996.

9 Pontus Ekberg, Nan Guan, Martin Stigge, and Wang Yi. An optimal resource sharing protocol for generalized multiframe tasks. *Journal of Logical and Algebraic Methods in Programming*, 84(1):92–105, 2015. `doi:10.1016/j.jlamp.2014.10.001`.

10 Ching-Chih Jason Han. A better polynomial-time schedulability test for real-time multi-frame tasks. In *Proceedings 19th IEEE Real-Time Systems Symposium*, pages 104–113, Dec 1998. `doi:10.1109/REAL.1998.739735`.

11 ITU-T. Advanced video coding for generic audiovisual services. `https://www.itu.int/rec/T-REC-H.264-201704-I/en`, 2017.

12 Anand Kotra and Gerhard Fohler. Resource aware real-time stream adaptation for mpeg-2 transport streams in constrained bandwidth networks. In *IEEE International Conference on Multimedia and Expo*, pages 729–730, July 2010. `doi:10.1109/ICME.2010.5583196`.

13 Anand Kotra and Gerhard Fohler. Resource aware real-time stream adaptation of mpeg-4 video in constrained bandwidth networks. In *Visual Communications and Image Processing*, pages 1–4, Nov 2011. `doi:10.1109/VCIP.2011.6116008`.

14 Shuai Li, Stephane Rubini, Frank Singhoff, and Michel Bourdelles. A task model for tdma communications. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 1–4, June 2014. `doi:10.1109/SIES.2014.7087455`.

15 Wan-Chen Lu, Kwei-Jay Lin, Hsin-Wen Wei, and Wei-Kuan Shih. New schedulability conditions for real-time multiframe tasks. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 39–50, July 2007. `doi:10.1109/ECRTS.2007.20`.

16 Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, Oct 1997. `doi:10.1109/32.637146`.

17 Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG. *"Draft ITU-T recommendation and final draft international standard of joint video specification (ITU-T Rec. H.264/ISO/IEC 14496-10 AVC)"*, 2003.

18 Paulo Pedreiras and Luís Almeida. The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems. In *International Parallel and Distributed Processing Symposium*, 2003.

19 Bo Peng and Nathan Fisher. Parameter adaption for generalized multiframe tasks and applications to self-suspending tasks. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 49–58, Aug 2016. `doi:10.1109/RTCSA.2016.15`.

20 Naomi Ramos, Debashis Panigrahi, and Sujit Dey. Dynamic adaptation policies to improve quality of service of real-time multimedia applications in IEEE 802.11e WLAN networks. *Wireless Networks*, 13(4):511–535, 2007. `doi:10.1007/s11276-006-9203-5`.

21 Bernhard Rinner and Wayne Wolf. An introduction to distributed smart cameras. *Proceedings of the IEEE*, 96(10), 2008.

22 Gautham Nayak Seetanadi, Javier Cámara, Luís Almeida, Karl-Erik Årzén, and Martina Maggio. Event-driven bandwidth allocation with formal guarantees for camera networks. In *IEEE Real-Time Systems Symposium*, 2017.

23 Gautham Nayak Seetanadi, Luis Oliveira, Luis Almeida, Karl-Erik Arzen, and Martina Maggio. Game-theoretic network bandwidth distribution for self-adaptive cameras. In *15th International Workshop on Real-Time Networks*, 2017.

24 Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 71–80, Washington, DC, USA, 2011. IEEE Computer Society. `doi:10.1109/RTAS.2011.15`.

25 Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. Popularity prediction of facebook videos for higher quality streaming. In *USENIX Annual Technical Conference, USENIX ATC*, 2017.

26 Laszlo Toka, András Lajtha, Éva Hosszu, Bence Formanek, Dániel Géhberger, and János Tapolcai. A Resource-Aware and Time-Critical IoT framework. In *IEEE International Conference on Computer Communications*, 2017.

27 Bobby Vandalore, Wu-Chi Feng, Raj Jain, and Sonia Fahmy. A survey of application layer techniques for adaptive streaming of multimedia. *Real-Time Imaging*, 7(3), 2001.

28 Vilas Veeraraghavan and Steven Weber. Fundamental tradeoffs in distributed algorithms for rate adaptive multimedia streams. *Computer Networks*, 52(6), 2008.

29 Xiaorui Wang, Ming Chen, Huang-Ming Huang, Venkita Subramonian, Chenyang Lu, and Christopher D. Gill. Control-based adaptive middleware for real-time image transmission over bandwidth-constrained networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(6), 2008.

30 Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, jul 2003. `doi:10.1109/TCSVT.2003.815165`.

31 Haibo Zeng and Marco Di Natale. Outstanding paper award: Using max-plus algebra to improve the analysis of non-cyclic task models. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 205–214, July 2013. doi:10.1109/ECRTS.2013.30.

32 Areej Zuhily and Alan Burns. Exact scheduling analysis of non-accumulatively monotonic multiframe tasks. *Real-Time Systems*, 43(2):119–146, 2009. doi:10.1007/s11241-009-9085-6.

Early Design Phase Cross-Platform Throughput Prediction for Industrial Stream-Processing Applications

Tjerk Bijlsma
ESI
High Tech Campus 25, 5600 HE, Eindhoven, The Netherlands
tjerk.bijlsma@tno.nl

Alexander Lint
Océ Technologies
P.O. Box 101, 5900 MA, Venlo, The Netherlands
alexander.lint@oce.com

Jacques Verriet
ESI
High Tech Campus 25, 5600 HE, Eindhoven, The Netherlands
jacques.verriet@tno.nl

Abstract

Industrial embedded platforms are often used to execute stream-processing applications, from which the results are used by actuators. On average, these stream-processing applications should at least meet the required throughput of their actuators, which poses a real-time requirement on the system. To avoid extra costs and delays, it is desired to estimate during the early design phase if a combination of an embedded platform and a stream-processing application can achieve the required throughput. The throughput of a stream-processing application executed on different embedded platforms can be predicted by modeling them using static or measurement based analysis. However, during the early design phase it can be desirable to have a model that allows a large set of embedded platforms to be considered, where embedded platforms with predictive instructions are supported.

This paper presents a gray-box approach applicable during the early design phase to perform cross-platform throughput predictions for industrial stream-processing applications and their embedded platforms. A three step regression-based approach is presented, which uses an expression based on Amdahl's law for the discrete scaling of workload over cores and a large database with CPU performance scores to perform cross-platform throughput predictions. Validation, with a limited set of platforms, showed the usability of the approach. The pragmatic approach is based on a prototype industrial digital image processing application for a printer from Océ, which is also used to present the approach.

2012 ACM Subject Classification General and reference → Estimation,General and reference → Performance,Computer systems organization → Real-time systems

Keywords and phrases throughput prediction, stream-processing application, early design phase, regression model, cross-platform

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.18

Acknowledgements The research is carried out as part of the Octo+ program under the responsibility of Embedded Systems Innovation by TNO (ESI) with Océ Technologies B.V. as the carrying industrial partner. The Octo+ research is supported by the Netherlands Ministry of Economic Affairs.

1 Introduction

Industrial embedded platforms typically execute applications that process streams of information, from which the results are used by an actuator. Variations in processing time can often be absorbed by a small buffer between the stream-processing application and the actuator. However, the stream-processing application should at least achieve the required throughput, such that outputs are available at a fixed rate for the actuator. This throughput puts a real-time requirement on the system, where not meeting the deadline a few times in a row results in an empty buffer towards the actuator. It is necessary to know if an embedded platform in combination with a stream-processing application can achieve its throughput and real-time requirements, as early as possible during the design time of such industrial systems. Not achieving the required throughput either influences the specifications of the industrial system, or causes quality loss for the industrial system. An alternative is to over-dimension the embedded platform, which increases production costs and thereby harms the competitiveness of the industrial system.

A trend is the constant increase of actuation speed and quality for stream-processing applications, thereby requiring more complex processing and a higher throughput. Often architecture or design patterns [4] are applied for stream-processing applications to manage their complexity and achieve scalability. The *Master Slave* pattern is often applied for scalability, as it allows data parallelism to be exploited by using more slave threads, as suggested by Amdahl's law [2]. The *Pipes and Filter* pattern manages complexity by clustering parts of the stream-processing application as filters, thereby allowing function parallelism to be exploited. Stream-processing applications using these patterns are applied throughout industry. Applying these patterns, manages the complexity and enables scalability of the throughput for these stream-processing applications. However, in addition to enabling scalability, knowledge of the achievable throughput for target embedded platforms is desirable. Having the knowledge of achievable performance on different embedded platforms during the early design phase enables Design Space Exploration (DSE), such that a trade-off between achievable throughput and costs can be made.

The throughput of a stream-processing application executed on different embedded platforms can be predicted by modeling their combinations using static analysis or by using measurement based analysis [26]. However, using static analysis for throughput prediction requires worst-case execution times [26], which can be hard to derive for CPUs of embedded platforms that use techniques like caching and predictive instructions. Measurement based analysis requires measurements on considered target CPUs to predict execution times, such that only physically available target CPUs and platforms can be considered. During the early design phase a light-weight analytical model [17, 13] for the performance of the combined application and embedded platform can be desirable, to be able to easily consider a large set of combinations. Preferably gray-box application and platform knowledge is sufficient to create such a model, where measurements on a *reference* platform suffice to predict the throughput for a large set of *target* platforms.

This paper presents a gray-box approach applicable during the early design phase to perform cross-platform throughput predictions for industrial stream-processing applications and their embedded platforms. This pragmatic approach is based on the early design phase of a real prototype Océ digital image processing application for a printer with real-time requirements, which is also used as running example. For stream-processing applications that implement the Master Slave design pattern, a three step regression-based approach is presented. The prediction uses an extended expression based on Amdahl's law that considers the discrete scaling of workload over slaves, which is to the best of our knowledge a novel contribution. A database with CPU performance scores is used for the cross-platform throughput prediction, enabling predictions for a very large set of CPUs and embedded platforms. Suitability of the approach has been validated, by using measurements for a limited set of platforms. Additionally, DSE can be performed to select a cost-effective embedded platform that delivers the required throughput. For its performance, the digital image processing application requires predictive instructions and caching, such that x86 instruction set based CPUs are considered.

The outline of this paper is as follows. In Section 2 related work is presented, followed by an overview of the throughput-prediction approach in Section 3. Following, the single-core execution time prediction is explained in Section 4. Section 5 presents the throughput prediction for multiple cores in an embedded platform. Next, the cross-platform throughput predictions are presented in Section 6 and validated in Section 7. A design space exploration is presented in Section 8. Finally, conclusions and future-work are discussed in Section 9.

2 Related Work

Typically, a model is used to predict the throughput of a stream-processing application, allowing predictions for different embedded platform configurations and input streams. The model to perform throughput predictions should be chosen depending on the internal application structure and the development phase. The gut feeling of the designer and back-of-the-envelope models are straightforward and suitable during the early design phase, for applications with a clear relation between the execution time and the features of their input stream. A small amount of time is needed to create and use such models, however, their accuracies vary. Detailed static analysis models, like data-flow models [25] or discrete-event simulation [13, 12] can capture complex interactions and perform accurate predictions for an application. However, creating such models requires a fair amount of time and worst-case execution times, which make them less suitable for the early design phase in which the application can change a lot. In [24], detailed modeling of applications in a real-time system and their contention during the early design phase is discussed. This approach focuses on detailed models of applications on a single platform and their contention to ease integration, rather than to predict performance. The approach presented in [11] models an application as a process network with worst-case and best-case execution times for the tasks together with the scheduling policy for the considered platform. This requires detailed knowledge of the underlying platform and does not allow predictions for other platforms without constructing a dedicated model for it. An alternative to detailed behavioral models are predictive models, as used in the SPORE approach proposed by [15], where code is instrumented to measure the impact of features from the input stream on the execution times of parts of the application, resulting in a simple expression to predict execution times. This approach considers predictions for a single platform using an elaborate tool flow for automated instrumentation and feature selection. In [20], a performance modeling approach using probability distributions for the execution times of an application together with a platform is presented. The modeled CPU is restricted to not contain caches or predictive instructions, which are typically present in commercially available CPUs and beneficial for the execution time of stream-processing applications. Furthermore, extensive overviews of software performance-prediction approaches are provided by both [3] and [27], including approaches based on queuing networks, stochastic process algebra, stochastic Petrinets, and stochastic processes. However, the discussed approaches do not address models of the underlying platform or CPUs that support predictive instructions and mostly contain detailed application models that may be time consuming to create.

Our approach performs regression for applications based on the Master Slave pattern [4] executed by a reference embedded platform, followed by a DSE according to the Y-chart approach [11] to identify cost-effective alternatives. The Y-chart approach describes multiple iterations of evaluating a mapping of an application to a target platform, where based on evaluation result the mapping, application, or platform is optimized. To predict the throughput when using multiple cores in a CPU, regression is performed using an extension of Amdahl's law [2] that considers the discrete partitioning and distribution of objects from the input stream. Other approaches predicted throughputs using a support vector machine, a neural network, or machine learning, as presented in [1] and [10]. However, in [1] the focus is on the time used in a production system rather than different embedded platforms and in [10] database queries for a fixed platform are considered. Extensions of Amdahl's law are proposed for multi-core and cloud computing systems [28, 23, 8, 14]. However, these do not consider the discrete scaling of workload over slave processes and cores. An approach applying DSE for heterogeneous system performance is presented in [18], searching mappings of tasks from an application among cores in CPUs, considering latency and energy for which back-of-the-envelope predictive models are used. Where this approach considers mappings for modeled platforms, our approach performs DSE for a large set of CPUs using information obtained from a performance database. Our approach targets the early design phase using gray-box knowledge of applications, such that an expression for throughput prediction is easily obtained and updated during the development of the application.

3 Overview of Throughput-Prediction Approach

This section presents an overview of the piecewise linear expression for the throughput prediction, which is obtained in three steps: a single-core, a multi-core, and a cross-platform prediction step. Also, a description is given of the industrial stream-processing applications that are targeted by this approach.

The proposed throughput-prediction approach predicts the throughput for Master Slave based stream-processing applications, for different input streams and embedded platforms. A piecewise linear expression is employed because of its simplicity and adaptability during the early design phase. The approach uses a piecewise linear expression obtained by the following three steps:

1. Single-core execution time prediction: translate the features of the input stream into a single-core execution time. For the running example, translate the (input) compressed bitmap size to an execution time on a single processing core. To derive this expression, execution times of the test-set processed by the application at a single processing core are used.
2. Multi-core throughput prediction: translate the single-core execution time to a multi-core execution time, from which the throughput can be derived. For the digital image processing application, this involves considering the distribution of data over the slaves. To derive this expression, execution times of the digital image processing application for the test-set while it is executed at one, two, and three processing cores are used.
3. Cross-platform translations: translate the performance to other platforms, using available performance scores. For the digital image processing application the scores from the PassMark [21] performance database are used.

Execution times can be obtained, by using an early version of an application mapped to a reference platform. The preceding three steps will be detailed in sections 4, 5, and 6.

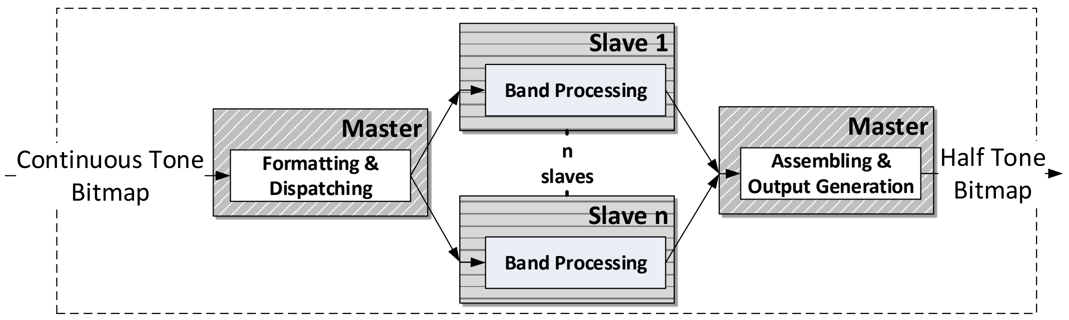


Figure 1 Digital image processing application structure, applying the Master Slave pattern.

The throughput-prediction approach targets industrial stream-processing applications that apply the Master Slave pattern [4]. Additionally, the MapReduce pattern [7] introduced by Google has a structure similar to the Master Slave pattern, such that the predictions may even be applicable for computing clusters. Typically, stream-processing applications have a tight throughput requirement, which means that the output has to be available at a fixed rate for an actuator. A small buffer is often used for the output of the stream-processing application, such that the throughput requirement can be averaged over the number of outputs in this buffer. The input stream consists of a stream of objects that the application should process, where the objects can be sub-divided into parts that can be distributed among the slaves for processing.

We use a prototype of an Océ digital image processing application for a printer as running example. The application has a tight throughput requirement, because converted bitmaps have to be available at a fixed rate to actuate the print head at the moment the paper passes it. A buffer for a few bitmaps can be used towards the print head, because buffering too many bitmaps hampers the ability to correct the print head for detected failures. This puts a firm real-time requirement on the system [5, 25], where it is undesirable that a deadline is missed but the system can continue afterward. Not meeting the deadline a few times in a row results in an empty buffer towards the print head, such that one or possibly multiple pages get lost in the case of duplex printing.

An overview of the digital image processing application is given in Figure 1. The input stream consists of compressed Continuous Tone (CT) bitmaps that are partitioned into 8 bands each for this example, where each band covers a fixed number of lines from a bitmap. The *master* has two function blocks, annotated with a white diagonal pattern. First, it receives a CT input bitmap from which the bands are dispatched to slaves. Next, it assembles the results of the slaves to return a Half Toned (HT) output bitmap. Note that a slave may have to process multiple bands. Internally slaves apply the Pipes and Filter pattern to process the bands, in such a way that typically the available cache memory is sufficient, thereby avoiding interference due to memory access between slaves. At initialization time, the number of slaves, which each will be assigned to a core or hyper-thread of a core, can be configured. For a processor that supports hyper-threading, typically a *physical core* can be seen as two *logical cores*, where the logical cores share the execution resources of the physical core. For the performance required by the digital image processing application, x86 instruction set based CPUs are considered that typically support hyper-threading, predictive instructions, and caching. To benchmark the application, a test-set with 455 bitmaps is used that represents the typical load of the prototype Océ digital image processing application, because the test-set includes simple, typical, and complex bitmaps.

4 Single-Core Execution Time Predictions

A linear expression is used to predict the single-core execution time for stream-processing applications. Typical features of the input stream, like size or resolution, relate to required processing time. The relation between one or more features from an input stream and the single-core execution time is captured using regression.

Gray-box knowledge of the stream-processing application can be used to identify features from the input stream that determine, or have a strong influence on, the execution time. Using a linear expression to relate these features and the execution time results in a simple and easy adaptable model, thereby making it especially suitable during the early design phase when many design decision still have to be made. For a good relation between the identified features and the execution time, an application designed for predictable temporal behavior is desirable, thus it should avoid large and abrupt changes in execution time for minor feature changes.

For the digital image processing application, the compressed size of a bitmap and the size of the bands in this compressed bitmap influence the execution time, as illustrated in Figure 2. The execution times to process bands or compressed bitmaps have been measured on a reference Intel Core i7 platform, where execution times were determined by timestamps at the beginning and end of the processing. Both a) and b) plot the compressed size of the CT bitmap versus the execution time of the slave to process all bands of the 455 bitmaps from the test set using a single core or hyper-thread of a core of the embedded platform. In a) and c) a single core is used, and in b) and d) a single hyper-thread of a core is used while the other hyper-thread is performing computations for another slave process. In Figure 2 c) and d) the compressed size of a band is plotted versus the execution time of the slave to process this band, where all bands of the 455 bitmaps have been processed. Note that Figure 2 c) and d) have significantly more measurements than a) and b), which is because c) and d) plot the relation for each of the 8 bands of the 455 bitmaps.

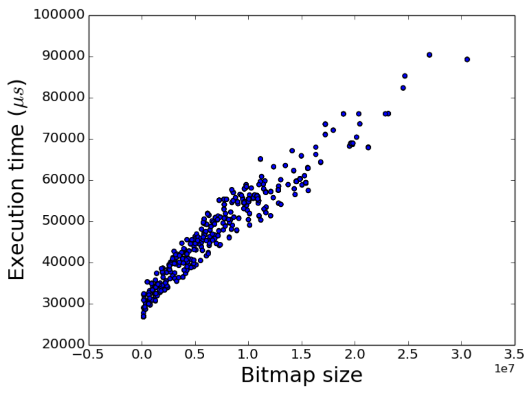
Figure 2 shows a relation between the execution time of the slaves of the digital image processing application and the compressed size of the bitmaps or bands, because the execution time increases when the bitmap or band size increases. The figure suggests a linear relation between bitmap or band size and image processing time, because most points are located around a line. This linear relation can be captured by Equation (1), with a coefficient c_0 that is multiplied with the size of the band or bitmap b to which a constant c_1 is added to obtain the execution time for a single core $T_s(b)$:

$$T_s(b) = c_0 b + c_1 \tag{1}$$

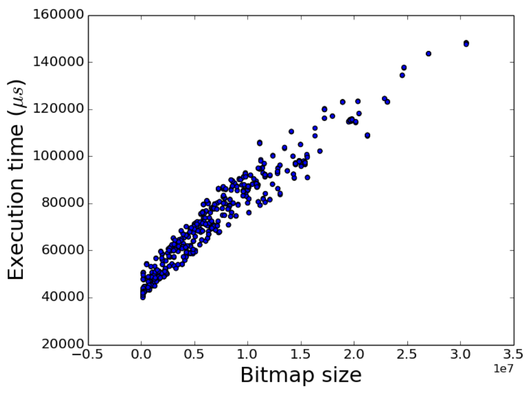
Note that the equation relates one feature, b in this case, to the execution time, but more features can be added if they are present and identified in the input stream.

Linear regression can be applied [19] using a tool box like StatsModels [22] to derive values for c_0 and c_1. Using the so-called ordinary least squares for the errors of the linear regression [19, 6], minimizes the sum of the squares of the differences between the measured execution times and those predicted by $T_s(b)$ via Equation (1). In Table 1, the coefficients c_0 and constants c_1 found by applying regression for Equation (1) for the four considered cases are given.

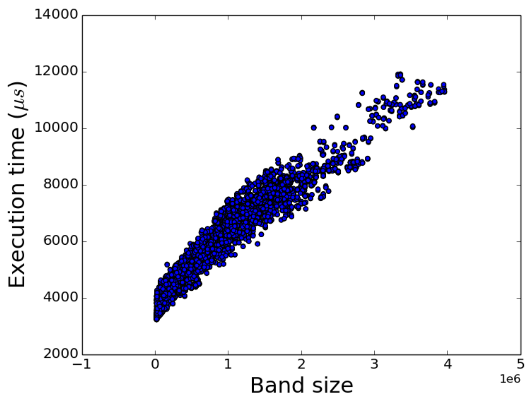
Additionally, Table 1 provides the R^2 and $P(F)$ values for the quality of the regression. The coefficient of determination, the R^2 value, gives the explained variation divided over the total variation for the regression. For the hyper-threading bitmap size regression, the R^2 value of 0.94 indicates that 6 percent of the variation is unaccounted for. The four cases for



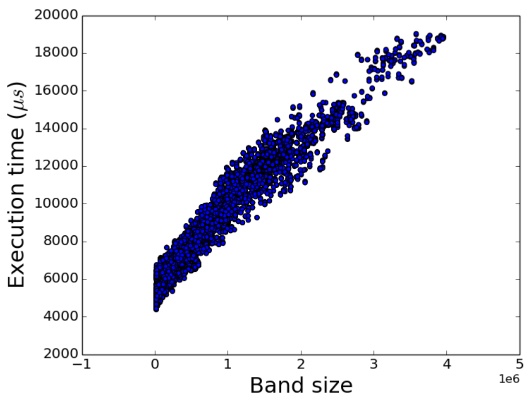
a) Single core bitmap processing.



b) Hyper-threaded bitmap processing.



c) Single core band processing.



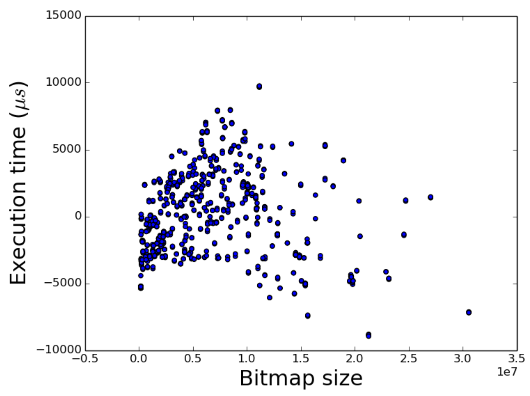
d) Hyper-threaded band processing.

Figure 2 Execution times of digital image processing application (μs) given the compressed bitmap size or compressed band size, for a processor using a single core or a core with a single hyper-thread.

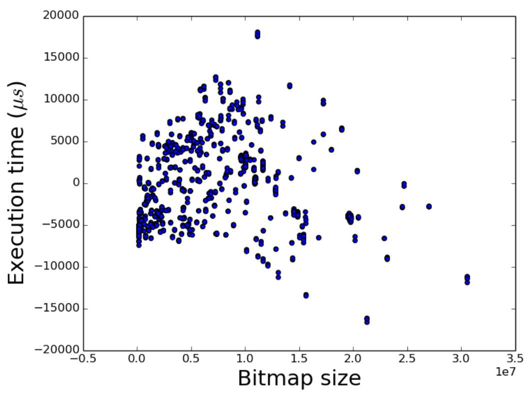
Table 1 Regression results for execution time using a single or hyper-threaded core with bitmap or band size.

	Single core bitmap processing	Hyper-threaded bitmap processing	Single core band processing	Hyper-threaded band processing
c_0	0.0021	0.0037	0.0022	0.0039
c_1	31,730	46,740	3883	5680
R^2	0.93	0.94	0.93	0.94
$P(F)$	0.0	0.0	0.0	0.0

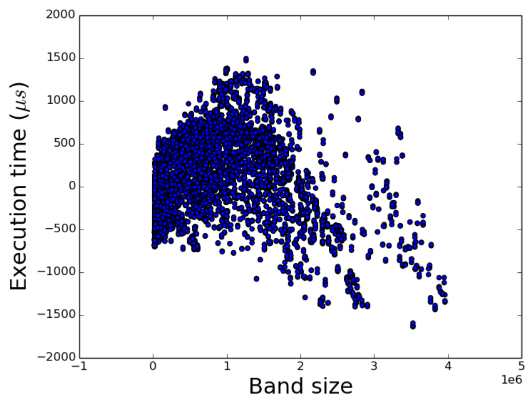
which regression is performed leave only 6 or 7 percent of the variation in the execution time unaccounted for, which is acceptable. The $P(F)$ values indicate whether the regression has significant predictive capability; the $P(F)$ values should be smaller than the significance level of 0.05. For all four regressions the $P(F)$ values are 0.0, indicating that the regression is



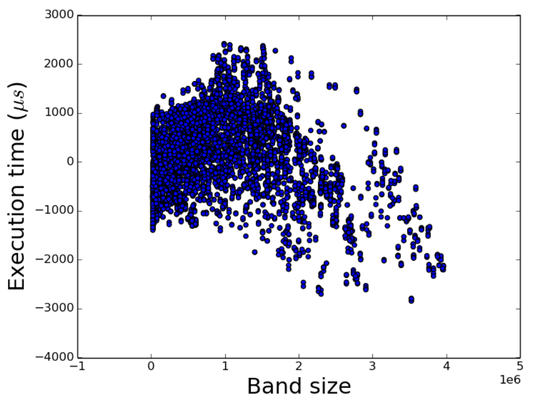
a) Single core bitmap processing.



b) Hyper-threaded bitmap processing.



c) Single core band processing.



d) Hyper-threaded band processing.

Figure 3 Residuals for linear relation from Equation (1) using variables from Table 1.

significant. All coefficients c_0 and constants c_1 also had a P-value of 0.0, indicating that it is very unlikely that they deviate significantly from the obtained values. Finally, residual plots of the differences between the measured and predicted execution times are given in Figure 3. These plots show no relation between the band or bitmap size and the difference between the measured and predicted execution times, indicating that it is captured by the regression.

5 Multi-Core Throughput Predictions

A piecewise linear expression for the speedup of the single-core execution time, when using multiple slaves that each process a part of the input stream, is discussed in this section. An expression based on Amdahl's law [2] is used that accounts for the workload of the input stream which is typically partitioned into a number of discrete parts. The multi-core throughput prediction is performed using the single-core execution times based on the compressed bitmap and band size, where using the compressed bitmap size results in the best predictions. The multi-core throughput prediction enables the different mappings of an application and platform to be evaluated, as suggested by the Y-chart approach [11].

An expression that expresses the speedup by using multiple cores or hyper-threaded cores to execute the slaves in the embedded platform, should express the scaling of the single-core execution time. Amdahl [2] gave an expression, where the execution time scales continuously with the number of additional cores that can be used. Amdahl's law assumes that a workload can be continuously distributed over the number of parallel resources, slaves when the Master Slave pattern is applied, which is given in the following equation:

$$T_a(s,b) = (1-p)T_s(b) + p(\frac{1}{s})T_s(b) \tag{2}$$

where $T_s(b)$ is the execution time when using a single core or hyper-thread for a workload b, p is the fraction of the execution time that benefits from additional parallel resources, and s is the number of parallel cores or hyper-threaded cores. Our observation is that typically an input stream consists of objects that have a discrete partitioning in a number of elements that each require processing. When considering continuous scaling to process a bitmap with 8 bands, going from 4 to 5, 6, or 7 slaves would speedup the throughput. However, due to the discrete partitioning of a bitmap in 8 bands no speedup is realized, because at least one of the slaves has to process two parts and thereby determines the throughput. Therefore, an extension of Amdahl's law is proposed that considers the discrete scaling of the workload. The discrete scaling expression for Amdahl's law $T_d(d,s,b)$ that returns an execution time, considers that a workload b has d discrete parts that are to be distributed over s parallel resources and is expressed as follows:

$$T_d(d,s,b) = (1-p)T_s(b) + p(\frac{\lceil \frac{d}{s} \rceil}{d})T_s(b) \tag{3}$$

where the ceiling $\lceil \frac{d}{s} \rceil$ determines the largest integer number of the discrete parts in the input stream that a slave has to process and this is divided by d to determine the speed up compared to the single-core execution time $T_s(b)$.

By adding coefficients and a constant, regression can be performed using the discrete scaling expression of Amdahl's law to express the execution time dependent on the number of used cores. For the digital image processing application, regression is performed using the following expression:

$$T_d(d,s,b) = T_s(b)c_2 + (\frac{\lceil \frac{d}{s} \rceil}{d})T_s(b)c_3 + c_4 \tag{4}$$

where $T_s(b)$ gives the single-core execution time for one bitmap from the input stream of the digital image processing application which has d discrete parts, which are processed by s slave processes that each run on their own core or hyper-thread of a core. Each part of the expression has a coefficient, c_2 and c_3, and a constant c_4 is added, which are determined by performing regression. Note that compared to Equation (3), Equation (4) does not contain p to represent the sequential fraction. The variable p is left out because regression captures it in the coefficients c_2, c_3, and c_4.

Regression for the digital image processing application is performed using Equation (4), where for the single-core execution time prediction $T_s(b)$ the regression results from Table 1 are used. The single-core execution time prediction for the bitmap size returns the total execution time for all slaves and can directly be used for $T_s(b)$. The single-core execution time prediction for the band size returns the execution time of the slave for the specific band, but the execution time for all bands of a bitmap is required. The average size of the n bands of a bitmap can be used to compute the single-core execution time, by multiplying the execution time for the average band size by n. However, the predicted execution times for the average band sizes are nearly similar to the predicted execution times for the bitmap sizes. Alternatively, the maximum band size can be used by taking the size of the largest band of a bitmap, computing its execution time, and multiply it by n, to get a pessimistic estimation of the execution time. Because execution times predicted for maximum band sizes differ from execution times predicted for bitmaps sizes, below multi-core throughput predictions based on both single-core execution time predictions will be compared.

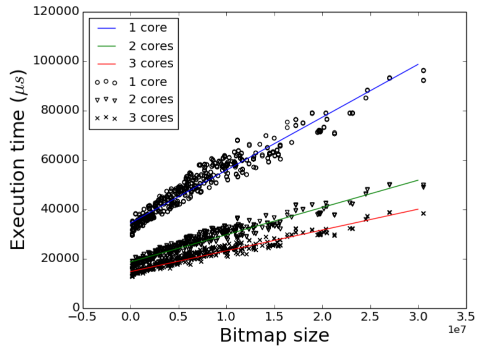
Table 2 Regression results for the execution time at one or more single or hyper-threaded cores, using the bitmap or band size to determine $T_s(b)$.

	Single core bitmap processing	Hyper-threading bitmap processing	Single core band processing	Hyper-threading band processing
c_2	0.0300	0.0151	-0.0369	-0.0146
c_3	0.9903	0.9661	0.8500	0.8131
c_4	1971.4	2795.0	5641.8	5420.5
R^2	0.98	0.95	0.95	0.91
$P(F)$	0.0	0.0	0.0	0.0

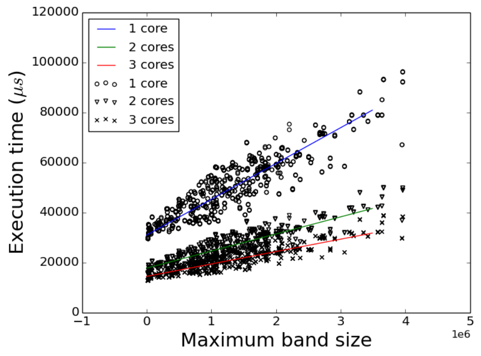
Table 2 gives the regression results for Equation (4) when using the single or hyper-threaded core execution time predictions from Table 1. Regression was performed using measured execution times for the test-set when using 1, 2, or 3 single or hyper-threaded cores in the platform. The R^2 indicates that between 2 and 9 percent of the variation is unaccounted for, which indicates a good result. Note that when using the band size for Equation (4), the obtained value for c_2 is negative. Together, the value of c_2 and the constant c_4 relate to the execution time of the master, where the rather small negative c_2 indicates that with increasing band sizes the execution time for the master decreases slightly. Furthermore, the $P(f)$ value being smaller than 0.05 indicates significant predictive capability of the regression. Also for this regression, each coefficient and constant had a P-value of 0.0, indicating that it is unlikely that the found values deviate significantly from the actual values.

Figure 4 shows measured execution times given the compressed bitmap size or the maximum band size on single or hyper-threaded cores, lines for the relations obtained for Equation (4), and corresponding residual plots. In a), b), e), and f) the execution times when using a single core or a hyper-threaded core are plotted versus the compressed bitmap size or the maximum band size. Additionally, for 1, 2, and 3 cores the line resulting from Equation (4) is plotted, where in e) and f) only two lines are visible because the lines for 4 and 6 hyper-threads overlap. The residual plots in c), d), g), and h) show that there is no remaining relation between the compressed bitmap size or maximum band size and the differences between the predicted and measured execution times. These plots show that the line of the piecewise linear equation relates to the measurements and that there is no remaining relation between the execution time and the compressed bitmap size.

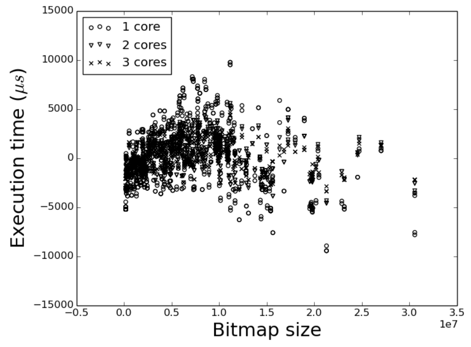
We choose to use the bitmap size to predict the single-core execution time, because the regressions in Table 1 and 2 that use the bitmap size show a slightly better R^2, compared to when the maximum band size is used. When using three cores with hyper-threading for the digital image processing application for a bitmap with a size of 7,255,824, which has 1,316,317 as corresponding maximum band size, an execution time of 20.23 ms is measured, where the bitmap size based expression predicts 21.68 ms and the band size based expression 21.74 ms. The slightly better R^2 indicates that the regression results using the bitmap size cover the variance in the execution times a little bit better compared to the regression using the maximum band size. Additionally, obtaining the size of the bitmap is more convenient compared to identifying the maximum size among the bands in a bitmap.



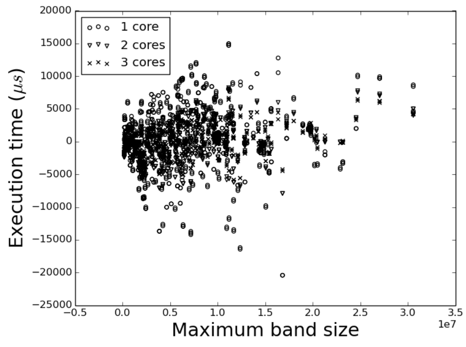
a) Single core bitmap processing.



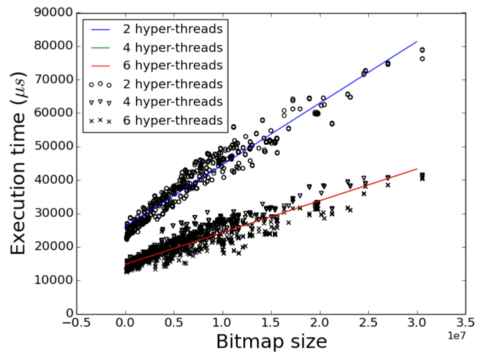
b) Single core band processing.



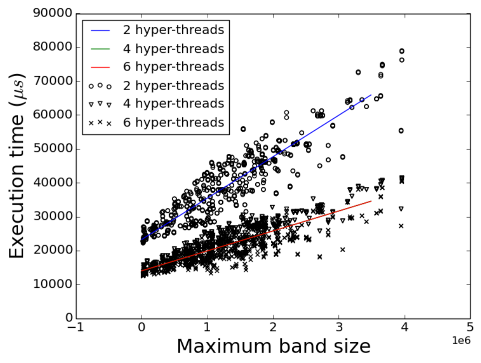
c) Residuals single core bitmap processing.



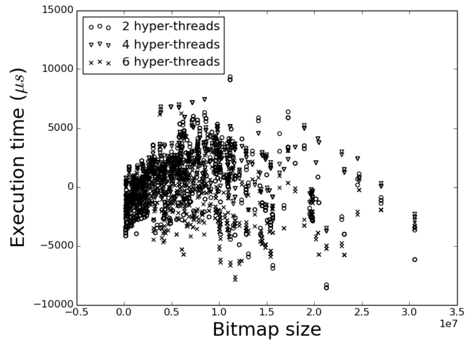
d) Residuals single core band processing.



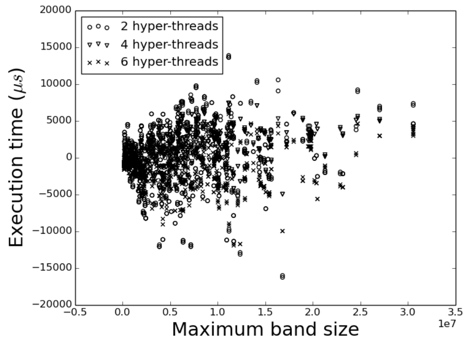
e) Hyper-threaded bitmap processing.



f) Hyper-threaded band processing.



g) Residuals hyper-threaded bitmap processing.



h) Residuals hyper-threaded band processing.

Figure 4 Execution time for the digital image processing application using multiple cores given the compressed bitmap or band size, including lines for Equation (4), and residual plots.

The discrete scaling Amdahl's law expression $T_d(d, s, b)$ returns an execution time in μs, where for the digital image processing application a throughput value is desired. The worst-case or average throughput can be determined by taking the reciprocal of the execution time for a large or average bitmap which is multiplied by $1 \cdot 10^6$, resulting in the Bitmaps Per Second (BPS) that can be processed by the digital image processing application. A buffer of n output bitmaps is used at the output of the digital image processing application to average the variations in execution times. Because the execution time is linearly related to the CT bitmap size, the average size of the n CT bitmaps that can fill the buffer determines the throughput. The average bitmap size of the test-set, being 7,255,824, is used as average size for these n bitmaps that determine the throughput. Note that typically, one would choose a bitmap size slightly larger than the average size that fits in the buffer, to include a margin on the throughput.

6 Cross-Platform Throughput Predictions

Cross-platform throughput prediction can be achieved by combining the multi-core throughput prediction with performance numbers of a reference and a target platform. During the early design phase, such performance numbers allow predicting the potential throughput of an application on considered platforms, before investing the effort of porting the application to the platform.

To translate the throughput from a reference to a target platform, performance numbers of both platforms are required to determine a ratio that represents the relative throughput increase or decrease. It is preferable to base the performance number on a benchmark that performs similar operations, also in similar proportions, as the considered stream-processing application. Accurate performance numbers can be realized by creating a representative benchmark to obtain performance numbers for both platforms. Creating such a benchmark is costly. Furthermore, only physically available platforms can be used for such benchmarks, which can limit the number of platforms that can be considered. An alternative is to use a database with performance numbers, like FutureMark [9] or PassMark [21], which provide performance numbers for server and desktop CPUs and mobile platforms. Note that a drawback of such a database is that the performance numbers may be less accurate in representing the performance of the considered application.

We use the PassMark CPU performance database [21], because it contains thousands of x86 instruction set based platforms, thereby enabling DSE for all these candidate platforms. For a broad range of CPUs, PassMark provides a *Full CPU score* that rates the overall performance of the CPU and a *Single-Threaded CPU score* that rates the performance of a single core or a core with hyper-threading. Users run benchmarks on their CPUs and submit their scores to the PassMark database, such that the score for common CPUs is typically averaged over thousands of benchmark results. The Full CPU score is based on a benchmark with nine tests, where the Single-Threaded CPU score is based on three tests from this benchmark. Both tests use weighting factors for the contribution of test results to the score, where compression, floating point math, and string sorting tests have a high impact. These tests are representative for the type of operations performed by the digital image processing application. Note that the PassMark database for mobile platforms might be interesting, however using these scores requires measured execution times for the digital image processing application at a mobile platform from this database, which we currently do not have.

The Full and Single-Threaded CPU scores can be used for the relative performance of platforms, such that a factor can be derived to translate the throughput to a target platform. Using the available multi-core throughput regression results from a number of platforms available in the PassMark database, we derive factors using the PassMark scores. Note that with multi-core regression results for a sufficiently large number of CPUs in the database, regression could be applied for an even better relation between the Passmark scores and the multi-core throughput regression results.

The factor to perform a cross-platform translation for the single-core execution time (C_s), between a reference and target platform is given by the following expression:

$$C_s(p_s^r, p_s^t) = \frac{p_f^r}{s_t^r} \bigg/ \frac{p_f^t}{s_t^t} \tag{5}$$

using $p_s^r = (s_t^r, p_f^r)$ and $p_s^t = (s_t^t, p_f^t)$ to keep the parameter list concise, where p_f^r and p_f^t represent the Full CPU scores and s_t^r and s_t^t the number of cores, for the reference and the target platform, respectively. Note that this equation uses the Full CPU scores rather than the Single-Threaded CPU scores, since attempts using the Full CPU scores resulted in a more accurate prediction, probably because the tests for the PassMark Full CPU score have a better match with the digital image processing application.

The factor (C_m) to perform a cross-platform translation for the throughput, considering the number of used cores, between a reference and target platform is given by the following expression:

$$C_m(p_m^r, p_m^t) = \frac{\left(\frac{p_f^r}{s_t^r}\right)}{p_s^r} \Bigg/ \frac{\left(\frac{p_f^t}{s_t^t}\right)}{p_s^t} \tag{6}$$

using $p_m^r = (s_t^r, p_f^r, p_s^r)$ and $p_m^t = (s_t^t, p_f^t, p_s^t)$ to keep the parameter list concise, where p_f^r and p_f^t represent the Full CPU scores, p_s^r and p_s^t the Single-Threaded CPU scores, and s_t^r and s_t^t the number of cores in the reference and target platform, respectively. For both the reference and target platform, this equation translates the Full CPU score to a score for a single core and divides it over the Single-Threaded CPU score, which results in a factor that indicates how much more multi-core performance is obtained compared to s_t times the Single-Threaded performance.

The factor C_s to translate the execution time from a reference to a target platform is included in the single-core execution time expressions from Equation (1) as follows:

$$T_{sc}(b, p_s^r, p_s^t) = (c_0 b + c_1)\, C_s(p_s^r, p_s^t) \tag{7}$$

where p_f^r and p_f^t represent the Full CPU scores and s_t^r and s_t^t the number of cores present, of the reference and target platform, respectively.

The factor C_m to consider the scaling of the multi-core performance and T_{sc} for the cross-platform single-core execution time is included in the multi-core throughput prediction from Equation (4) as follows:

$$T_{dc}(d, s, b, p_s^r, p_s^t, p_m^r, p_m^t) = T_{sc}(b)c_2 + \left(\frac{\lceil \frac{d}{s} \rceil}{d}\right) T_{sc}(b) c_3 C_m(p_m^r, p_m^t) + c_4 C_m(p_m^r, p_m^t) \tag{8}$$

where T_s is replaced by T_{sc} and both c_3 and c_4, which relate to the amount of used parallelism, are multiplied by C_m.

Table 3 CPU information of benchmark platforms.

CPU	Launch	Cores(s_t)	Speed (GHz)	p_f	p_s
Reference Core i7	2015	4/8		10,040	2,159
Core i7 4770	Q2'13	4/8	3.4	9,797	2,226
Core i7 860	Q3'09	4/8	2.8	5,060	1,226
Xeon E5 2650	Q1'12	8/16	2.0	10,262	1,310
Atom x5 Z8500	Q1'15	4/4	1.44	1,697	503
Atom C2758	Q3'13	8/8	2.4	3,162	512

7 Throughput Prediction Validation

The cross-platform throughput prediction is validated by comparing its predictions based on a reference platform with multi-core predictions for a limited set of platforms, by using execution times measured on these platforms. Differences between 13% and -8% are found, which is acceptable for a light-weight execution time prediction that guides decisions during the early design phase.

The CPUs of the platforms used for the validation are shown in Table 3, where the first row shows the reference platform with an Intel Core i7 from the Skylake generation. The names of the CPUs are given in the first column and the launch dates in the second column. These names and launch dates indicate that high-end and low-end CPUs with varying introduction years will be used for the validation. In the column *cores*, the number of cores and hyper-threads is given. The Full CPU score (p_f) and Single-Threaded score (p_s) from the PassMark database, sampled in May 2017, are given in the last two columns. The platforms contain a varying amount of memory with different speeds, however the slaves of the digital image processing application can typically prefetch the data in the cache, such that the memory has limited influence on the execution time.

To validate the accuracy of the cross-platform prediction, execution times have been measured for the digital image processing application for the platforms from Table 3 and regression has been performed for these measurements. For each of these platforms, for the different number of cores or hyper-threads that could be used, the execution times for the 455 bitmaps from the test-set were measured. Regression was performed for the measurements to obtain compact models from which results can be compared. Table 4, provides the coefficients c_0, c_1, c_2, c_3, and c_4 for the platforms from Table 3. The i7 and Xeon CPUs have two rows with coefficients, one for using them with single-cores (sc) and one for using their cores with hyper-threading (ht), as indicated in the second column. The last column gives the R^2 value of the regression for the multi-core expression from Equation (4), where the values above 0.92 indicate that good relations have been found.

Table 5 provides the relative differences between measured and predicted execution times for the platforms, for the case where four cores or four hyper-threads are used. Differences between results from the expressions are compared, rather than comparing the slopes and constants for the expression. Comparing slopes and constants between the expression for the multi-core and cross-platform throughput predictions showed to be impractical, because the performed regressions for the five coefficients has multiple solutions. The differences shown in Table 5 are given for three bitmap sizes, the average bitmap size in the test set, and a small and a large bitmap size. The large and small bitmap size are determined by subtracting and adding the standard deviation among the bitmap sizes in the test set, respectively. Considering the differences given in Table 5, for the average size bitmap the multi-core and

Table 4 Regression results for the digital image processing application executed at the platforms from Table 3.

		c_0	c_1	c_2	c_3	c_4	R^2
Reference	sc	0.0021	31,730	0.0300	0.9903	1971	0.98
Core i7	ht	0.0037	46,740	0.0151	0.9661	2795	0.95
Core	sc	0.0024	35,860	0.0147	1.0063	1919	0.97
i7 4770	ht	0.0040	56,160	0.0026	0.9995	3058	0.97
Core	sc	0.0047	60,380	0.1059	0.9170	2140	0.96
i7 860	ht	0.0075	105,400	0.0020	0.9972	4285	0.96
Xeon	sc	0.0041	66,310	0.0141	1.0075	3365	0.97
E5 2650	ht	0.0062	95,580	0.0419	0.9239	4542	0.97
Atom x5 Z8500	sc	0.0105	137,400	0.0035	1.0192	6085	0.97
Atom C2758	sc	0.0126	157,800	-0.0603	1.0868	5425	0.92

Table 5 Difference between measurement and prediction for the digital image processing application, using 4 cores.

	Bitmap size	small 1,450,148	average 7,255,824	large 13,061,500
Intel	sc	0.01	0.00	0.00
i7 4770	ht	-0.07	-0.05	-0.04
Intel	sc	-0.05	-0.08	-0.10
i7 860	ht	-0.02	0.01	0.04
Intel	sc	0.08	0.10	0.11
Xeon E5 2650	ht	0.03	0.09	0.13
Atom x5 z8500	sc	-0.06	-0.07	-0.07
Atom C2758	sc	0.10	0.08	0.06

cross-platform predictions differ between 10% and -8% and for all bitmaps between 13% and -8%. Similar numbers are obtained using a different number of cores and cores with hyper-threads. These differences do not seem to relate to platform generations nor cores with or without hyper-threads. Given that it is a light-weight model to guide decisions during the early design phase, we find this error acceptable.

8 Design Space Exploration using Throughput Predictions

Cross-platform throughput predictions enable exploration of suitable and cost-effective embedded platforms. First, cost-effective embedded platforms are compared, followed by an exploration of effective combinations of platforms.

Often costs and performance are Key Performance Indicators (KPI) for stream-processing applications. However, cost is a KPI that can be refined in platform purchase costs, development costs, and life-cycle costs. Note that quantifying development and life-cycle costs is difficult and that they are likely larger than the platform purchase cost. By quantifying the throughput of a stream-processing application for a large list of embedded platforms, the most cost-effective platform can be selected. However, among products, the development costs can be reduced by selecting a cost-effective low, medium, and high-performance embedded platform for which development and updates will be performed. For the life-cycle, it would be even nicer to be able to combine a number of these low, medium, and high-performance embedded platforms to be able to scale the throughput by adding an embedded platform. Note that this requires an adapted stream-processing application that distributes scaled bands to the slaves at the different platforms and balances the load, as described by the MapReduce pattern for computing cluster.

Combining the cross-platform throughput prediction with an extensive and detailed list of Intel CPUs [16], a motherboard list, a list with memory modules, and a PassMark score list, the design space can be explored for cost-effective embedded platforms and embedded platform combinations. Intel provides extensive lists of their CPUs with details like sockets, supported memory types, and the CPU name, which allows linking the CPU information with compatible motherboards, memory modules, and PassMark scores, respectively. In the performed exploration 44 motherboards were considered and 20 different DDR memory modules, with varying technologies and sizes, for which the information and costs were obtained via supplier information. The list of Intel contained 2504 CPUs from which 987 were selected as relevant and the PassMark list contained 2253 entries with 1424 relevant entries. Combining the Intel and PassMark list resulted in a list with detailed CPU information, with amongst others the CPU costs, for 754 CPUs. The combination of this list with detailed CPU information and the list with motherboards and memory modules, resulted in 1838 different combinations that each represent an embedded platform to be considered.

In the list with embedded platforms, for each platform the throughput was added for the digital image processing application, the costs of the platform, and cost per one BPS. For each embedded platform, the throughput in BPS was calculated, using Equation (7) and (8) and the PassMark score. To calculate the throughput, a large bitmap of size 13,061,500 is considered, similar to the large bitmap used in Table 5. Additionally, the number of cores or cores with hyper-threads is decreased by one or two, respectively, to allow room for the master and other processing.

Table 6 provides a low, medium and high-performance platform, obtained from the list of embedded platforms that may be cost-effective alternatives to the reference platforms to reduce development costs. To compare the costs, throughput in BPS, and costs per BPS of these platforms, values relative to the Intel i7 Skylake reference platform are provided. The first column indicates the performance category of the platform and the second column gives the name of the CPU. The relative increase or decrease in cost, predicted throughput, and costs per BPS, are given in the third, fourth, and fifth column, respectively. The final two columns give the PassMark scores, p_f and p_s, of the CPU. The table illustrates that the high-performance platform with an Intel i7 5820k CPU reduces the cost per BPS by a fraction of 0.375, where the low-performance platform costs 1.44 times as much, compared to the reference platform. Still, the low or medium-performance platforms are interesting as cost-effective solutions for products in which the predicted throughput would be sufficient, because their platform costs are a fraction compared to the high-performance platform costs. Note that for industrial systems, the platform selection also considers the period for which CPUs and components will be long-term available for production, which we did not include here.

Figure 5 plots the relative costs to the reference platform, when using one or multiple embedded platforms to realize a required throughput, suggesting that considering multiple embedded platforms results in smoother increasing platform costs. Note that size and power are not considered in this comparison, but that they are an important element in the trade-off for the platform selection. Using the list with embedded platforms, for a throughput of 220 until 280 BPS, the most cost-effective solutions are searched to realize the required

Table 6 CPUs for high, medium, and low-performance embedded platforms.

Performance category	CPU name	Relative costs	Relative BPS	Relative costs/BPS	p_f	p_s
High	i7 5820k	1.29	3.35	0.375	12,994	2,016
Medium	i7 6700te	1.00	1.10	0.875	10,514	2,113
Low	i5 6402p	0.80	0.55	1.44	7,750	2,081

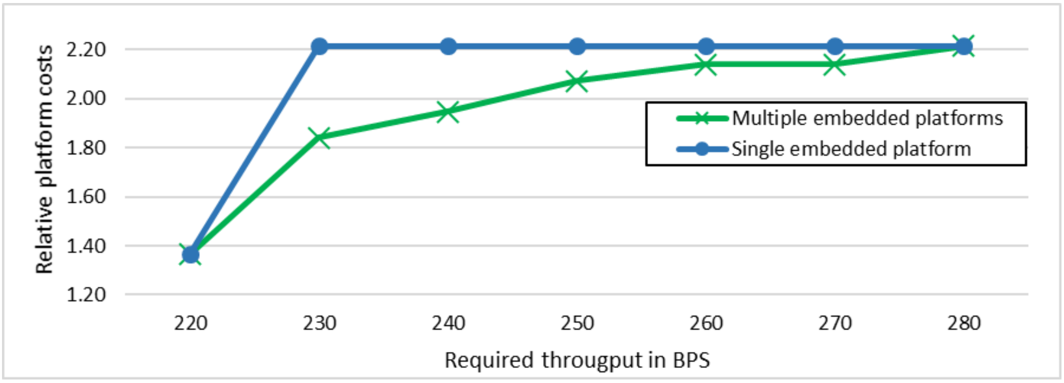


Figure 5 Cost comparison for platform combinations and single platforms to realize a desired BPS.

throughput, either by only using one embedded platform or by allowing multiple embedded platforms to be used. Multiple platforms can be used that each process a part of a bitmap, where partitioning the bitmap requires no additional effort and the effort for combining the parts is negligible. In case multiple embedded platforms are considered, an exhaustive search is performed that considers all platform combinations to realize the required throughput. The speed of communicating bitmaps plays a role, if we compare solutions with one or multiple embedded platforms. Therefore, network interfaces, with their speed and costs, were added to the platforms and the search accounts for the achievable BPS via the network interface. It is sufficient to consider the number of BPS that can be communicated via a network interface, because communicating a next bitmap can overlap with processing the current. Performing the searches requires less than 30 seconds on a single core, making it a scalable approach. An interesting result is that for a throughput of 230 until 270 BPS, the usage of multiple embedded platforms is more cost-effective and also results in a smoother increasing platform cost.

Table 7 provides details for the selected platforms, which are plotted in Figure 5, with costs relative to the costs of the Intel Core i7 reference platform. In the case of multiple embedded platforms, cost-effective Core i7 and i5 CPUs can be combined until a throughput of 270 BPS. In contrast, for a throughput of 230 BPS, a single embedded platform with two powerful Xeon CPUs at one motherboard is selected. For a throughput between 230 and 270 BPS the platform cost for multiple embedded platforms is up to 17% lower compared to a single platform. The cost difference is because cost-effective Core i7 and i5 CPUs can be combined, rather than using two the same Xeon CPUs on a motherboard.

Table 7 Comparison between multiple and single embedded platforms selected to realize a desired number of BPS.

	Multiple embedded platforms			Single embedded platform		
BPS	CPUs	BPS	Costs	CPU	BPS	Costs
220	Core i7 6800K	220	1.37	Core i7 6800K	220	1.37
230	Core i7 5820K Atom x7 Z8700 Atom x5 Z8550	230	1.84	Xeon E5 2620 Xeon E5 2620	349	2.22
240	Core i7 6800K Core i3 4170	241	1.95	Xeon E5 2620 Xeon E5 2620	349	2.22
250	Core i7 6800K Core i5 4590	251	2.07	Xeon E5 2620 Xeon E5 2620	349	2.22
260	Core i7 5820K Xeon E3 1231 v3	272	2.14	Xeon E5 2620 Xeon E5 2620	349	2.22
270	Core i7 5820K Xeon E3 1231 v3	272	2.14	Xeon E5 2620 Xeon E5 2620	349	2.22
280	Xeon E5 2620 Xeon E5 2620	349	2.22	Xeon E5 2620 Xeon E5 2620	349	2.22

9 Conclusion

A throughput-prediction approach for stream-processing applications and their embedded platforms has been presented in this paper. A real prototype industrial Océ digital image processing application for a printer, for which this approach was applied, has been used to demonstrate the approach. For this application, a design space exploration was performed, where the piecewise linear expression for the throughput prediction made it possible to consider combinations of more than 1800 different embedded platforms with the digital image processing application to realize desired throughputs.

The throughput prediction targets stream-processing applications that apply the Master Slave or possibly the MapReduce pattern, during the early design phase. A piecewise linear expression is related to features in the input stream, such that only gray-box knowledge of the application is necessary and updating the expression is easy. First, the execution time when using a single core is related to one or more features from the input stream. The second step scales the single-core execution time for the multiple cores or cores with hyper-threading that are used in the embedded platform. This step uses an expression for the discrete scaling of workload over slaves, which is a novel extension of Amdahl's law. The third step uses performance scores from a performance database to translate the performance from a reference platform to target platforms. To demonstrate the applicability of the approach, it was applied to an Océ digital image processing application for a printer. The obtained piecewise linear expression allowed throughput predictions during the early design phase and exploring a large set of embedded platforms. Validation of the cross-platform throughput prediction, using the digital image processing application and a limited set of platforms, showed an acceptable error and thereby its usability. An interesting extension of this throughput-prediction approach would be the inclusion of key configuration parameters of the application, like the output bitmap resolution, to enable predictions for next generations of the system.

References

1 Abdulrahman Alenezi, Scott A. Moses, and Theodore B. Trafalis. Real-time prediction of order flowtimes using support vector regression. *Elsevier Computers and Operations Research*, 35(11):3489–3503, 2008.

2 Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. Spring Joint Computer Conference*, pages 483–485. ACM, apr 1967.

3 Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

4 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

5 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

6 Abraham Charnes, Edward L. Frome, and Po-Lung Yu. The equivalence of generalized least squares and maximum likelihood estimates in the exponential family. *journal of the American Statistical Association*, 71(353):169–171, 1976.

7 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commununications of the ACM*, 51(1):107–113, 2008.

8 Fernando Diaz-del Rio, Javier Salmeron-Garcia, and Jose Luis Sevillano. Extending amdahl's law for the cloud computing era. *IEEE Computer*, 49(2):14–22, feb 2016.

9 Futuremark Corporation© 2016. Best CPUs - July 2017, 2017. https://www.futuremark.com/hardware/cpu/.

10 Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. PQR: Predicting query execution times for autonomous workload management. In *Proc. Int. Conf. on Autonomic Computing (ICAC)*, ICAC '08, pages 13–22, Washington, DC, USA, 2008. IEEE Computer Society.

11 Wolfgang Haid, Matthias Keller, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Generation and calibration of compositional performance analysis models for multi-processor systems. In *Proc. Int. symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*, pages 92–99, jul 2009.

12 Martijn Hendriks, Twan Basten, Jacques Verriet, Marco Brassé, and Lou Somers. A blueprint for system-level performance modeling of software-intensive embedded systems. *Springer Software Tools for Technology Transfer*, 18(1):21–40, feb 2016.

13 Martijn Hendriks, Jacques Verriet, Twan Basten, Marco Brassé, Reinier Dankers, René Laan, Alexander Lint, Hristina Moneva, Lou Somers, and Marc Willekens. Performance engineering for industrial embedded data-processing systems. In *Proc. Int. Conf. Product-Focused Software Process Improvement (PROFES)*, pages 399–414, Cham, 2015. Springer International Publishing.

14 Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7), 2008.

15 Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Proc. Int. Conf. on Neural Information Processing Systems*, NIPS'10, pages 883–891, USA, 2010. Curran Associates Inc.

16 Intel Corporation©. Product specifications, 2016. https://ark.intel.com/.

17 Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees Vissers. A methodology to design programmable embedded systems. In *Proc. Embedded Processor Design Challenges*, pages 18–37. Springer, 2002.

18 Sumit Mohanty, Viktor K. Prasanna, Sandeep K. Neema, and James R. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *ACM SIGPLAN Notices*, 37(7):18–27, 2002.

19 Douglas C. Montgomery, Elizabeth A. Peck, and Geoffrey G. Vining. *Introduction to Linear Regression Analysis*. Wiley & Sons, New York, third edition, 2006.

20 Ayoub Nouri, Marius Bozga, Anca Molnos, Axel Legay, and Saddek Bensalem. Astrolabe: A rigorous approach for system-level performance modeling and analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(2):31:1–31:26, 2016.

21 PassMark© Software. CPU benchmarks, may 2017. https://www.cpubenchmark.net/.

22 Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *Proc. 9th Python in Science Conference*, 2010.

23 Xian He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Academic Press Parallel Distributed Computing*, 70(2):183–188, 2010.

24 David Trilla, Javier Jalle, Mikel Fernandez, Jaume Abella, and Francisco J. Cazorla. Improving early design stage timing modeling in multicore based real-time systems. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

25 Maarten H. Wiggers. *Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, 2009.

26 Reinhard Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, 2008.

27 Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *Proc. Future of Software Engineering (FOSE)*, pages 171–187. IEEE, 2007.

28 Leonid Yavits, Amir Morad, and Ran Ginosar. The effect of communication and synchronization on Amdahl's law in multicore systems. *Elsevier Parallel Computing*, 40(1):1–16, jan 2014.

Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms

Waqar Ali
University of Kansas
Lawrence, USA
wali@ku.edu

Heechul Yun
University of Kansas
Lawrence, USA
heechul.yun@ku.edu

Abstract

Integrated CPU-GPU architecture provides excellent acceleration capabilities for data parallel applications on embedded platforms while meeting the size, weight and power (SWaP) requirements. However, sharing of main memory between CPU applications and GPU kernels can severely affect the execution of GPU kernels and diminish the performance gain provided by GPU. For example, in the NVIDIA Jetson TX2 platform, an integrated CPU-GPU architecture, we observed that, in the worst case, the GPU kernels can suffer as much as 3X slowdown in the presence of co-running memory intensive CPU applications. In this paper, we propose a software mechanism, which we call BWLOCK++, to protect the performance of GPU kernels from co-scheduled memory intensive CPU applications.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability, Computer systems organization → Heterogeneous (hybrid) systems, Computer systems organization → Processors and memory architectures

Keywords and phrases GPU, memory bandwidth, resource contention, CPU throttling, fair scheduler

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.19

Supplement Material ECRTS Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.4.2.3`

Acknowledgements This research is partly supported by NSF CNS 1718880.

1 Introduction

Graphic Processing Units (GPUs) are increasingly important computing resources to accelerate a growing number of data parallel applications. In recent years, GPUs have become a key requirement for intelligent and timely processing of large amount of sensor data in many robotics applications, such as UAVs and autonomous cars. These intelligent robots are, however, resource constrained real-time embedded systems that not only require high computing performance but also must satisfy a variety of constraints such as size, weight, power consumption (SWaP) and cost. This makes integrated CPU-GPU architecture based computing platforms, which integrate CPU and GPU in a single chip (e.g., NVIDIA's Jetson [5] series), an appealing solution for such robotics applications because of their high performance and efficiency [15].

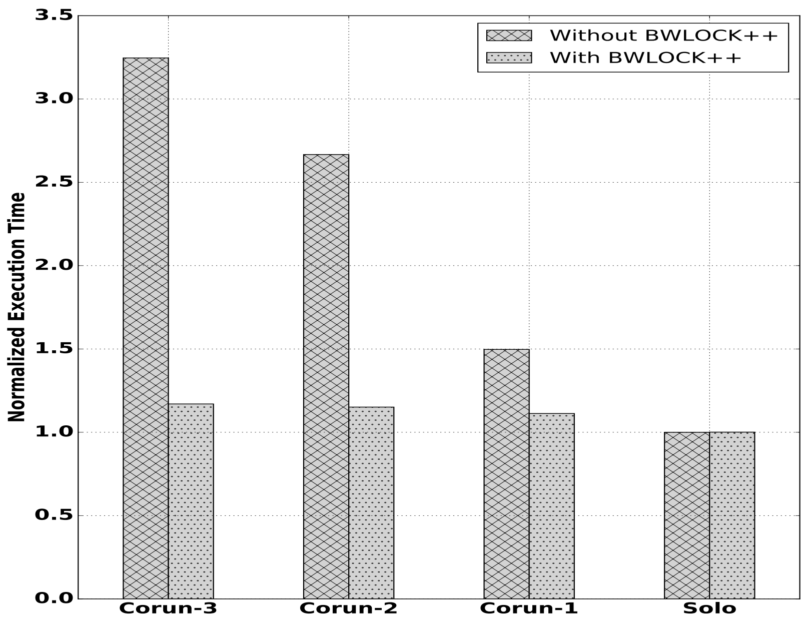


Figure 1 Performance of *histo* benchmark on NVIDIA Jetson TX2 with CPU corunners.

Designing critical real-time applications on integrated CPU-GPU architectures is, however, challenging because contention in the shared hardware resources (e.g., memory bandwidth) can significantly alter the applications' timing characteristics. On an integrated CPU-GPU platform, such as NVIDIA Jetson TX2, the CPU cores and the GPU typically share a single main memory subsystem. This allows memory intensive batch jobs running on the CPU cores to significantly interfere with the execution of critical real-time GPU tasks (e.g., vision based navigation and obstacle detection) running in parallel due to memory bandwidth contention.

To illustrate the significance of the problem stated above, we evaluate the effect of co-scheduling memory bandwidth intensive synthetic CPU benchmarks on the performance of a GPU benchmark *histo* from the parboil benchmark suite [18] on a NVIDIA Jetson TX2 platform (See Table 3 in Section 6 for the detailed time breakdown of *histo*.) We first run the benchmark alone and record the solo execution statistics. We then repeat the experiment with an increasing number of interfering memory intensive benchmarks on the idle CPU cores to observe their impact on the performance of the *histo* benchmark with and without the BWLOCK++ framework, which we propose in this paper. Figure 1 shows the results of this experiment. As can be seen in *'Without BWLOCK++'*, co-scheduling the memory intensive tasks on the idle CPU cores significantly increase the execution time of the GPU benchmark—a 3.3X increase—despite the fact that the benchmark has exclusive access to the GPU. The main cause of the problem is that, in the Jetson TX2 platform, both CPU and GPU share the main memory and its limited memory bandwidth becomes a bottleneck. As a result, even though the platform offers plenty of raw performance, no real-time execution guarantees can be provided if the system is left unmanaged. In *'With BWLOCK++'*, on the other hand, performance of the GPU benchmark remains close to its solo performance measured in isolation.

BWLOCK++ is a software framework designed to mitigate the memory bandwidth contention problem in integrated CPU-GPU architectures. More specifically, we focus on protecting real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks. BWLOCK++ dynamically instruments GPU tasks at run-time and inserts a *memory bandwidth lock* while critical GPU kernels are being executed on the GPU. When the bandwidth lock is being held by the GPU, the OS throttles the maximum memory bandwidth usage of the CPU cores to a certain threshold value to protect the GPU kernels. The threshold value is determined on a per GPU task basis and may vary depending on the GPU task's sensitivity to memory bandwidth contention. Throttling CPU cores inevitably negatively affects the CPU throughput. To minimize the throughput impact, we propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS). TFS favors CPU intensive tasks over memory intensive ones while the GPU is busy executing critical GPU tasks in order to minimize CPU throttling. Our evaluation shows that BWLOCK++ can provide good performance isolation for bandwidth intensive GPU tasks in the presence of memory intensive CPU tasks. Furthermore, the TFS scheduling algorithm reduces the CPU throughput loss by up to 75%. Finally, we show how BLWOCK++ can be incorporated in existing CPU focused real-time analysis frameworks to analyze schedulability of real-time tasksets, utilizing both CPU and GPU.

In this paper, we make the following contributions:

- We apply memory bandwidth throttling to the problem of protecting GPU accelerated real-time tasks from memory intensive CPU tasks on integrated CPU-GPU architecture
- We identify a negative feedback effect of memory bandwidth throttling when used with Linux's CFS [13] scheduler. We propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS), to mitigate the problem
- We introduce an automatic GPU kernel instrumentation method that eliminates the need of manual programmer intervention to protect GPU kernels
- We implement the proposed framework, which we call BWLOCK++, on a real platform, NVIDIA Jetson TX2, and present detailed evaluation results showing practical benefits of the framework [1]
- We show how the proposed framework can be integrated into the existing CPU focused real-time schedulability analysis framework

The remainder of this paper is organized as follows. We present necessary background and discuss related work in Section 2. In Section 3, we present our system model. Section 4 describes the design of our software framework BWLOCK++ and Section 5 presents implementation details. In Section 6, we describe our evaluation platform and present evaluation results using a set of GPU benchmarks. In Section 7, we present the analysis framework of BWLOCK++ based real-time systems. We discuss limitations of our approach in Section 8 and conclude in Section 9.

2 Background and Related Work

In this section, we provide necessary background and discuss related work.

GPU is an accelerator that executes some specific functions requested by a master CPU program. Requests to the GPU can be made by using GPU programming frameworks such as CUDA that offer standard APIs. A request to GPU is typically composed of the following four predictable steps:

- Copy data from host memory to device (GPU) memory
- Launch the function—called *kernel*—to be executed on the GPU
- Wait until the kernel finishes
- Copy the output from device memory to host memory

[1] The source code of BWLOCK++ is publicly available at: `https://github.com/wali-ku/BWLOCK-GPU`

In the real-time systems community, GPUs have been studied actively in recent years because of their potential benefits in accelerating demanding data-parallel real-time applications [11]. As observed in [1], GPU kernels typically demand high memory bandwidth to achieve high data parallelism and, if the memory bandwidth required by GPU kernels is not satisfied, it can result in significant performance reduction. For discrete GPUs, which have dedicated graphic memories, researchers have focused on addressing interference among the co-scheduled GPU tasks. Many real-time GPU resource management frameworks adopt scheduling based approaches, similar to real-time CPU scheduling, that provide priority or server based scheduling of GPU tasks [9, 10, 23]. Elliot et al., formulate the GPU resource management problem as a synchronization problem and propose the GPUSync framework that uses real-time locking protocols to deterministically handle GPU access requests [6]. Here, at any given time, one GPU kernel is allowed to utilize the GPU to eliminate the unpredictability caused by co-scheduled GPU kernels. In [12], instead of using a real-time locking protocol that suffers from busy-waiting at the CPU side, the authors propose a GPU server mechanism which centralizes access to the GPU and allows CPU suspension (thus eliminating the CPU busy-waiting). All the aforementioned frameworks primarily work for discrete GPUs, which have dedicated graphic memory, but they do not guarantee predictable GPU timing on integrated CPU-GPU based platforms because they do not consider the problem of the shared memory bandwidth contention between the CPU and the GPU.

Integrated GPU based platforms have recently gained much attention in the real-time systems community. In [15, 14], the authors investigate the suitability of NVIDIA's Tegra X1 platform for use in safety critical real-time systems. With careful reverse engineering, they have identified undisclosed scheduling policies that determine how concurrent GPU kernels are scheduled on the platform. In SiGAMMA [4], the authors present a novel mechanism to preempt the GPU kernel using a high-priority spinning GPU kernel to protect critical real-time CPU applications. Their work is orthogonal to ours as it solves the problem of protecting CPU tasks from GPU tasks while our work solves the problem of protecting GPU tasks from CPU tasks.

More recently, GPUGuard [7] provides a mechanism for deterministically arbitrating memory access requests between CPU cores and GPU in heterogeneous platforms containing integrated GPUs. They extend the PREM execution model [16], in which a (CPU) task is assumed to have distinct computation and memory phases, to model GPU tasks. GPUGuard provides deterministic memory access by ensuring that only a single PREM memory phase is in execution at any given time. Although GPUGuard can provide strong isolation guarantees, the drawback is that it may require significant restructuring of application source code to be compatible with the PREM model.

In this paper, we favor a less intrusive approach that requires minimal or no programmer intervention. Our approach is rooted on a kernel-level memory bandwidth throttling mechanism called MemGuard [22], which utilizes hardware performance counters of the CPU cores to limit memory bandwidth consumption of the individual cores for a fixed time interval on homogeneous multicore architectures. MemGuard enables a system designer—not individual application programmers—to partition memory bandwidth among the CPU cores. However, MemGuard suffers from system-level throughput reduction due to its coarse-grain bandwidth control (per-core-level control). In contrast, [21] is also based on a memory bandwidth throttling mechanism on homogeneous multicore architectures but it requires a certain degree of programmer intervention for fine-grain bandwidth control by exposing a simple lock-like API to applications. The API can enable/disable memory bandwidth control in a fine-grain manner within the application source code. However, this means that the application source code must be modified to leverage the feature.

Our work is based on memory bandwidth throttling, but, unlike prior throttling based approaches, focuses on the problem of protecting GPU accelerated real-time tasks on integrated CPU-GPU architectures and does not require any programmer intervention. Furthermore, we identify a previously unknown negative side-effect of memory bandwidth throttling when used with Linux's CFS scheduler, which we mitigate in this work. In the following, we start by defining the system model, followed by detailed design and implementation of the proposed system.

3 System Model

We assume an integrated CPU-GPU architecture based platform, which is composed of multiple CPU cores and a single GPU that share the same main memory subsystem. We consider independent periodic real-time tasks with implicit deadlines and best-effort tasks with no real-time constraints.

Task Model. Each task is composed of at least one CPU execution segment and zero or more GPU execution segments. We assume that *GPU execution is non-preemptible* and we do not allow concurrent execution of multiple GPU kernels from different tasks at the same time. Simultaneously co-scheduling multiple kernels is called GPU co-scheduling, which has been avoided in most prior real-time GPU management approaches [10, 6, 12] as well due to unpredictable timing. According to [15], preventing GPU co-scheduling does not necessarily hurt—if not improve—performance because concurrent GPU kernels from different tasks are executed in a time-multiplexed manner rather than being executed in parallel. [2]

Executing GPU kernels typically requires copying considerable amount of data between the CPU and the GPU. In particular, synchronous copy directly contributes to the task's execution time, while asynchronous copy can overlap with GPU kernel execution. Therefore, we model synchronous copy separately. Lastly, we assume that a task is single-threaded with respect to the CPU. Then, we can model a real-time task as follows:

$$\tau_i := (C_i, G_i^m, G_i^e, P_i)$$

where:

- C_i is the cumulative WCET of CPU-only execution
- G_i^m is the cumulative WCET of synchronous memory operations between CPU and GPU
- G_i^e is the cumulative WCET of GPU kernels
- P_i is the period

Note that the goal of BWLOCK++ is to reduce G_i^m and G_i^e under the presence of memory intensive best-effort tasks running in parallel.

CPU Scheduling. We assume a fixed-priority preemptive real-time scheduler is used for scheduling real-time tasks and a virtual run-time based fair sharing scheduler (e.g., Linux's Completely Fair Scheduler [13]) is used for best-effort tasks. For simplicity, we assume a single dedicated real-time core schedules all real-time tasks, while any core can schedule best-effort tasks. Because GPU kernels are executed serially on the GPU, as mentioned

[2] Another recent study [2] finds that GPU kernels can only be executed in parallel if they are submitted from a single address space. In this work, we assume that a task has its own address space, whose GPU kernels are thus time-multiplexed with other tasks' GPU kernels at the GPU-level.

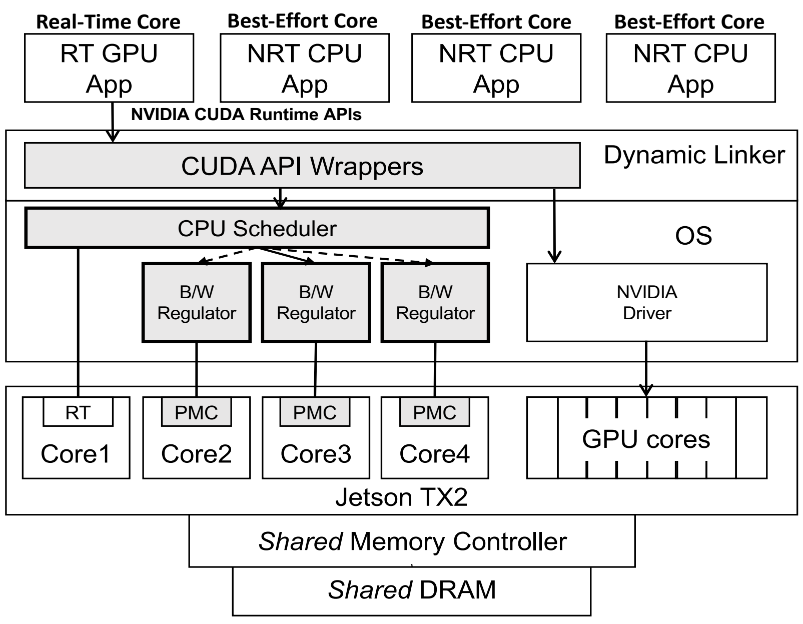


Figure 2 BWLOCK++ System Architecture.

above, for GPU intensive real-time tasks, which we focus on in this work, this assumption does not significantly under-utilize the system, especially when there are enough co-scheduled best-effort tasks, while it enables simpler analysis.

4 BWLOCK++

In this section, we provide an overview of BWLOCK++ and discuss its design details.

4.1 Overview

BWLOCK++ is a software framework to protect GPU applications on integrated CPU-GPU architecture based SoC platforms. We focus on the problem of the shared memory bandwidth contention between GPU kernels and CPU tasks in integrated CPU-GPU architectures. More specifically, we focus on protecting GPU execution intervals of real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks.

In BWLOCK++, we exploit the fact that each GPU kernel is executed via explicit programming interfaces from a corresponding host CPU program. In other words, we can precisely determine when the GPU kernel starts and finishes by instrumenting these functions.

To avoid memory bandwidth contention from the CPU, we notify the OS before a GPU application launches a GPU kernel and after the kernel completes with the help of a system call. Apart from acquiring the bandwidth lock on the task's behalf, this system call also implements the priority ceiling protocol [17] to prevent preemption of the GPU using task. While the bandwidth lock is being held by the GPU task, the OS regulates memory bandwidth consumption of the best-effort CPU cores to minimize bandwidth contention with the GPU kernel. Concretely, each best-effort core is periodically given a certain amount of memory bandwidth budget. If the core uses up its given budget for the specified period, the (non-RT) CPU tasks running on that core are throttled. In this way, the GPU kernel suffers minimal memory bandwidth interference from the best-effort CPU cores. However,

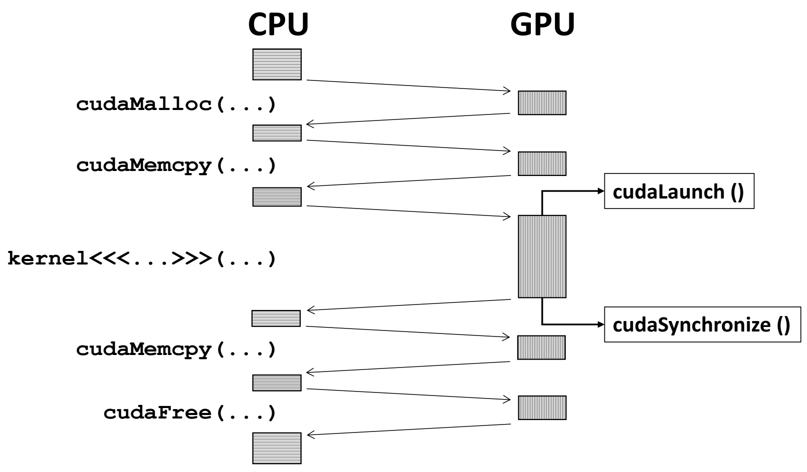


Figure 3 Phases of GPU Application under CUDA Runtime.

throttling CPU cores can significantly lower the overall system throughput. To minimize the negative throughput impact, we propose a new CPU scheduling algorithm, which we call the Throttle Fair Scheduler (TFS), to minimize the duration of CPU throttling without affecting memory bandwidth guarantees for real-time GPU applications.

Figure 2 shows the overall architecture of the BWLOCK++ framework on an integrated CPU-GPU architecture (NVIDIA Jetson TX2 platform). BWLOCK++ is comprised of three major components: (1) Dynamic run-time library for instrumenting GPU applications; (2) the Throttle Fair Scheduler; (3) Per-core B/W regulator. Working together, they protect real-time GPU kernels and minimize CPU throughput reduction. We will explain each component in the following sub-sections.

4.2 Automatic Instrumentation of GPU Applications

To eliminate manual programming efforts, we automatically instrument the program binary at the dynamic linker level. We exploit the fact that the execution of a GPU application using a GPU runtime library such as NVIDIA CUDA typically follows fairly predictable patterns. Figure 3 shows the execution timeline of a typical synchronous GPU application that uses the CUDA API.

In order to protect the runtime performance of a GPU application from co-running memory intensive CPU applications, we need to ensure that the GPU application automatically holds the memory bandwidth lock while a GPU kernel is executing on the GPU or performing a memory copy operation between CPU and GPU. Upon the completion of the execution of the kernel or memory copy operation, the GPU application again shall automatically release the bandwidth lock. This is done by instrumenting a small subset of CUDA API functions that are invoked when launching or synchronizing with a GPU kernel or while performing a memory copy operation. These APIs are documented in Table 1. More specifically, we write wrappers for these functions of interest which request/release bandwidth lock on behalf of the GPU application before calling the actual CUDA library functions. We compile these functions as a shared library and use Linux' `LD_PRELOAD` mechanism [8] to force the GPU application to use those wrapper functions whenever the CUDA functions are called. In this way, we automatically throttle CPU cores' bandwidth usage whenever real-time GPU kernels are being executed so that the GPU kernels' memory bandwidth can be guaranteed.

Table 1 CUDA APIs instrumented via `LD_PRELOAD` for BWLOCK++.

API	Action	Description
cudaConfigureCall	Update active streams	Specify the launch parameters for the CUDA kernel
cudaMemcpy	Acquire BWLOCK++ (*Before*) Release BWLOCK++ (*After*)	Perform synchronous memory copy between CPU and GPU
cudaMemcpyAsync	Acquire BWLOCK++ and update active streams	Perform asynchronous memory copy between CPU and GPU
cudaLaunch	Acquire BWLOCK++	Launch a GPU kernel
cudaDeviceSynchronize	Release BWLOCK++ and clear active streams	Block the calling CPU thread until all the previously requested tasks in a specific GPU device have completed
cudaThreadSynchronize	Release BWLOCK++ and clear active streams	Deprecated version of cudaDeviceSynchronize
cudaStreamSynchronize	Update active streams and release BWLOCK++ if there are no active streams	Block the calling CPU thread until all the previously requested tasks in a specific CUDA stream have completed

A complication to the automatic GPU kernel instrumentation arises when the application uses CUDA streams to launch multiple GPU kernels in succession in multiple streams and then waits for those kernels to complete. In this case, the bandwidth lock acquired by a GPU kernel launched in one stream can potentially be released when synchronizing with a kernel launched in another stream. In our framework, this situation is averted by keeping track of active streams and associating bandwidth lock with individual streams instead of the entire application whenever stream based CUDA APIs are invoked. A stream is considered active if:

- A kernel or memory copy operation is launched in that stream
- The stream has not been explicitly (using `cudaStreamSynchronize`) or implicitly (using `cudaDeviceSynchronize` or `cudaThreadSynchronize`) synchronized with

Our framework ensures that a GPU application continues holding the bandwidth lock as long as it has one or more active streams.

The obvious drawback of throttling CPU cores is that the CPU throughput may be affected especially if some of the tasks on the CPU cores are memory bandwidth intensive. In the following sub-section, we discuss the impact of throttling on CPU throughput and present a new CPU scheduling algorithm that minimizes throughput reduction.

4.3 Throttle Fair CPU Scheduler

As described earlier in this section, BWLOCK++ uses a throttling based approach to enforce memory bandwidth limit of CPU cores at a regular interval. Although effective in protecting critical GPU applications in the presence of memory intensive CPU applications, this approach runs into the risk of severely under-utilizing the system's CPU capacity; especially in cases when there are multiple best-effort CPU applications with different memory characteristics running on the best-effort CPU cores. In the throttling based design, once a core exceeds its memory bandwidth quota and gets throttled, that core cannot be used for the remainder of the period. Let us denote the regulation period as T (i.e., $T = 1ms$) and the time instant at which an offending core exceeds its bandwidth budget as t. Then the wasted time due to throttling can be described as $\delta = T - t$ and the smaller the value of t (i.e., throttled earlier in the period) the larger the penalty to the overall system throughput. The value of t depends on the rate at which a core consumes its allocated memory budget and that in turn depends on the memory characteristics of the application executing on that core. To maximize the overall system throughput, the value of δ should be minimized—that is if throttling never occurs, $t \geq T \Rightarrow \delta = 0$, or occurs late in the period, throughput reduction will be less.

4.3.1 Negative Feedback Effect of Throttling on CFS

One way to reduce CPU throttling is to schedule less memory bandwidth demanding tasks on the best-effort CPU cores while the GPU is holding the bandwidth lock. Assuming that each best-effort CPU core has a mix of memory bandwidth intensive and CPU intensive tasks, then scheduling the CPU intensive tasks while the GPU is holding the lock would reduce CPU throttling or at least delay the instant at which throttling occurs, which in turn would improve CPU throughput. Unfortunately, Linux's default scheduler CFS [13] actually aggravates the possibility of early and frequent throttling when used with BWLOCK++'s throttling mechanism.

The CFS algorithm tries to allocate fair amount of CPU time among tasks by using each task's weighted virtual runtime (i.e., weighted execution time) as the scheduling metric. Concretely, a task τ_i's virtual runtime V_i is defined as

$$V_i = \frac{E_i}{W_i} \tag{1}$$

where E_i is the actual runtime and W_i is the weight of the task. The CFS scheduler simply picks the task with the smallest virtual runtime.

The problem with memory bandwidth throttling under CFS arises because the virtual run-time of a memory intensive task, which gets frequently throttled, increases more slowly than the virtual run-time of a compute intensive task which does not get throttled. Due to this, the virtual runtime based arbitration of CFS tends to schedule the memory intensive tasks more than the CPU intensive tasks while bandwidth regulation is in place.

4.3.2 TFS Approach

In order to reduce the throttling overhead while keeping the undesirable scheduling of memory intensive tasks quantifiable, TFS modifies the throttled task's virtual runtime to take the task's throttled duration into account. Specifically, at each regulation period, if there exists a throttled task, we scale the throttled duration of the task by a factor, which we call TFS punishment factor, and add it to its virtual runtime.

Under TFS, a throttled task τ_i's virtual runtime V_i^{new} at the end of j^{th} regulation period is expressed as:

$$V_i^{new} = V_i^{old} + \delta_i^j \times \rho \tag{2}$$

where δ_i^j is the throttled duration of τ_i in the j^{th} sampling period, and ρ is the TFS punishment factor.

The more memory intensive a task is, the more likely the task get throttled in each regulation period for a longer duration of time (i.e., higher δ_i). By adding the throttled time back to the task's virtual runtime, we make sure that the memory intensive tasks are not

Table 2 Taskset for Example.

Task	Compute Time (C)	Period (P)	Description
τ_{RT}	4	15	Real-time task
τ_{MEM}	4	N/A	Memory intensive best-effort task
τ_{CPU}	4	N/A	CPU intensive best-effort task

favored by the scheduler. Furthermore, by adjusting the TFS punishment factor ρ, we can further penalize memory intensive tasks in favor of CPU intensive ones. This in turn reduces the amount of throttled time and improves overall CPU utilization. On the other hand, the memory intensive tasks will still be scheduled (albeit less frequently so) according to the adjusted virtual runtime. Thus, no tasks will suffer starvation.

Scheduling of tasks under TFS is fair with respect to the adjusted virtual runtime metric but it can be considered unfair with respect to the CFS's original virtual runtime metric. A task τ_i's "lost" virtual runtime Δ_i^{TFS} (due to TFS's inflation) over J regulation periods can be quantified as follows:

$$\Delta_i^{TFS} = \sum_{j=0}^{J} \delta_i^j \times \rho. \tag{3}$$

4.3.3 Illustrative Example

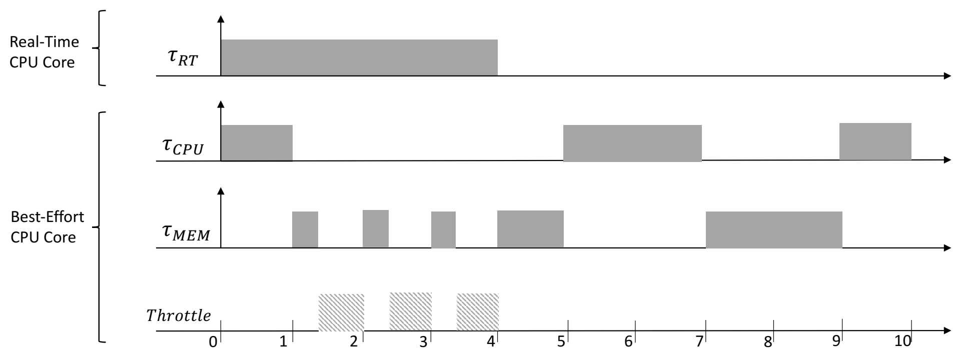
We elaborate the problem of CFS and the benefit of our TFS extension with a concrete illustrative example.

Let us consider a small integrated CPU-GPU system, which consists of two CPU cores and a GPU. We further assume, following our system model, that Core-1 is a real-time core, which may use the GPU, and Core-2 is a best-effort core, which doesn't use the GPU.

Table 2 shows a taskset to be scheduled on the system. The taskset is composed of a GPU using real-time task, which needs to be protected by our framework for the entire duration of its execution; and two best-effort tasks (of equal CFS priority), one of which is CPU intensive and the other is memory intensive.

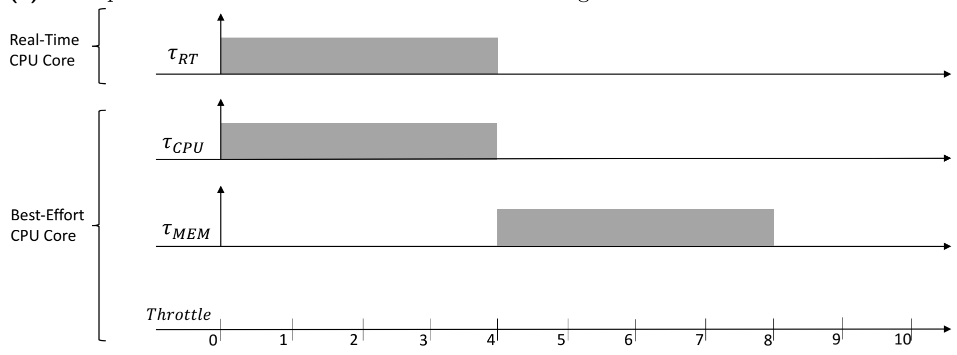
Figure 4a shows how the scheduling would work when CFS is used to schedule best-effort tasks τ_{CPU} and τ_{MEM} on the best-effort core with its memory bandwidth is throttled by our kernel-level bandwidth regulator. Note that in this example, both OS scheduler tick timer interval and the bandwidth regulator interval are assumed to be 1ms. At time 0, τ_{CPU} is first scheduled. Because τ_{CPU} is CPU bound, it doesn't suffer throttling. At time 1, the CFS schedules τ_{MEM} as its virtual runtime 0 is smaller than τ_{CPU}'s virtual runtime 1. Shortly after the τ_{MEM} is scheduled, however, it gets throttled at time 1.33 as it has used the best-effort core's allowed memory bandwidth budget for the regulation interval. When the budget is replenished at time 2, at the beginning of the new regulation interval, the τ_{MEM}'s virtual runtime is 0.33 while τ_{CPU} is 1. So, the CFS picks the τ_{MEM} (smaller of the two) again, which gets throttled again. This pattern continues until the τ_{MEM}'s virtual runtime finally catches up with τ_{CPU} at time 4 by which point the best-effort core has been throttled 66% of time between time 1 and 4. As can be seen in this example, CFS favors memory intensive tasks as their virtual runtimes increase more slowly than CPU intensive ones when memory bandwidth throttling is used.

Figure 4b shows a hypothetical schedule in which the execution of τ_{MEM} is delayed in favor of the τ_{CPU} while τ_{RT} is running (thus, memory bandwidth regulation is in place.)



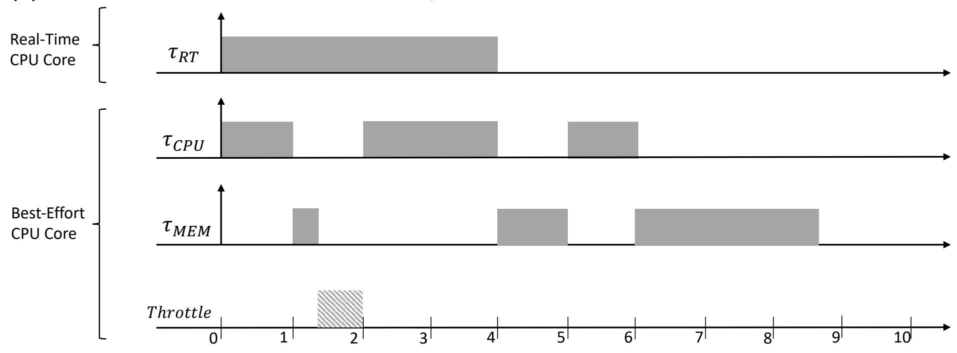
$V^{CPU}_{runtime}$	1	1	1	1	1	2	3	3	3	4
$V^{MEM}_{runtime}$	0	0.33	0.67	1	2	2	2	3	4	4

(a) Example schedule under CFS with 1-msec scheduling tick.



$V^{CPU}_{runtime}$	1	2	3	4	4	4	4	4		
$V^{MEM}_{runtime}$	0	0	0	0	1	2	3	4		

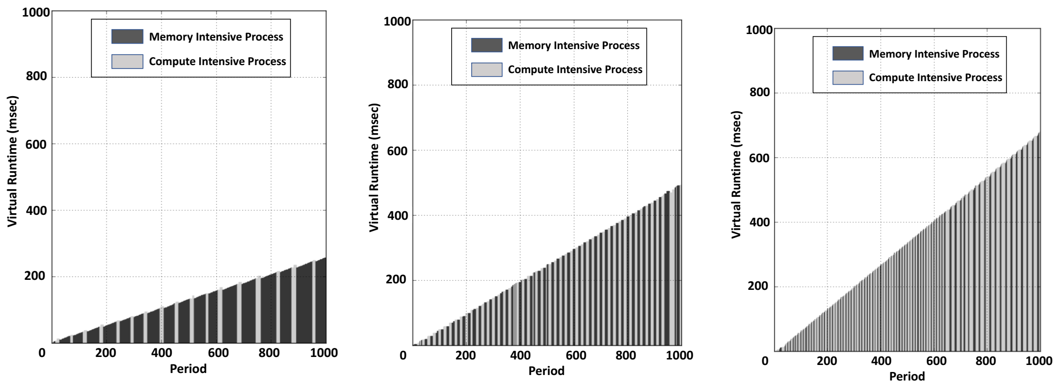
(b) Example schedule with zero throttling.



$V^{CPU}_{runtime}$	1	1	2	3	3	4	4	4	4	
$V^{MEM}_{runtime}$ (TFS-3X)	0	2.34	2.34	2.34	3.34	3.34	4.34	5.34	6	
$V^{MEM}_{runtime}$ (Actual)	0	0.33	0.33	0.33	1.33	1.33	2.33	3.33	4	

(c) Example schedule under TFS with $\rho = 3$.

Figure 4 Example schedules under different scheduling schemes.



(a) CFS. **(b)** TFS. **(c)** TFS-3X.

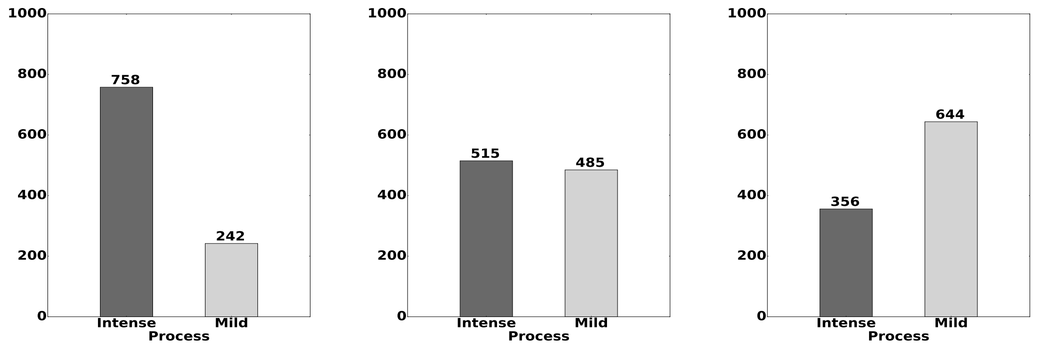
Figure 5 Virtual runtime progress of the two synthetic tasks. One is cpu-intensive and the other is memory-intensive.

In this case, because τ_{CPU} never exhausts the memory bandwidth budget, it never gets throttled. As a result, the best-effort core never experiences throttling and thus is able to achieve high throughput. While this is ideal behavior from the perspective of throughput, it may not be ideal for the τ_{MEM} as it can suffer starvation.

Figure 4c shows the schedule under the TFS (with a TFS punishment factor $\rho = 3$). The TFS works identical to CFS until at time 2, when the BWLOCK++'s periodic timer is called. At this point, the τ_{MEM}'s virtual runtime (V^{MEM}) is 0.33ms. However, because it has been throttled for 0.67ms during the regulation period ($\delta = 0.67$), according to Equation 2, TFS increases the task's virtual runtime to 2.34 ($V^{MEM} + \delta \times \rho = 0.33 + 0.67 \times 3 = 2.34$). Because of the increased virtual runtime, the TFS scheduler then picks τ_{CPU} as its virtual runtime is now smaller than that of τ_{MEM} ($1 < 2.34$). Later, when the τ_{CPU}'s virtual runtime becomes 3 at time 4, the TFS scheduler can finally re-schedule the τ_{MEM}. In this manner, TFS favors CPU intensive tasks over memory-intensive ones, while preventing starvation of the latter. Note that TFS works at each regulation period (i.e., 1ms) independently and thus automatically adapts to the task's changing behavior. For example, if a task is memory intensive only for a brief period of time, the task will be throttled only for the memory intensive duration, and the throttled time will be added back to the task's virtual runtime at each 1ms regulation period. Furthermore, even for a period when a task is throttled, the task always makes small progress as allowed by the memory bandwidth budget for the period. Therefore, no task suffers complete starvation for an extended period of time.

4.3.4 Effects of TFS using Synthetic Tasks

We experimentally validate the effect of TFS in scheduling best-effort tasks on a real system. In this experiment, we use two synthetic tasks: one is CPU intensive and the other is memory-intensive. We use *Bandwidth* benchmark for both of these tasks. In order to make *Bandwidth* memory intensive, we configure its working-set size to be twice the size of LLC on our platform. Similarly, to make *Bandwidth* compute (CPU) intensive, we make its working set size one half the size of L1 data cache in our platform. We assign these two best-effort tasks on the same best-effort core, which is bandwidth regulated with a 100 MB/s memory bandwidth budget.



(a) CFS **(b)** TFS **(c)** TFS-3X

Figure 6 The number of periods during which the two tasks are scheduled. 'Intense' refers to the memory-intensive task. 'Mild' refers to the CPU-intensive task.

Figure 5 shows the virtual runtime progression over 1000 sampling periods of the two tasks under three scheduler configurations: CFS, TFS ($\rho = 1$), and TFS-3X ($\rho = 3$). In CFS, the memory intensive process gets preferred by the CFS scheduler at each scheduling instance, because its virtual run-time progresses more slowly. In TFS and TFS-3X, however, as memory-intensive task's virtual runtime is increased, CPU-intensive task is scheduled more frequently.

This can be seen more clearly in Figure 6, which shows the number of periods utilized by each task on the CPU core, over the course of one thousand sampling periods. Under CFS, out of all the sampling periods, 75% are utilized by the memory intensive process and only 25% are utilized by the compute intensive process. With TFS, the two tasks get to run in roughly the same number of sampling periods whereas in TFS-3x, the CPU intensive task gets to run more than the memory intensive task.

5 Implementation

In this section, we describe the implementation details of BWLOCK++.

5.1 BWLOCK++ System Call

We add a new system call `sys_bwlock` in Linux kernel 4.4.38. The system call serves two purposes. 1) It acquires or releases the memory bandwidth lock on behalf of the currently running task on the real-time core; and 2) it implements a priority-ceiling protocol, which boosts the calling task's priority to the system's ceiling priority, to prevent preemption. We introduce two new integer fields, `bwlock_val`, `bw_old_priority`, in the task control block: `bwlock_val` stores the current status of the memory bandwidth lock and `bw_old_priority` keeps track of the original real-time priority of the task while it is holding the bandwidth lock.

Algorithm 1 shows the implementation of the system call. To acquire the memory bandwidth lock, the system call must be invoked from the real-time system core and the task currently scheduled on the real-time core must have a real-time priority (*line 2*). At the time of acquisition of bandwidth lock, the priority of the calling task, which is tracked by

Algorithm 1: BWLOCK++ System Call.

```
1  syscall sys_bwlock(bw_val)
2      if smp_processor_id () == RT_CORE_ID ∧ rt_task (current) then
3          rt_core_data := get_rt_core_data ()
4          rt_core_data → current_task := current
5          if bw_val ≥ 1 then
6              current → bwlock_val := 1
7              current → bw_old_priority := current → rt_priority
8              current → rt_priority := MAX_USER_RT_PRIO - 1
9          else
10             current → bwlock_val := 0
11             current → rt_priority := current → bw_old_priority
12         end
13     end
14 return;
```

the globally accessible `current` pointer in Linux kernel, is raised to the maximum allowed real-time priority value (the ceiling priority) for any user-space task to prevent preemption (*line 7*). The real-time priority value of the the task is restored to its original priority value when the bandwidth lock is released (*line 10*). In this manner, the system call updates the state of the currently scheduled real-time task on the real-time system core, which is then used by the memory bandwidth regulator on best-effort cores to enforce memory usage thresholds, as explained in the following subsection.

5.2 Per-Core Memory Bandwidth Regulator

The per-core memory bandwidth regulator is composed of a periodic timer interrupt handler and a performance monitoring counter (PMC) overflow interrupt handler. Algorithm 2 shows the implementation of the memory bandwidth regulator.

The periodic timer interrupt handler is invoked at a periodic interval (currently every 1 msec) using a high resolution timer in each best-effort core. The timer handler begins a new bandwidth lock regulation period and performs the following operations:

- Unthrottle the core if it was throttled in the last regulation period (*line 3*)
- Scale the virtual runtime of the task currently scheduled on the core based on the throttling time in the last period and the TFS punishment factor (*line 4-5*)
- Determine the new memory usage budget based on the bandwidth lock status of the task currently scheduled on the real-time system core (*line 7-12*)
- Program the performance monitoring counter on the core based on the new memory usage budget for the current regulation period (*line 13*). We use the *L2D_CACHE_REFILL* event for measuring the memory bandwidth traffic in ARM Cortex-A57 processor core

The PMC overflow interrupt occurs when the core at hand exceeds its memory usage budget in the current regulation period. The interrupt handler prevents further memory transactions from this core by scheduling a high priority idle kernel thread on it for the remainder of the regulation period (*line 17*).

Algorithm 2: Memory Bandwidth Regulator.

```
procedure periodic_interrupt_handler(core_data)
    if core_is_throttled (core_data → core_id) == TRUE then
        unthrottle_core (core_data → core_id)
        record_throttling_end_time (core_data → current_task)
        scale_virtual_runtime (core_data → current_task)
    end
    rt_core_data := get_rt_core_data ()
    if rt_core_data → current_task → bwlock_val == 1 then
        core_data → new_budget := rt_core_data → throttle_budget
    else
        core_data → new_budget := MAX_BANDWIDTH_BUDGET
    end
    program_pmc (core_data → new_budget)
return;

procedure pmc_overflow_handler(core_data)
    record_throttling_start_time (core_data → current_task)
    throttle_core (core_data → core_id)
return;
```

6 Evaluation

In this section, we present the experimental evaluation results of BWLOCK++.

6.1 Setup

We evaluate BWLOCK++ on NVIDIA Jetson TX2 platform. We use the Linux kernel version 4.4.38, which is patched with the changes required to support BWLOCK++. The CUDA runtime library version installed on the platform is 8.0, which is the latest version available for Jetson TX2 at the time of writing. In all our experiments, we place the platform in maximum performance mode by maximizing GPU and memory clock frequencies and disabling the dynamic frequency scaling of CPU cores. We also shutdown the graphical user interface and disable the network manager to avoid run to run variation in the experiments. As per our system model, we designate the Core-0 in our system as real-time core. The remaining cores execute best-effort tasks only. All the tasks are statically assigned to their respective cores during the experiment. While NVIDIA Jetson TX2 platform contains two CPU islands, a quad-core Cortex-A57 and a dual-core Denver, we only use the Cortex-A57 island for our evaluation and leave the Denver island off because we were unable to find publicly available documentation regarding the Denver cores' hardware performance counters, which is needed to implement throttling. In order to evaluate BWLOCK++, we use six benchmarks from parboil suite which are listed as memory bandwidth sensitive in [18].

Table 3 GPU execution time breakdown of selected benchmarks.

Benchmark	Dataset	Copy (KBytes)	Timing Breakdown (msec)			
			Kernel (G^e)	Copy (G^m)	Compute (C)	Total (E)
histo	Large	5226	83409	18	0	83428
sad	Large	709655	152	654	53	861
bfs	1M	62453	174	72	0	246
spmv	Large	30138	69	51	10	131
stencil	Default	196608	749	129	9	888
lbm	Long	379200	43717	358	2004	46080

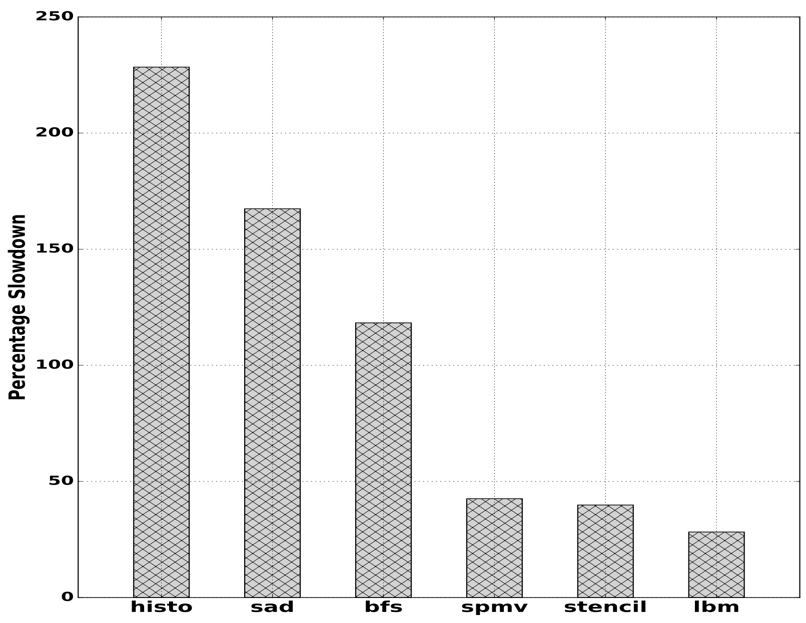


Figure 7 Slowdown of the total execution time of GPU benchmarks due to three *Bandwidth* corunners.

6.2 Effect of Memory Bandwidth Contention

In this experiment, we investigate the effect of memory bandwidth contention due to co-scheduled memory intensive CPU applications on the evaluated GPU kernels.

First, we measure the execution time of each GPU benchmark in isolation. From this experiment, we record the GPU kernel execution time (G^e), memory copy time for GPU kernels (G^m) and CPU compute time (C) for each benchmark. The data collected is shown in Table 3. We then repeat the experiment after co-scheduling three instances of a memory intensive CPU application as co-runners. We use the *Bandwidth* benchmark from the IsolBench suite [19] as the memory intensive CPU benchmark, which updates a big 1-D array sequentially. The sequential write access pattern of the benchmark is known to cause worst-case interference on several multicore platforms [20].

The results of this experiment are shown in Figure 7 and they demonstrate how much the total execution time of GPU benchmarks ($E = G^e + G^m + C$) suffers from memory bandwidth contention due to the co-scheduled CPU applications.

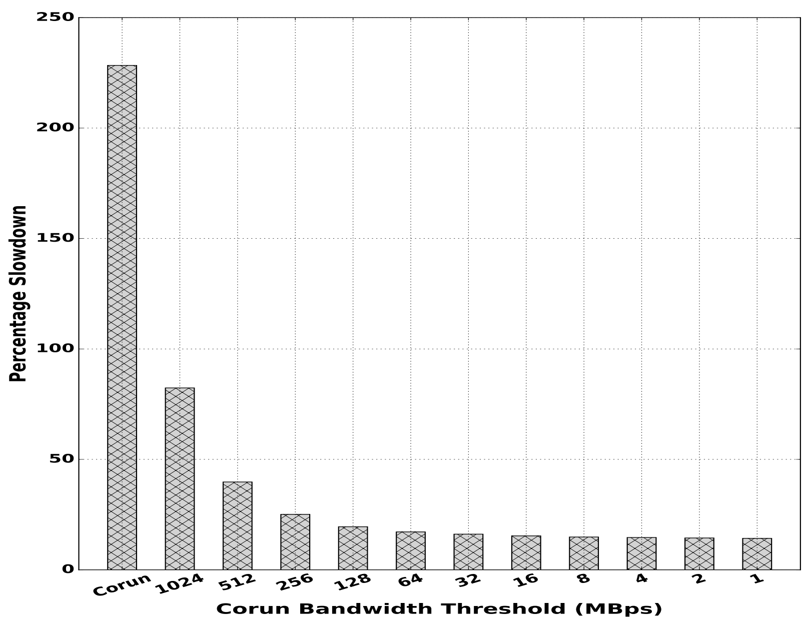


Figure 8 Effect of corun bandwidth threshold on the execution time of histo benchmark.

From Figure 7, it can be seen that the worst case slowdown, in case of *histo* benchmark, is more than 250%. Similarly, for SAD benchmark, the worst case slowdown is more than 150%. For all other benchmarks, the slowdown is non-zero and can be significant in affecting the real-time performance. These results clearly show the danger of uncontrolled memory bandwidth sharing in an integrated CPU-GPU architecture as GPU kernels may potentially suffer severe interference from co-scheduled CPU applications. In the following experiment, we investigate how this problem can be addressed by using BWLOCK++.

6.3 Determining Memory Bandwidth Threshold

In order to apply BWLOCK++, we first need to determine safe memory budget that can be given to the best-effort CPU cores in the presence of GPU applications. However, an appropriate threshold value may vary depending on the characteristics of individual GPU applications. If the threshold value is set too high, then it may not be able to protect the performance of the GPU application. On the other hand, if the threshold value is set too low, then the CPU applications will be throttled more often and that would result in significant CPU capacity loss.

We calculate the safe memory budget for best-effort CPU cores by observing the trend of slowdown of the total execution time of GPU application as the allowed memory usage threshold of CPU co-runners is varied. We start with a threshold value of 1-GB/s for each best-effort CPU core. We then continue reducing the threshold value for best-effort cores by half and measure the impact of this reduction on the slowdown of execution time (E) of the benchmark.

Figure 8 shows this trend for the execution time of *histo* benchmark from the parboil suite. From the figure, it can be seen that after `64-MBps` threshold value for best-effort CPU cores, further reduction of threshold value does not yield significant improvement in reducing the slowdown of benchmark. Hence, for *histo* benchmark, we select `64-MBps` as the threshold value for the best-effort CPU cores. In a similar fashion, we plot this trend for all the selected benchmarks and determine the value of corun threshold for the best-effort CPU cores.

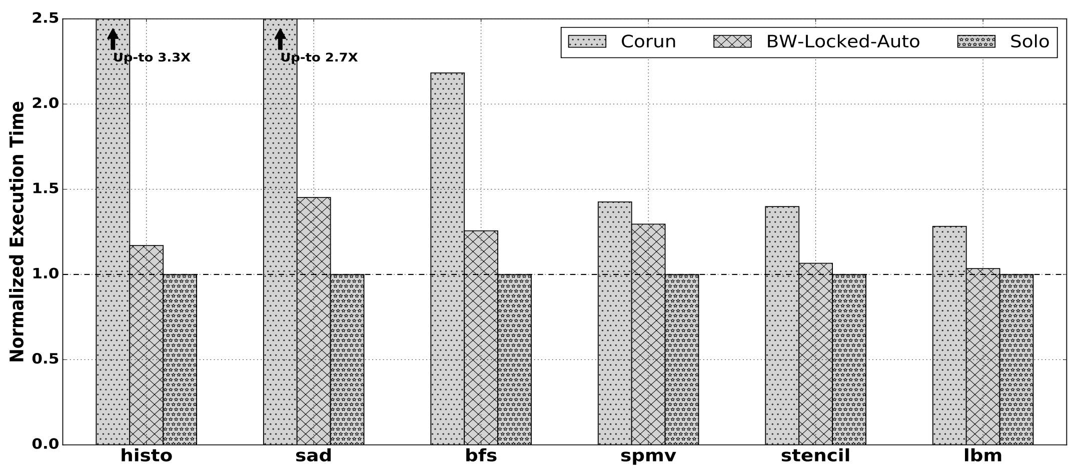


Figure 9 BWLOCK++ Evaluation Results.

6.4 Effect of BWLOCK++

In this experiment, we evaluate the performance of BWLOCK++. Specifically, we record the corun execution of GPU benchmarks with the automatic instrumentation of BWLOCK++. We call this scenario *BW-Locked-Auto*. We compare the performance under *BW-Locked-Auto* against the *Solo* and *Corun* execution of the GPU benchmarks which represent the measured execution times in isolation and together with three co-scheduled memory intensive CPU applications, respectively.

To get the data-points for *BW-Locked-Auto*, we configure BWLOCK++ according to the allowed memory usage threshold of the benchmark at hand and use our dynamic GPU kernel instrumentation mechanism to launch the benchmark in the presence of three *Bandwidth* benchmark instances (write memory access pattern) as CPU co-runners. The results of this experiment are plotted in Figure 9. In Figure 9, we plot the total execution time of each benchmark for the above mentioned scenarios. All the time values are normalized with respect to the total execution time ($E_{solo} = C_{solo} + G^{e}_{solo} + G^{m}_{solo}$) of the benchmark in isolation. As can be seen from this figure, execution under *BW-Locked-Auto* incurs significantly less slowdown of the total execution time of GPU benchmarks due to reduction of both GPU kernel execution time and memory copy operation time.

6.5 Throughput improvement with TFS

As explained in Section 4.3, throttling under CFS results in significant system throughput reduction. In order to illustrate this, we conduct an experiment in which the GPU benchmarks are executed with six CPU co-runners. Each CPU core, apart from the one executing the GPU benchmark, has a memory intensive application and a compute intensive application scheduled on it. For both of these applications, we use the *Bandwidth* benchmark with different working set sizes. In order to make *Bandwidth* memory intensive, we configure its working set size to be twice the size of LLC on our evaluation platform. Similarly for compute intensive case, we configure the working set size of *Bandwidth* to be half of the L1-data cache size. We record the total system throttle time statistics with BWLOCK++ for all the GPU benchmarks. The total system throttle time is the sum of throttle time across all system cores. We then repeat the experiment with our Throttle Fair Scheduling scheme. In *TFS-1*, we configure the TFS punishment factor as one for the memory intensive threads

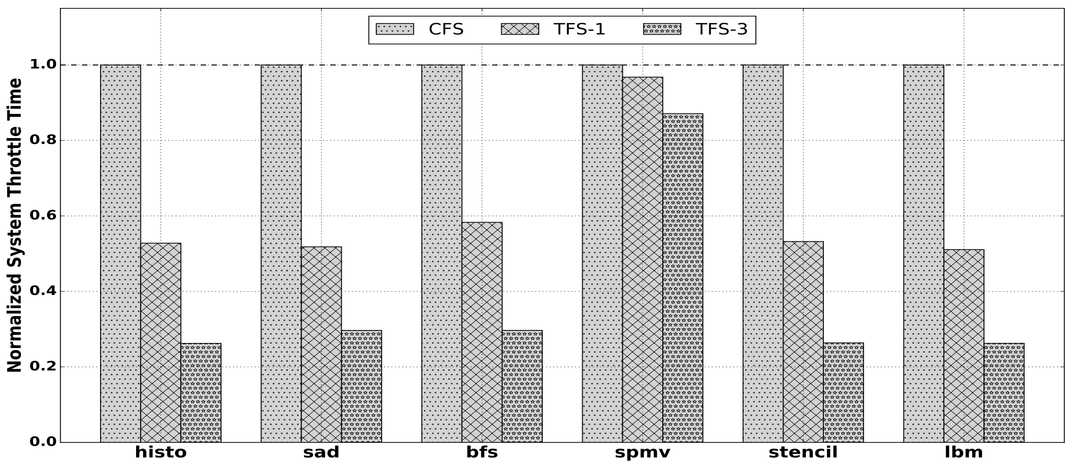


Figure 10 Comparison of total system throttle time under different scheduling schemes.

and in *TFS-3*, we set this factor to three. We plot the normalized total system throttle time for all the scheduling schemes and present them in Figure 10. It can be seen that TFS results in significantly less system throttling (On average, 39% with *TFS-1* and 62% with *TFS-3*) as compared to CFS.

6.6 Overhead due to BWLOCK++

The overhead incurred by real-time GPU applications due to BWLOCK++ comes from the following sources:

- `LD_PRELOAD` overhead for CUDA API instrumentation
- Overhead due to BWLOCK++ system call

The overhead due to `LD_PRELOAD` is negligible since we cache CUDA API symbols for all the instrumented functions inside our shared library; after searching for them only once through the dynamic linker. We calculate the overhead incurred due to BWLOCK++ system call by executing the system call one million times and taking the average value. In NVIDIA Jetson TX2, the average overhead due to each BWLOCK++ system call is $1.84usec$. Finally, we experimentally determine the overhead value for all the evaluated benchmarks by running the benchmark in isolation with and without BWLOCK++. Our experiment shows that for all the evaluated benchmarks, the total overhead due to BWLOCK++ is less than 1% of the total solo execution time of the benchmark.

7 Schedulability Analysis

As we limit the scheduling of real-time tasks on a single real-time core, our system can be analyzed using the classical unicore based response time analysis for preemptive fixed priority scheduling with blocking [3], because we model each GPU execution segment as a critical section, which is protected by acquiring and releasing the bandwidth lock. The bandwidth lock serializes GPU execution and regulates memory bandwidth consumption of co-scheduled best-effort CPU tasks. The bandwidth lock implements the standard priority ceiling protocol [17], which boosts the priority of the lock holding task (i.e., the task executing a GPU kernel) to the ceiling priority of the lock, which is the highest real-time priority of the system, so as to prevent preemption. With this constraint, a real-time task τ_i's response time is expressed as:

$$R_i^{n+1} = E_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil E_j \tag{4}$$

where $hp(i)$ represents the set of higher priority tasks than τ_i and B_i is the longest GPU kernel or copy duration—protected by the memory bandwidth lock—of one of the lower priority tasks.

The benefit of BWLOCK++ lies in the reduction of worst-case GPU kernel execution or GPU memory copy interval of real-time tasks (which would in turn reduce E_i and B_i terms in Equation 4). As shown in Section 6.2, without BWLOCK++, GPU execution of a task can suffer severe slowdown (up to 230% slowdown in our evaluation), which would result in pessimistic WCET estimation for GPU kernel and copy execution times, hampering schedulability of the system. BWLOCK++ helps reduce pessimism of GPU execution time estimation and thus improves schedulability.

8 Discussion

Our approach has following limitations. First, we assume that all real-time tasks are scheduled on a single dedicated real-time core while the rest of the cores only schedule best-effort tasks. In addition, we assume only real-time tasks can utilize the GPU while best-effort tasks cannot. While restrictive, recall that scheduling multiple GPU using real-time tasks on a single dedicated real-time core does not necessarily reduce GPU utilization because multiple GPU kernels from different tasks (processes) are serialized at the GPU hardware anyway [15] as we already discussed in Section 3. Also, due to the capacity limitation of embedded GPUs, it is expected that a few GPU using real-time task can easily achieve high GPU utilization in practice. We claim that our approach is practically useful for situations where a small number of GPU accelerated tasks are critical, for example, a vision-based automatic braking system.

Second, we assume that GPU applications are given a priori and they can be profiled in advance so that we can determine proper memory bandwidth threshold values. If this assumption cannot be satisfied, an alternative solution is to use a single threshold value for all GPU applications, which eliminates the need of profiling. But the downside is that it may lower the CPU throughput because the memory bandwidth threshold must be conservatively set to cover all types of GPU applications.

9 Conclusion

In this paper, we presented BWLOCK++, a software based mechanism for protecting the performance of GPU kernels on platforms with integrated CPU-GPU architectures.

BWLOCK++ automatically instruments GPU applications at run-time and inserts a memory bandwidth lock, which throttles memory bandwidth usage of the CPU cores to protect performance of GPU kernels. We identified a side effect of memory bandwidth throttling on the performance of Linux default scheduler CFS, which results in the reduction of overall system throughput. In order to solve the problem, we proposed a modification to CFS, which we call Throttle Fair Scheduling (TFS) algorithm. Our evaluation results have shown that BWLOCK++ effectively protects the performance of GPU kernels from memory intensive CPU co-runners. Also, the results showed that TFS improves system throughput, compared to CFS, while protecting critical GPU kernels. In the future, we plan to evaluate BWLOCK++ on other integrated CPU-GPU architecture based platforms. We also plan to extend BWLOCK++ not only to protect critical GPU tasks but also to protect critical CPU tasks.

References

1 Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

2 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

3 N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

4 Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based gpu arbitration mechanism for memory accesses. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2017.

5 NVIDIA Corp. Nvidia jetson platforms. `https://developer.nvidia.com/embedded-computing`.

6 Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. Gpusync: A framework for real-time gpu management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

7 Björn Forsberg, Andrea Marongiu, and Luca Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous soc. In *Design, Automation & Test in Europe (DATE)*, 2017.

8 Greg Kroah Hartman. Modifying a dynamic library without changing the source code | linux journal. `http://www.linuxjournal.com/article/7795`.

9 Shinpei Kato, Karthik Lakshmanan, Ragunathan (Raj) Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference (ATC)*, 2011.

10 Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Brandt Scott. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference (ATC)*, 2012.

11 Shinpei Kato, Eijiro Takeuchi, Yoshiki Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

12 Hyoseung Kim, Pratyush Patel, Shige Wang, and Ragunathan (Raj) Rajkumar. A server based approach for predictable gpu access control. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

13 Ingo Molnar. Modular scheduler core and completely fair scheduler. `https://lwn.net/Articles/230501`.

14 Nathan Otterness, Ming Yang, Sarah Rust, and Eunbyun Park. Inferring the scheduling policies of an embedded cuda gpu. In *Workshop on Operating Systems Platforms for Embedded Real Time Systems Applications (OSPERT)*, 2017.

15 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alexander C. Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

16 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

17 Lui Sha, Ragunathan (Raj) Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

18 John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

19 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

20 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing isolation challenges of non-blocking caches for multicore real-time systems. *Real-Time Systems*, 53(5):673–708, 2017.

21 Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers (TC)*, PP(99):1–1, 2016.

22 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

23 Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: a preemptive execution system for gpgpu computing. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems

Ming Yang
The University of North Carolina at Chapel Hill, USA
yang@cs.unc.edu

Nathan Otterness
The University of North Carolina at Chapel Hill, USA
otternes@cs.unc.edu

Tanya Amert
The University of North Carolina at Chapel Hill, USA
tamert@cs.unc.edu

Joshua Bakita
The University of North Carolina at Chapel Hill, USA
jbakita@cs.unc.edu

James H. Anderson
The University of North Carolina at Chapel Hill, USA
anderson@cs.unc.edu

F. Donelson Smith
The University of North Carolina at Chapel Hill, USA
smithfd@cs.unc.edu

Abstract

NVIDIA's CUDA API has enabled GPUs to be used as computing accelerators across a wide range of applications. This has resulted in performance gains in many application domains, but the underlying GPU hardware and software are subject to many non-obvious pitfalls. This is particularly problematic for safety-critical systems, where worst-case behaviors must be taken into account. While such behaviors were not a key concern for earlier CUDA users, the usage of GPUs in autonomous vehicles has taken CUDA programs out of the sole domain of computer-vision and machine-learning experts and into safety-critical processing pipelines. Certification is necessary in this new domain, which is problematic because GPU software may have been developed without any regard for worst-case behaviors. Pitfalls when using CUDA in real-time autonomous systems can result from the lack of specifics in official documentation, and developers of GPU software not being aware of the implications of their design choices with regards to real-time requirements. This paper focuses on the particular challenges facing the real-time community when utilizing CUDA-enabled GPUs for autonomous applications, and best practices for applying real-time safety-critical principles.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems, Computer systems organization → Embedded software, Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Software and its engineering → Scheduling, Software and its engineering → Concurrency control, Software and its engineering → Process synchronization

Keywords and phrases real-time systems, graphics processing units, scheduling algorithms, parallel computing, embedded software

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.20

Funding Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.

1 Introduction

A fundamental shift is reshaping how real-time analysis is applied in all forms of autonomous systems (*e.g.*, UAVs, robotics, and, especially, self-driving automobiles). These systems are increasingly dependent on escalating computational requirements for various applications based on machine learning (ML). Examples include computer-vision applications that recognize people and objects in high-bit-rate streams from multiple video cameras, and applications that process 3-D models of the surrounding environment from high-volume streams of LIDAR data. These and other ML applications in autonomous vehicles have prompted the adoption of specialized computing accelerators to match computational demands. Graphics processing units (GPUs) are among the most prominent and accessible of these specialized accelerators because of their high-throughput performance. While high throughput is necessary for ML applications based on multiple streams of sensor inputs, it alone is not sufficient. *Safe operation of autonomous vehicles also requires temporal correctness from GPU-using tasks* – this is where real-time analysis becomes essential for autonomous systems.

Why there is a problem. Unfortunately, GPUs present many challenges, so modeling, analyzing, and certifying a safety-critical autonomous system using GPUs is currently beyond the state-of-the-art. One reason is that GPUs are fundamentally different from CPUs. Real-time analysis is based on well-understood scheduling algorithms that allocate CPU capacity. In contrast, GPU hardware and software together implement GPU-specific scheduling algorithms that are proprietary, opaque, and can change without notice. Modeling and analysis efforts under these conditions are subject to many pitfalls when applied to real-time safety-critical workloads in GPU-using autonomous systems.

Focus of this paper. Our motivation for this work is to provide guidance, recommendations, and warnings about numerous pitfalls to both research and implementation practitioners. We have found that writing programs for real-time tasks that combine CPU and GPU computations is harder than we first thought. Based on several years of study, experimentation, and experience with GPU programming, we are presenting here a compendium of specific issues that are essential background for developing task systems where real-time design meets GPUs.

Choice of GPU platforms. We base our findings on our experiences with NVIDIA GPUs for a number of reasons. The most salient reason is that *NVIDIA GPUs are in cars on the road today.* Further, NVIDIA has positioned itself as a market leader in automotive applications. For example, NVIDIA's "Jetson" line of embedded platforms specifically targets autonomous systems, and is marketed as "the embedded platform for autonomous everything" [21]. Three generations of the Jetson series of embedded single-board computers have been produced by NVIDIA; the TK1, TX1, and TX2. NVIDIA also markets a higher-performance line of embedded platforms, the "Drive PX" series, which includes multiple models such as the Drive PX2, Drive PX Xavier, and Drive PX Pegasus.

NVIDIA GPUs serve as an exemplar of the push for throughput over predictability in GPUs . Recent developments in the NVIDIA GPU ecosystem are focused on improving ML applications, especially those for autonomous driving. Most of these improvements center around increasing throughput or reducing execution latency, but little, if any, attention has been paid to requirements of the real-time tasks used in autonomous systems. This lack of attention is evident in the sparse efforts by NVIDIA to improve or document GPU scheduling behavior or improve the predictability of GPU execution times.

1.1 Contributions

The major contribution of this paper lies in discussing pitfalls for real-time GPU usage of relevance to both those conducting research on autonomous systems and those who design and build them. These pitfalls fall within three categories:

Synchronization and blocking. In any task consisting of a combination of CPU and GPU computations, there are necessary synchronization points (*e.g.*, a CPU program needs to wait until a GPU has produced a result). Synchronization inherently leads to blocking terms in scheduling analysis. Unfortunately, we have learned that why and when synchronization blocking occurs in a GPU-using task is not straightforward to determine. Further, some forms of synchronization can lead to significant capacity loss on both CPUs and GPUs. We have constructed experiments that expose these synchronization effects and carefully describe them along with a list of specific pitfalls the unwary programmer may encounter. This contribution is fully presented in Sec. 3.

GPU concurrency. We have realized that there is a fundamental trade-off that exists for designing real-time tasks that use a GPU. A conventional choice is to write and execute the task program as an operating system (OS) process in its own non-shared address space. This provides cross-task memory isolation. If this choice is used, however, the NVIDIA GPU programming environment (described in Sec. 2) does not permit any concurrent computations on the GPU even if sufficient GPU resources are available. Depending on how GPU programs are organized and written, this can lead to capacity loss on the GPU. The alternate choice is to write and execute a task as a schedulable thread that shares a process address space with other task threads. Cross-task memory isolation is lost, but the GPU programming environment provides mechanisms that allow concurrent computations on the GPU. NVIDIA provides a third option with a middleware environment that is claimed to provide the best of both choices – memory isolation with concurrency enabled. We have performed a case study using algorithms that are exemplars for computer-vision tasks in autonomous vehicles to evaluate these trade-off options. The results and guidelines are fully presented in Sec. 4.

CUDA programming perils. Our research has necessarily involved constructing many thousands of lines of GPU programming for performing experiments. This experience has been especially enlightening about the perils one can encounter in programming for NVIDIA GPUs. The perils span a spectrum of pain ranging from simple documentation errors to functions that default in strange ways, to programming "gotchas." We present a list of perils with descriptions and examples of the ones most likely to cause problems in Sec. 5.

Value for autonomous systems. We believe that this paper will help bridge the gap between research and implementation in autonomous systems. For example, real-time researchers may not be familiar with GPU programming for applications of ML and other forms of AI used in real-time tasks. Likewise, programmers responsible for implementations are given little guidance about creating GPU-using task systems amenable to real-time analysis.

We provide the necessary understanding required to apply GPUs in real-time tasks while avoiding numerous hidden pitfalls. We also expose GPU-related issues that must be mitigated for real-time guarantees to be possible in autonomous systems. We further believe that the fundamental issues presented herein are relevant to any real-time application using computational accelerators, and likely hold for other manufacturers' GPUs, digital signal processors (DSPs), or FPGAs.

1.2 Related Work

Treating GPUs as non-shared devices has been a consistent theme in much of the prior research on GPU scheduling for real-time systems. More predictable execution times result from restricting access to the entire GPU (or its independent execution and data movement components) to a single task at a time [11, 15, 16, 28, 29, 27, 31].

Other prior research takes a slightly different approach and improves schedulability by simulating preemptive execution [3, 15, 17, 33]. These designs typically split GPU computations into smaller fragments, which can be individually scheduled and preempted. One of these frameworks, called Kernelet [32], even allows GPU sharing as a means to improve utilization, but interference effects caused by sharing are not addressed.

The decision to treat GPUs as non-shared devices is largely motivated by a perceived need to work with a greatly simplified *model* of GPU execution (resulting, we believe, primarily from a lack of information from GPU manufacturers). If GPU scheduling behavior is an opaque "black box," it is a rational conclusion that sharing must be avoided because execution ordering and interference effects *cannot be known.* Our research is motivated, however, by an observation that GPU sharing will become essential for effectively utilizing less-capable embedded GPUs. Our research goal is to enable the modeling and analysis of a combined CPU+GPU scheduling framework that allows real-time tasks to share multicore CPUs and one or more GPUs.

We began our research by experimentally investigating the impacts of GPU sharing on the NVIDIA Jetson TK1 [24] and TX1 [25]. In these studies, we focused on GPU sharing by CPU processes (tasks) that have separate address spaces. We found that sharing in this context happens only through round-robin time-sliced multiplexing of GPU computations onto the GPU execution hardware. This multiplexing form of scheduling presents many challenges for modeling and analysis. In later work, we experimentally investigated GPU sharing by CPU tasks that share an address space (threads) on both the TX1 [26] and the more-capable TX2 [1]. In these studies, we found that truly concurrent sharing can indeed occur and deduced rules the GPU uses to schedule execution.

The work summarized so far was all directed at scheduling real-time tasks that use a GPU for parts of their executions. Other work has focused on timing analysis for GPU workloads [4, 5, 6, 7, 8], techniques for remedying performance bottlenecks [13], direct I/O communication [2], and techniques for managing or evaluating GPU hardware resources, including the cache and DRAM [9, 10, 12, 14, 18, 19, 30].

2 Background

In this section, we provide background information on the NVIDIA GPUs used in this research. The CUDA programming framework is described, and a simple example of a CUDA program is explained.

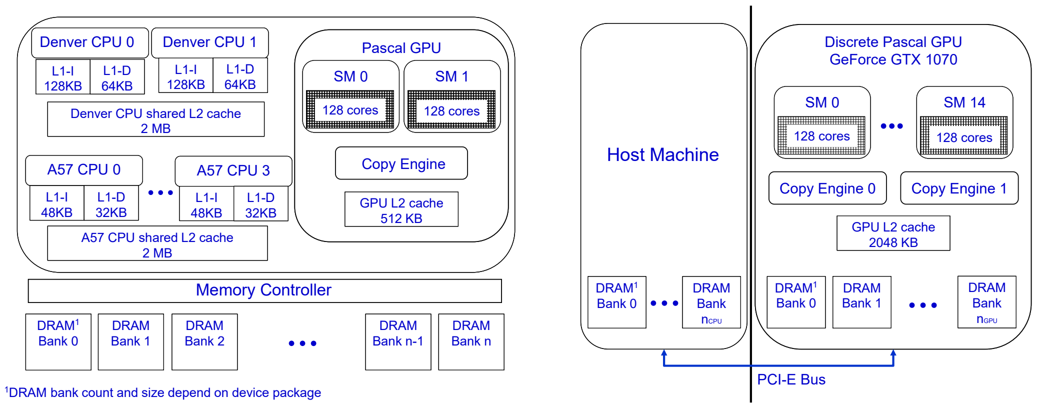


Figure 1 Jetson TX2 Architecture (left) and GeForce GTX 1070 Architecture (right).

2.1 CUDA-Enabled Devices

The work presented here refers to the Kepler, Maxwell, Pascal, and Volta architectures of NVIDIA GPUs. NVIDIA introduced these four different generations of GPU architectures, in that order, within a time span of about five years (2012 - 2017) – a pace of change more rapid than normally seen in CPU generations. GPUs are programmed using the CUDA API, which is an NVIDIA-provided set of libraries and language extensions for C/C++.

We consider both *discrete* GPUs and *integrated* GPUs. An integrated GPU, such as the NVIDIA Jetson TX2 shown on the left in Fig. 1, is part of a System-On-Chip (SoC) implementation combined with conventional multicore CPUs. The SoC is packaged along with DRAM and external connectors as a small (approximately 7 inches square) single-board computer. The integrated GPU shares hardware resources, such as DRAM, with CPU cores. The TX2 runs the Linux operating system, with additional support from closed-source binary drivers provided by NVIDIA. The TX2's low size, weight, and power (SWaP) requirements and low price tag make it a good exemplar of GPU-enabled platforms intended for embedding in autonomous systems.

Fig. 1 (left) shows the high-level architecture of the TX2. The TX2 contains a six-core heterogeneous ARMv8 CPU, 8 GB of DRAM, and an integrated Pascal GPU. The TX2's GPU consists of two *streaming multiproccessors* (*SMs*), each comprised of 128 GPU cores. The SMs together can be logically viewed as an *execution engine* (*EE*). Additionally, there is a hardware *copy engine* (*CE*) that can copy data between memory regions allocated for CPU use and those allocated for GPU use. The integrated GPU has fewer GPU cores than found in typical high-end GPUs used for graphics, gaming, and high-performance computing applications. We are interested in exploiting any potential for sharing the TX2's GPU by multiple tasks so that its computing capacity is not unnecessarily wasted.

Shown on the right in Fig. 1 is the architecture of the GTX 1070, an example of a discrete GPU. Discrete GPUs consist only of the SMs and local device memory, typically packaged on an adapter card for mounting in a PCIe expansion slot on a computer motherboard. Like all discrete GPUs, the GTX 1070 does not share memory with the host CPU, instead using the PCIe bus to copy data to and from host memory. This GPU features many more SMs than the TX2, increasing the potential benefit attainable if shared among multiple tasks. It also has two CEs, and a larger cache.

Algorithm 1 Vector Addition Pseudocode.

```
 1: kernel VECADD(A ptr to int, B: ptr to int, C: ptr to int)
      ▷ Calculate index based on built-in thread and block information
 2:     i := blockDim.x * blockIdx.x + threadIdx.x
 3:     C[i] := A[i] + B[i]
 4: end kernel

 5: procedure MAIN
      ▷ (i) Allocate GPU memory for arrays A, B, and C
 6:     cudaMalloc(d_A)
 7:     ...
      ▷ (ii) Copy data from CPU to GPU memory for arrays A and B
 8:     cudaMemcpy(d_A, h_A)
 9:     ...
      ▷ (iii) Launch the kernel
10:     vecAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C)
      ▷ (iv) Copy results from GPU to CPU array C
11:     cudaMemcpy(h_C, d_C)
      ▷ (v) Free GPU memory for arrays A, B, and C
12:     cudaFree(d_A)
13:     ...
```

2.2 Relevant CUDA Programming Fundamentals

A *CUDA program* runs as a task (process or thread) on a CPU and relies on a GPU for some part of its computational requirements.[1] The general structure of a CUDA program when it needs to interact with the GPU is as follows: **(i)** allocate memory for GPU use; **(ii)** copy input data from CPU memory to GPU memory; **(iii)** launch execution of a GPU program called a *kernel*[2] to process the data; **(iv)** copy the results from the GPU memory back to the CPU memory; **(v)** free unneeded memory.

CUDA kernels are written from the perspective of a single *GPU thread*. As an example, consider the CUDA program expressed in pseudocode in Algorithm 1. It uses the kernel VECADD to add a single pair of elements per GPU thread, storing the sum in a corresponding location in an output array. Line 2 demonstrates the use of special global system-defined variables to determine the array element on which to operate. When the kernel executes, threads will run in lock-step with each thread performing the same operation simultaneously on different data. To avoid confusion with GPU threads, we will henceforth refer to CPU threads as *CPU tasks* (or just *tasks*).

A kernel is run on the GPU as a set of thread blocks that can be executed in any order. These thread blocks, or simply *blocks*, are each comprised of a number of threads. As seen in Line 10 of Algorithm 1, the number of blocks and threads per block are programmer-specified and can be set at runtime when a kernel is launched. The GPU scheduler uses these values to assign work to the SMs. *Blocks are the schedulable entities on the GPU.* All threads in a block are always executed on the same SM, and run non-preemptively until completion. A kernel completes when all threads in all blocks have exited.

We refer to kernels and memory-copy operations collectively as *GPU operations*. GPU operations are submitted to a GPU in *CUDA streams*. Operations within a stream are executed in FIFO order. By default, the *NULL stream* is used, but users can submit operations to multiple user-defined streams.[3] Kernels from different streams can run concurrently by

[1] Note that both CPU and GPU computations are specified in the same CUDA program.
[2] Unfortunate terminology, not to be confused with an OS kernel.
[3] CUDA documentation only guarantees that operations within a stream are executed in FIFO order, but

sharing the GPU's cores if sufficient internal resources are available. Copy operations are handled by the GPU's CE and can be concurrent with kernel executions on the EE.

CUDA API calls can be synchronous or asynchronous; for many calls, a variant of both is available. For example, `cudaMemcpy` and `cudaMemcpyAsync` both copy data between regions of CPU memory and GPU memory, or between two regions of GPU memory, but `cudaMemcpyAsync` can return control to the calling CPU task before the copy is completed, whereas `cudaMemcpy` blocks the CPU task until the memory copy completes.

Kernel launches are always supposed to be asynchronous. The CUDA documentation[4] [23], however, uses a narrow definition of "asynchronous" that can be misleading. According to the documentation, "asynchronous library functions that return control to the host thread before the device completes the requested task." Notably, this definition does not imply that asynchronous API calls are *nonblocking* to the CPU. As noted in Sec. 3, we have found situations in which kernel launches still cause CPU blocking even if the API call returns before the requested kernel completes.

3 Synchronization and Blocking

CPU scheduling has been studied and well-understood for decades; in particular, real-time scheduling analysis of task systems is based on predictable scheduler and task behaviors. A worst-case execution time (WCET) for each task can be determined using clear specifications of the machine's architecture including the cache, bus, and DRAM operations. Incorporating GPUs into real-time analysis (as with all coprocessors), requires different models with new sets of issues to be considered. In this section, we discuss one set of issues that lead to a surprising number of pitfalls when CUDA GPUs are used: *synchronization*.

In prior work, we investigated the scheduling rules for kernels and copy operations in CUDA programs [1]. However, this investigation focused on a limited context where few CUDA operations beyond kernel launches and memory copies were used. In most real-world CUDA software, programmers will likely encounter (both intentionally and unintentionally) the need for synchronization between CPU and GPU operations. The added complexity of synchronization can result in utilization loss, potentially leading to unbounded response times in task sets with high utilization. In this section, we explore various forms of CPU-GPU synchronization and the resulting implications for real-time systems. We limit attention for now to CPU tasks that share a single Linux address space and create user-defined streams. As covered in detail in Sec. 4, this setup allows potential concurrency among operations on the GPU.

3.1 Overview of GPU Synchronization

Most developers are familiar with the concepts of synchronization in a CPU-only context where two or more tasks must communicate or coordinate their actions. Synchronization becomes more complicated when a CPU task must coordinate with programs executed on the GPU. The common case is that the CPU task must determine when data in GPU memory is safe to access (*e.g.*, copy back to CPU memory). This is accomplished using *GPU synchronization*, where the GPU must complete outstanding work and reach a *synchronization*

does not describe how operations from different streams are ordered.

[4] Specifically, Section 3.2.5.1 of the Programming Guide for CUDA version 9.1.85.

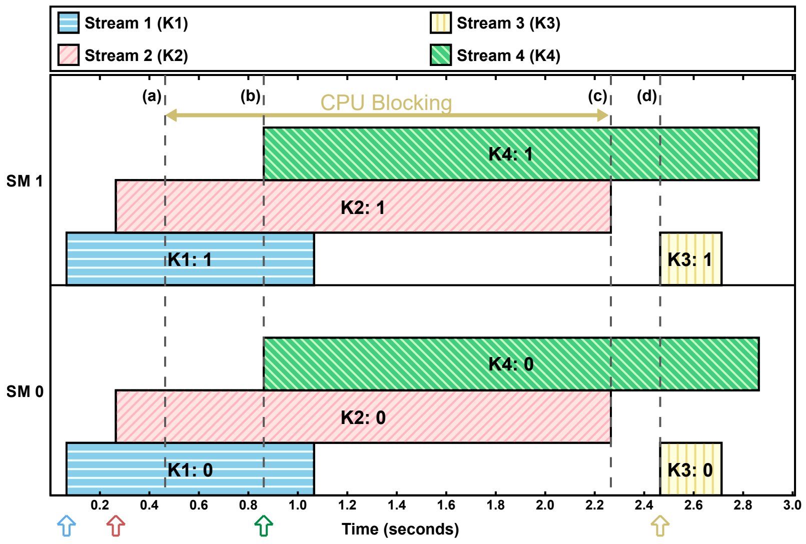


Figure 2 Explicit synchronization requested before K3, observed on the Jetson TX2.

point: a point in time when data access can safely occur. There are also other, less common, cases when GPU synchronization is necessary.

In CUDA there are multiple ways to achieve GPU synchronization. They fall into two broad categories: *explicit synchronization*, which is always programmer-requested, and *implicit synchronization*, which can occur as a side effect of CUDA API functions intended for purposes other than synchronization. We have uncovered in our research some unfortunate pitfalls relating to actual GPU synchronization behavior, especially with respect to *blocking*. So, while these may not be pitfalls for non-safety-critical applications, ignoring the effects of certain specific mechanisms for achieving synchronization would be perilous in a safety-critical system where blocking must be anticipated and accounted for in analysis.

3.1.1 Explicit Synchronization

Explicit synchronization refers to synchronization points that the CUDA programmer explicitly requests using the CUDA API. Explicit synchronization is typically used after a program has launched one or more asynchronous CUDA kernels or memory-transfer operations and must wait for computations to complete. In contrast to implicit synchronization, the sole purpose of explicit-synchronization functions is to block the calling CPU task until the GPU reaches a synchronization point.

The CUDA documentation[5] states that explicit synchronization will block the calling task until "all preceding commands" have completed. For example, if the API function `cudaDeviceSynchronize` is invoked, "preceding commands" may encompass all commands issued to the device from all CPU tasks. Other explicit-synchronization options, including `cudaStreamSynchronize`, will only block until preceding commands from a specified stream have completed.

We carried out experiments using our open-source framework[6] to investigate the specific behaviors of GPU synchronization on real GPU hardware. Fig. 2 shows the behavior of

[5] Section 3.2.5.5.3 of the Programming Guide for CUDA version 9.1.85.
[6] Available at `https://github.com/yalue/cuda_scheduling_examiner_mirror`.

explicit synchronization observed in one such experiment. In Fig. 2 (also in Figs. 3 and 4), each shaded rectangle corresponds to a separate thread block. The left and right endpoints of each rectangle correspond to the times at which the block started and completed execution, as measured on the GPU. Each rectangle's height represents its size in CUDA threads. Additionally, the vertical axis is subdivided by SM. The particular experiments presented in Figs. 2-4 were performed using the Jetson TX2, which features two SMs. Up to 2,048 CUDA threads can be assigned to a single SM at once.

The CUDA program executed to produce Fig. 2 consists of four CPU tasks all sharing a single address space. Each CPU task launched one kernel in a separate user-defined stream. Kernel launches were separated by a small amount of time. Each kernel consisted of two blocks of 512 threads, and the figure shows that one block from each kernel was scheduled on each SM. Each thread performed a busy-loop for a set amount of time.

An explicit-synchronization command, `cudaDeviceSynchronize`, was issued at time **(a)** by the CPU task responsible for launching kernel K3. This caused K3's CPU task to be blocked until the prior commands, the execution of kernels K1 and K2, had both completed at time **(c)**. This behavior is exactly what one would expect, given the description of explicit synchronization from official documentation. However, our experiments also uncovered Pitfall 1 for the unwary:

▶ **Pitfall 1.** *Explicit synchronization does not block future commands issued by other tasks.*

The fact that the launch of K4 by its CPU task was not blocked at time **(b)** is an example of this pitfall. Implicit synchronization, which we cover next, presents even more serious pitfalls.

3.1.2 Implicit Synchronization

Implicit synchronization occurs as a side effect of CUDA API calls that are otherwise unrelated to synchronization. For example, implicit GPU synchronization may occur due to freeing GPU memory or launching a kernel to the default stream. Presumably, this is because some modifications to GPU device state can only occur while no kernels are executing. The CUDA documentation about implicit synchronization[7] states that "two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

1. A page-locked host memory allocation
2. A device memory allocation
3. A device memory set
4. A memory copy between two addresses to the same device memory
5. Any CUDA command to the NULL stream"

Unlike the relatively straightforward documentation about explicit synchronization, our experiments revealed that this list includes several operations that do not necessarily cause implicit synchronization, and fails to include some functions that do. We consider this particularly problematic for real-time systems, where the ability to accurately model blocking is critical.

▶ **Pitfall 2.** *Documented sources of implicit synchronization may not occur.*

[7] Section 3.2.5.5.4 of the Programming Guide for CUDA version 9.1.85.

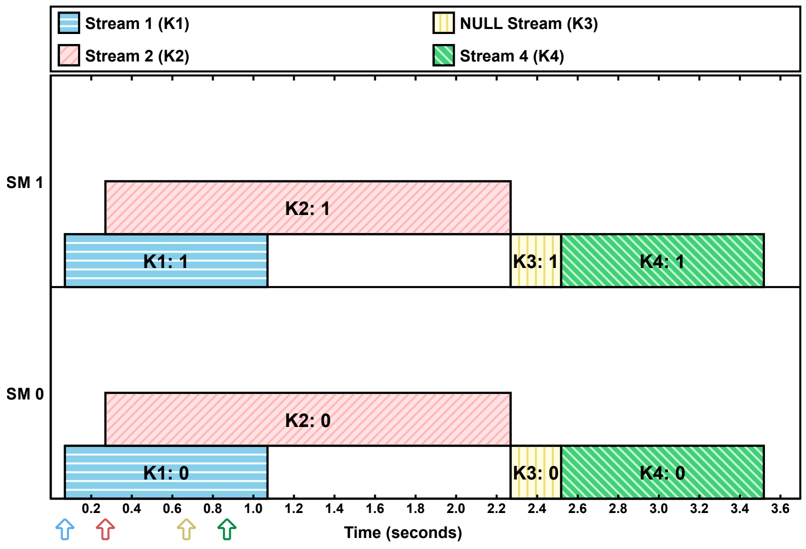


Figure 3 Implicit synchronization caused by launching kernel K3 in the NULL stream.

Pitfall 2 became apparent to us when, in all of our experiments, we never observed implicit synchronization as a result of a device-memory operation (allocation, set, or copy) or a page-locked host memory allocation. Our experiments covered the two most recent CUDA versions, 8.0 and 9.0, and the three most recent NVIDIA GPU architectures, Maxwell, Pascal, and Volta. This, of course, does not prove that implicit synchronization can *never* happen under such circumstances, but it does indicate that the documentation's statement that "two commands cannot run concurrently" is not a reliable rule. The only case (from this list) in which we did observe implicit synchronization was launching GPU operations in the NULL stream.

Fig. 3 shows a similar scenario to the one in Fig. 2, with one key difference: the CPU task for K3 did not call `cudaDeviceSynchronize` before K3 was launched, but instead launched K3 in the NULL stream. The implicit synchronization, and resulting loss of concurrency, is clearly visible in the figure. Execution of K3 must wait for the first two kernels to complete, and, in contrast to explicit synchronization, K4 is also prevented from running concurrently. Even though this loss of concurrency may be striking, it is notably explicitly documented, and can be used (or avoided) in a careful design for a real-time task.

We found, however, a different source of implicit synchronization that is a far more problematic pitfall, and is not even listed in the documentation on synchronization: *freeing device memory.*

▶ **Pitfall 3.** *The CUDA documentation neglects to list some functions that cause implicit synchronization.*

▶ **Pitfall 4.** *Some CUDA API functions will block future, unrelated, CUDA tasks* ***on the CPU****.*

Fig. 4 shows the results of an experiment identical to the one in Fig. 2, but this time the call to `cudaDeviceSynchronize` at time **(a)** was replaced with a call to `cudaFree`, which was used to de-allocate memory on the GPU. Pitfalls 3 and 4 can be observed in this plot. The fact that this blocked the calling CPU thread until all prior GPU work had completed at time **(c)** indicates that `cudaFree` created implicit synchronization. Similar to the NULL-stream behavior, implicit synchronization also prevented subsequent kernels from starting to

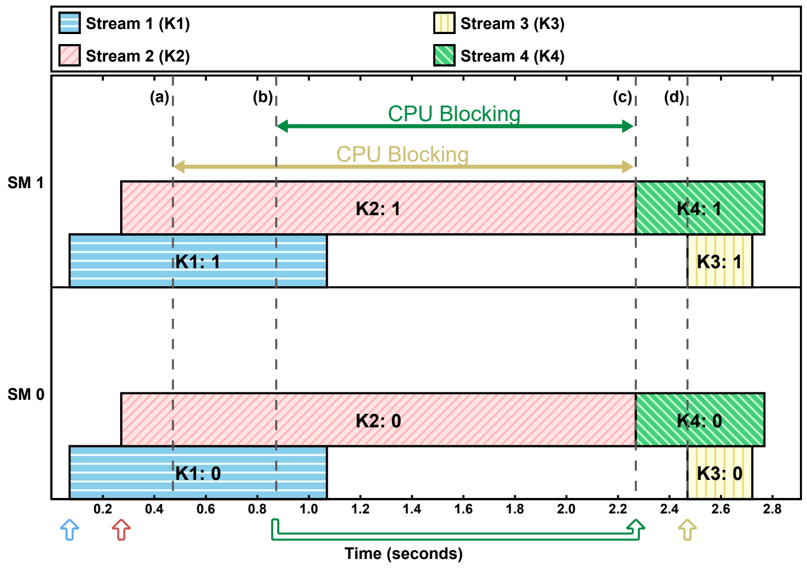


Figure 4 Implicit synchronization causing additional CPU blocking due to `cudaFree`.

execute until `cudaFree` completed at time **(c)**. We speculate that this behavior by `cudaFree` is necessary because alterations to memory-mapping state requires a quiescent execution environment. However, the most surprising effect was not that K4 was blocked, but that K4's task was blocked *on the CPU* until time **(c)**, even though it issued an "asynchronous" kernel launch. This reveals a pitfall that can harm real-time analysis that does not consider the fact that CPU tasks can experience blocking from GPU operations that are launched from unrelated tasks.

3.2 Overcoming Synchronization-Related Pitfalls

GPU synchronization has two problematic effects – introducing indeterminate amounts of blocking and reducing GPU concurrency. This means that programmers who develop real-time systems must understand the pitfalls inherent in explicit and implicit synchronization. This is especially true if the schedulability of a real-time task system relies on minimizing blocking or high GPU utilization. Avoiding pitfalls can be accomplished through careful construction of CUDA programs to, for example, avoid using the NULL stream or freeing memory outside of certain time intervals. A more robust method would be to adopt middleware that handles such problems transparently.

Our experiments indicate that GPU synchronization does not extend across GPU-using tasks that are isolated in separate address spaces. If synchronization is the dominant limiting factor on schedulability, it may be desirable to place each task in a separate address space (OS process). As explained in the next section, this organization means that that CUDA kernels from different tasks can no longer execute concurrently, but it may still be beneficial overall if synchronization-related blocking is a greater limiting factor.

It turns out that NVIDIA may be aware of this issue. Even though it is not currently available for embedded platforms such as the TX2, NVIDIA does provide useful middleware for discrete GPUs: the *CUDA Multi-Process Service* (MPS). MPS allows kernels from multiple processes to execute concurrently on a single GPU, while maintaining the desirable property that GPU synchronization from one process will not affect other processes. We explore the benefits of MPS further in Sec. 4.

4 Concurrency and Performance

In prior work, we investigated different GPU scheduling behavior when running GPU-using real-time task systems in two contexts: **(i)** each task has its own distinct address space, *i.e.*, it runs as an OS process, and **(ii)** all tasks belong to the same address space, *i.e.*, each task runs as a schedulable thread within a process. We refer to these two contexts as *process-based* and *thread-based* tasks, respectively.

While process-based tasks have the advantage of memory protection, they do not actually execute on the GPU concurrently; instead, GPU operations are multiprogrammed in a way that makes predictable scheduling of GPU-related resources difficult if not impossible to achieve [1]. When operations are multiprogrammed on a GPU, their execution times depend on contention for shared GPU resources, making it hard to bound a task's overall execution time. Additionally, concurrency among GPU operations may be important in order to avoid wasting GPU processing cycles, especially when a single kernel cannot fully utilize the GPU's resources. Although this may be avoided by running tasks with user-defined streams in a shared address space, a shared address space may actually *reduce* concurrency in task systems where tasks regularly interfere with each other via implicit synchronization (Sec. 3). Fortunately, NVIDIA provides a third option: middleware called the *Multi-Process Service* (*MPS*) [20].

4.1 Multi-Process Service (MPS)

MPS enables concurrent execution of GPU operations launched by independent CPU address spaces. It has the potential to combine the advantages of both thread- and process-based tasks. Programs written using the CUDA API require no changes to use MPS – if MPS is running, CUDA programs transparently issue requests to MPS rather than directly to a GPU. Official documentation reports that MPS operates as a server process with its own CUDA context, and that CUDA API requests are redirected from client processes to the MPS server. Because the server's CUDA context is effectively shared, GPU operations launched by separate processes can execute concurrently on a shared GPU, providing the benefits of thread-based tasks. However, MPS also continues to preserve the advantage of process-based tasks: separate processes will not block each other with implicit or explicit synchronization.

It is not clear from available documentation how MPS actually schedules GPU operations and whether the GPU scheduling rules revealed in prior work [1] are followed under MPS. For example, the documentation for MPS only mentions possible overlap between kernels and copy operations.[8] Given the documentation flaws discussed in Secs. 3 and 5.2, one could be skeptical of the veracity of this claim, so we verified experimentally that those scheduling rules are also followed under MPS. We omit from this paper the experimental methods used for verifying the scheduling rules; readers can refer to [1].

Maximizing the utilization of GPU resources using streams in thread-based tasks is suggested by NVIDIA's "Best Practices Guide" [22]. However, it would be unwise to simply take this recommendation at face value when choosing between MPS or a process- or thread-based task organization in a safety-critical system. Additionally, **MPS is not yet supported on embedded ARM platforms** like the Jetson TX2, so the other management systems are still necessary on some systems. Therefore, we conducted a case study on computer-vision software, demonstrating the performance differences among the available configurations.

[8] "MPS allows kernel and memory copy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times" [20].

Table 1 Abbreviations used for our four experimental scenarios.

	Multiple Process-based Tasks	Multiple Thread-based Tasks
Without MPS	MP	MT
With MPS	MP(MPS)	MT(MPS)

4.2 Case Study of Computer-Vision Tasks

Our motivation primarily remains autonomous driving, so we chose to study algorithms for computer-vision tasks that provide functions commonly used for autonomous driving. In evaluating the results from this case study, we consider that the real-time tasks that use GPUs for autonomous driving may have multiple levels of criticality. Some may be safety-critical with hard deadlines and be provisioned for worst-case execution plus a margin for safety. Others may have only bounded tardiness requirements, or even be background work that can be provisioned for average-case execution.

We focus here on five programs from NVIDIA's provided sample code for VisionWorks:

- **Video Stabilization.** Smooths shaky video content. This is often a preprocessing step for a computer-vision pipeline.
- **Feature Tracking.** Tracks features between consecutive frames. This algorithm is used to track the positions of objects in a scene.
- **Motion Estimation.** Estimates the direction of moving pixels, which is fundamental to calculating trajectories of moving objects, *e.g.*, pedestrians and other vehicles.
- **Hough Transform.** (*Hough*) A feature-extraction algorithm; the provided sample detects circles and lines in images.
- **Stereo Matching.** Uses input from two cameras to generate depth information by matching features in both frames.

Methodology. We adapted NVIDIA's VisionWorks samples to be compatible with our open-source experimental framework.[9] These samples generally only use a single CUDA stream. We ran four instances of the same sample program in each experiment. We configured each instance to process 1,000 frames from a video sequence while recording per-frame response times. Our framework allows running each program instance in a shared address space (multiple thread-based tasks, MT) or in independent address spaces (multiple process-based tasks, MP), both with and without the MPS server active. This produces experiments for each algorithm in four different scenarios as summarized in Tbl. 1. Experimental results under MT(MPS) were always similar to MT with slight overheads caused by MPS, so we omit it in all of our results for clarity. We conducted these experiments on a Maxwell-architecture discrete GPU with CUDA 9.0. We briefly summarize results on other devices and different CUDA versions later.

Results. We show cumulative distribution function (CDF) and kernel density estimation (KDE)[10] plots of Hough and feature tracker as representatives in Figs. 5–8. The KDE curve was produced using the `Python` package `scipy.stats.gaussian_kde`. In both the CDF and KDE plots, each curve represents the recorded response-time data in an experimental scenario.

[9] Again, `https://github.com/yalue/cuda_scheduling_examiner_mirror`.
[10] KDE is a statistical method for estimating a continuous probability density function (PDF) from a set of discrete sample values.

Table 2 Per-frame response time data (in milliseconds) of VisionWorks samples. The fastest scenario for each time metric is indicated by bold text.

VisionWorks Samples	Scenarios	Max	99^{th}%	90^{th}%	Mean	Median
Video Stabilization	MP	17.55	12.88	5.43	3.31	2.69
	MP (MPS)	36.73	**11.12**	**5.37**	**2.81**	**2.06**
	MT	**17.0**	13.87	8.94	4.72	3.63
Feature Tracking	MP	**5.64**	**3.87**	**1.45**	**1.08**	**0.96**
	MP (MPS)	14.73	6.04	1.51	1.31	1.09
	MT	31.11	20.86	11.51	4.68	2.68
Motion Estimation	MP	**28.64**	**21.25**	17.33	16.75	17.24
	MP (MPS)	33.05	22.66	**15.75**	**14.3**	**14.89**
	MT	42.86	26.12	16.53	15.07	15.14
Hough Transform	MP	**13.56**	**11.61**	7.28	5.68	5.7
	MP (MPS)	18.35	11.66	**6.44**	**3.74**	**3.18**
	MT	58.65	22.64	15.82	9.12	8.94
Stereo Matching	MP	75.13	50.54	30.42	24.14	24.77
	MP (MPS)	**59.73**	**45.05**	**26.87**	22.59	24.41
	MT	125.96	58.82	34.36	**20.75**	**18.95**

For example, the curves labeled "x4 MP" in Figs. 5 and 6 represent the per-frame response time distributions where each of four Hough instances is run in a separate process. Result data for all five algorithms is summarized in Tbl. 2, which lists the maximum, 99^{th}-percentile, 90^{th}-percentile, mean, and median frame times for each scenario and algorithm.

▶ **Observation 1.** *MP(MPS) exhibits good average-case performance.*

Obs. 1 is supported by the data in Tbl. 2. 90^{th}-percentile, mean, and median performance under configuration MP(MPS) were consistently good with the top performance for three of the five algorithms. For Feature Tracking, MP was best in all metrics, and for Stereo Matching, MT had better mean and median performance. The results for average-case performance indicate that using MP(MPS) would likely be an attractive option for soft-real-time systems, *e.g.*, systems that can occasionally drop a video frame without compromising safety. We conjecture that the average-case performance advantage of MP(MPS) over MP in most cases is due to improved concurrency and lower GPU context-switching overheads.

Feature Tracking was the most notable exception to Obs. 1. In this case, MP was only slightly better than MP(MPS) when comparing the 90^{th}-percentile, mean, and median performance. We conducted additional experiments using NVIDIA's CUDA-profiling tool, `nvprof`, to gain some insight into this behavior. We found that Feature Tracking's overall execution time is heavily influenced by a large number of memory transfers, rather than CUDA kernel executions. This likely means that MPS only provides limited GPU concurrency benefits to Feature Tracking, which failed to outweigh other MPS-related overheads.

▶ **Observation 2.** *Worst-case and 99^{th}-percentile runtimes were typically better under MP.*

While MP(MPS) largely resulted in average-case improvements, Tbl. 2 shows three of our five applications (Feature Tracking, Motion Estimation, and Hough Transform) showed the smallest worst-case and 99^{th}-percentile execution times under MP. This indicates that MP may be a better option for certain task systems where worst-case performance is more

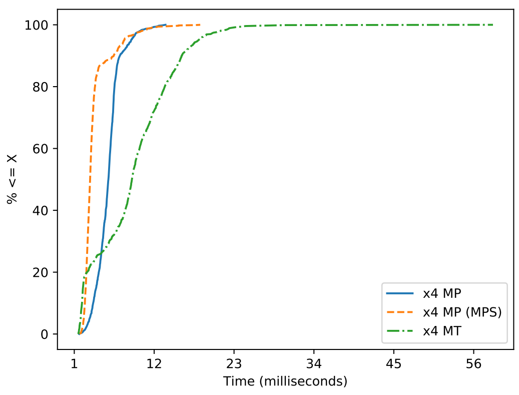


Figure 5 Per-frame response time CDFs for Hough.

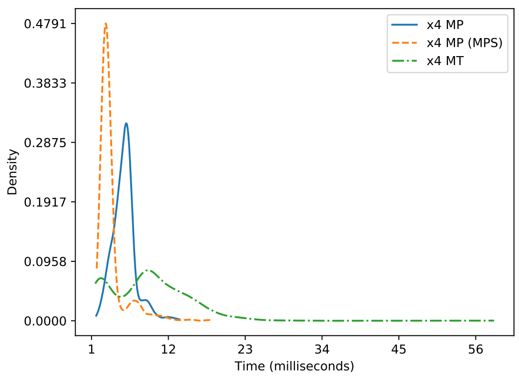


Figure 6 Per-frame response time KDEs for Hough.

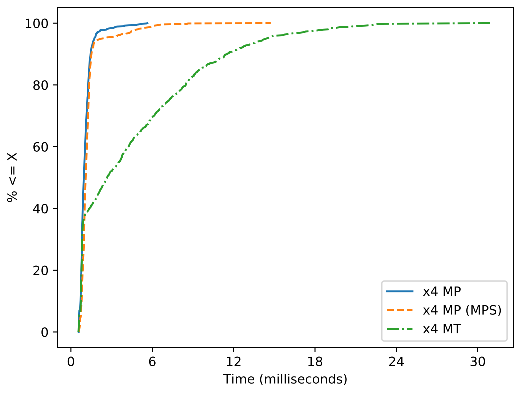


Figure 7 Per-frame response time CDFs for Feature Tracking.

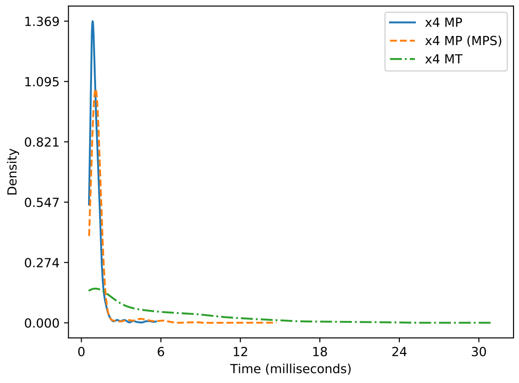


Figure 8 Per-frame response time KDEs for Feature Tracking.

important than average-case. Our results illustrate why the trade-offs between process-based and thread-based designs for tasks must be evaluated for individual algorithms.

▶ **Observation 3.** *MP and MP(MPS) exhibit more predictable execution times than MT.*

Obs. 3 is supported by Figs. 6 and 8, where the KDE shows a tight unimodal distribution for MP and MP(MPS) but not MT. A unimodal distribution function with little dispersion indicates that the response times exhibit low variance. MT, in contrast, shows both bimodal (in Fig. 6) and unimodal (in Fig. 8) distributions with significant dispersions (indicating high variance). Even if specific "spikes" are more difficult to observe in the corresponding CDF plots, the difference in response-time ranges are also apparent from the endpoints of the CDF curves in Figs. 5 and 7.

▶ **Observation 4.** *The MT configuration generally performed poorly.*

Obs. 4 is supported by Tbl. 2 and the plots. The only metrics where MT outperformed the other scenarios were the mean and median times for Stereo Matching, and worst-case response time for Video Stabilization (where MT was only slightly better than MP).

Other Results. In addition to the results presented above, we also conducted this case study using CUDA 8.0 on a Maxwell discrete GPU (GTX 860M) and CUDA 9.0 on Pascal discrete

Listing 1 Causes implicit synchronization.

```
if (!CheckCUDAError(cudaMemsetAsync(
  state->device_block_smids, 0,
  data_size))) {
  return 0;
}
```

GPUs (GTX 1050 and GTX 1070). Even though we chose to omit tables of results from the other GPUs and CUDA versions in this paper, we made similar observations excepting that the performance of all configurations was better on a Pascal GPU. Additionally, the experimental results with CUDA 8.0 on the same Maxwell GPU stayed nearly identical to those using CUDA 9.0.

Summary. Our case study compared the impact of different GPU-sharing approaches on the performance of computer-vision algorithms. The results we obtained for these algorithms ran contrary to some of our observations regarding GPU concurrency from prior work [1, 26].

▶ **Pitfall 5.** *The suggestion from NVIDIA's documentation to exploit concurrency through user-defined streams may be of limited use for improving performance in thread-based tasks.*

We assumed that enabling concurrent GPU execution was of significant importance for limiting capacity loss in real-time workloads on embedded systems, and therefore fell victim to Pitfall 5. Instead, our results show that MT rarely outperforms tasks running as multiple processes, even without MPS. Additionally, any performance improvement via fine-tuned stream organization for MT can also be achieved with MP(MPS). That being said, even though enabling concurrency using MP(MPS) is generally beneficial, it unfortunately is not an option on ARM-based embedded platforms like the Jetson TX2. We would encourage NVIDIA to consider this shortcoming in hope that one day it may be addressed.

5 Perils of CUDA Programming for Real-Time Tasks

In the previous sections we presented several specific pitfalls in correctly designing and running CUDA programs for real-time tasks. Elements of both CUDA's design and documentation contribute to this ensemble of perils to avoid. In this section, we discuss some of the broader categories of pitfalls.

5.1 Synchronous Defaults

As hinted in Sec. 3, one of the primary pitfalls when designing a real-time task system that uses a GPU is that *all possible* blocking must be accounted for in analysis. Therefore, reducing the amount of blocking on both the CPU and GPU is essential. On the GPU, this requires issuing all CUDA operations to user-defined (non-NULL) streams, and carefully controlling the use of other API functions, like `cudaFree`, that cause blocking via implicit synchronization.

Even though it may seem like an easy task for a programmer to just specify a user-defined stream as opposed to the NULL stream, we note that simple mistakes in doing so may be easy to miss. This is particularly true when using the `Async` versions of CUDA API functions, such as `cudaMemsetAsync`. For example, consider the code snippets in Listings 1 and 2, which present a particular example of Pitfall 6 below.

Listing 2 Correctly asynchronous.

```
if (!CheckCUDAError(cudaMemsetAsync(
  state->device_block_smids, 0,
  data_size, state->stream))) {
  return 0;
}
```

▶ **Pitfall 6.** *Async CUDA functions use the GPU-synchronous NULL stream by default.*

Listing 1's call to `cudaMemsetAsync` is missing a final argument specifying a user-defined stream, which causes the NULL stream to be used by default. As pointed out in Sec. 3.1.2, NULL-stream usage causes implicit synchronization and hence blocking. This mistake is corrected in Listing 2. This specific mistake actually led to *months* of mystifyingly inconsistent results in our own experiments – despite our relatively deep experience examining the subtleties of CUDA behavior (note that these code snippets are parts of much larger listings). *Would an ML application developer catch such a mistake or appreciate its impact?* Note that NVIDIA's CUDA compiler does not catch this mistake because the compiler is based on the C++ programming language, which allows default arguments to functions.

Even though the examples in Listings 1 and 2 only use `cudaMemsetAsync`, Pitfall 6 applies to other CUDA API functions as well, such as `cudaMemcpyAsync`. The fact that the CUDA documentation indicates that these functions cause implicit synchronization, as discussed in Sec. 3 and Sec. 5.2, makes potential programmer errors even harder to notice in cases where synchronization is due to NULL-stream usage rather than memory operations.

To summarize this discussion, CUDA provides a brittle programming environment: difficult-to-spot mistakes can have profound consequences for real-time tasks.

5.2 Flawed Documentation

Another substantial danger stems from the inaccurate official documentation provided by NVIDIA. While function signatures and data structures seem to receive accurate (but often sparse) official documentation, scheduling and synchronization remain under-discussed. Our group's past work includes demystifying some scheduling rules [1]. In our work to demystify implicit synchronization (see definition in Sec. 3.1.2), however, we came across not only missing documentation, but incorrect documentation.

▶ **Pitfall 7.** *Observed CUDA behavior often diverges from what the documentation states or implies.*

Consider Tbl. 3. In all but one of the cases we investigated, the documentation claims implicit synchronization will occur when it does not. While this absence of synchronization may positively benefit performance, it also may cause incorrect timing analysis. Furthermore, program logic may be broken in the (albeit unlikely) case that the program relies on a function like `cudaMemsetAsync` to trigger GPU synchronization.

Unfortunately, the documentation also contains less-benign flaws. Take `cudaFree` and `cudaFreeHost` as an example. Our experiments in Sec. 3 found these functions to not only cause implicit synchronization, but block other CPU tasks from proceeding while `cudaFree` waits on the GPU. Much to our surprise, the documentation mentions neither of these side effects, leaving the reader to assume that these functions behave similarly to other CUDA functions and have no side effects.

Table 3 Observed vs. documented synchronization sources in CUDA. For `cudaMemcpyAsync` we distinguish the direction of copy between device and host: (D-D) internal to GPU memory; (D-H) GPU memory to CPU memory; (H-D) CPU memory to GPU memory. *The documentation is contradictory for these instances, but the more detailed option indicates that these functions only cause synchronization if host memory is not page-locked. We were unable to observe this regardless of whether host memory was page-locked or not.

	Observed Behavior			Documented Behavior	
Source	Blocks Other CPU Tasks	Implicit Sync. (Sec. 3.1.2)	Caller Must Wait for GPU	Implicit Sync. (Sec. 3.1.2)	Caller Must Wait for GPU
`cudaDeviceSynchronize`	No	No	Yes	No	Yes
`cudaFree`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaFreeHost`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaMalloc`	?	No	No	**Yes**	No (impl.)
`cudaMallocHost`	?	No	No	**Yes**	No (impl.)
`cudaMemcpyAsync D-D`	No	No	No	**Yes**	No
`cudaMemcpyAsync D-H`	No	No	No	**Yes***	No
`cudaMemcpyAsync H-D`	No	No	No	**Yes***	No
`cudaMemset (sync.)`	No	Yes	No	Yes	No
`cudaMemsetAsync`	No	No	No	**Yes**	No
`cudaStreamSynchronize`	No	No	Yes	No	Yes

Our experiments also revealed that `cudaMalloc` and `cudaMallocHost` may also cause cross-task CPU blocking in a similar manner to `cudaFree` in certain situations, even though these functions do not trigger implicit synchronization. As we have not yet determined the specific causes for this behavior, this property is indicated by an entry of '?' in certain cells in Tbl. 3. In any case, we failed to find any mention of this variant of CPU blocking in the CUDA documentation, and investigating these functions remains an open topic that we plan to explore in future work.

An especially worrying pitfall is the following:

▶ **Pitfall 8.** *CUDA documentation can be contradictory.*

In one case, namely `cudaMemcpyAsync`, we discovered that the CUDA documentation actively contradicts itself. Section 3.2.5.1 of the CUDA Programming Guide states "The following device operations are asynchronous with respect to the host: ... Memory copies performed by functions that are suffixed with `Async`," but Section 2 of the CUDA Runtime API documentation states "For transfers from device memory to pageable host memory, [`cudaMemcpyAsync`] will return only once the copy has completed." This raises further doubts about the correctness of other parts of the CUDA documentation.

We note that the CUDA API contains 146 non-deprecated or compatibility-related functions, and we have only tested a small fraction of these in depth. Therefore, it is likely that our findings with Pitfalls 7 and 8 apply to other portions of the documentation that we have yet to observe.

5.3 Unknown Future

All of the pitfalls discussed in this paper, as well as the need to compare the alternatives considered in Sec. 4 empirically, can be attributed to a single overarching problem: the black-box nature of current GPU-enabled platforms means that *developers do not have a reliable model of GPU behavior.* Much of our group's prior work has focused on developing such a model. However, this highlights what is perhaps the most important pitfall:

▶ **Pitfall 9.** *What we learn about current black-box GPUs may not apply in the future.*

Despite the fact that we validated our experimental results on several of the most recent CUDA versions and GPU architectures, there is no guarantee that our results will hold after future GPU-architecture or CUDA-version updates. This applies not only to rules about scheduling or blocking, but also may apply to performance characteristics like memory-access times, as we found in prior work [26].

Even though other safety-critical hardware inevitably undergoes changes and updates, future-proof programs can still be developed against a stable specification. Likewise, the only way to truly mitigate Pitfall 9 is for GPU manufacturers to release stable, accurate documentation about their GPU platforms, along, preferably, with giving developers greater control over GPU scheduling and synchronization. Only then we can have a reliable GPU model upon which to base real-time analysis and certification. We hope that work such as ours signals to manufacturers like NVIDIA that greater openness is a desirable feature when marketing in safety-critical domains.

Unfortunately, there is little indication that NVIDIA plans to move towards open hardware or software in the immediate future. In the meantime, one of our continuing objectives is to produce tools, such as our experimental framework, that can be quickly adapted to new GPU hardware. So far, our tools have allowed us to quickly re-validate our prior results every time NVIDIA updates its black-box hardware or software.

6 Conclusion

Vehicles on the road today are already running highly complex GPU-accelerated applications. We anticipate a future where safety-critical autonomous vehicles must be certified, but this will require a change in the GPU-programming paradigm. Currently, computer-vision applications are developed with little guidance about how to achieve temporal safety. Even if a single programmer or application avoids some mistakes, it is increasingly difficult to avoid all of them, especially as applications and task systems grow in complexity. This necessitates work such as ours, which seeks to reduce the gap between computer-vision application developers and those responsible for certifying new systems' real-time safety.

With little openness in NVIDIA's hardware and software ecosystem, this paper contributes a list of potential pitfalls when developing CUDA applications for real-time systems. Reasons for these pitfalls include GPU synchronization, application performance, and problems with documentation. We uncovered these pitfalls via microbenchmark experiments, examining the performance of real-world computer-vision applications, and a careful reading of official GPU documentation. While there is no guarantee of stability in our observations as NVIDIA's hardware and software continues to evolve, we hope that our open-source experimental system will at least make it apparent when changes do occur.

This paper is part of an ongoing project with the aim of developing an abstract model of GPU execution. In the future, we plan to continue this investigation and eventually develop middleware capable of intercepting and reordering or delaying GPU operations. Our hope is that the control afforded by such middleware will enable us to produce reasonable analytical bounds on blocking and response times, while maintaining high GPU utilization wherever possible. However, even with better management, certifiable safety in the face of GPU sharing requires a *guarantee* that pitfalls including blocking due to GPU synchronization are controlled, which is only possible if developers of GPU-using software are aware of the consequences and how to avoid them. Fortunately, the best practices we have laid out herein should alleviate much of the strain on application developers on their first foray into real-time systems.

In addition to NVIDIA's GPU, we will also investigate other GPU implementations, *e.g.*, AMD's open-source GPU runtime and driver stack. Given the chances of modifying AMD's open-source implementation, we are interested in improving the real-time guarantees of AMD's GPUs and comparing them with NVIDIA's GPUs.

References

1 T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *RTSS 2017*, pages 104–115. IEEE Computer Society, 2017. doi:10.1109/RTSS.2017.00017.

2 J. Aumiller, S. Brandt, S. Kato, and N. Rath. Supporting low-latency CPS using GPUs and direct I/O schemes. In *RTCSA '12*, pages 437–442. IEEE Computer Society, 2012. doi:10.1109/RTCSA.2012.59.

3 C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *ECRTS '12*, pages 287–296. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.15.

4 K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *ECRTS '12*, pages 277–286. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.16.

5 K. Berezovskyi, K. Bletsas, and S. Petters. Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *ETFA '13*, pages 1–8. IEEE, 2013. doi:10.1109/ETFA.2013.6647966.

6 K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar. Measurement-based probabilistic timing analysis for graphics processor units. In *ARCS '16*, volume 9637 of *Lecture Notes in Computer Science*, pages 223–236. Springer, 2016. doi:10.1007/978-3-319-30695-7_17.

7 K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *RTNS '14*, page 279. ACM, 2014. doi:10.1145/2659787.2659827.

8 A. Betts and A. Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *ECRTS '13*, pages 193–202. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.29.

9 N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna. SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses. In *RTNS 2017*, pages 48–57. ACM, 2017. doi:10.1145/3139258.3139270.

10 R. Cavicchioli, N. Capodieci, and M. Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *RTNS 2017*, pages 1–10. IEEE, 2017. doi:10.1109/ETFA.2017.8247615.

11 G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*, pages 33–44, 2013. doi:10.1109/RTSS.2013.12.

12 B. Forsberg, A. Marongiu, and L. Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous SoC. In *DATE '17*, pages 318–321. IEEE, 2017. doi:10.23919/DATE.2017.7927008.

13 A. Horga, S. Chattopadhyayb, P. Elesa, and Z. Peng. Systematic detection of memory related performance bottlenecks in GPGPU programs. In *JSA '16*, 2016.

14 P. Houdek, M. Sojka, and Z. Hanzálek. Towards predictable execution model on ARM-based heterogeneous platforms. In *ISIE '17*, pages 1297–1302. IEEE, 2017. doi:10.1109/ISIE.2017.8001432.

15 S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *RTSS '11*, pages 57–66. IEEE Computer Society, 2011. `doi:10.1109/RTSS.2011.13`.

16 S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC '11*. USENIX Association, 2011. URL: `https://www.usenix.org/conference/usenixatc11/timegraph-gpu-scheduling-real-time-multi-tasking-environments`.

17 H. Lee and M. Abdullah Al Faruque. Run-time scheduling framework for event-driven applications on a GPU-based embedded system. In *TCAD '16*, 2016.

18 A. Li, G. van den Braak, A. Kumar, and H. Corporaal. Adaptive and transparent cache bypassing for GPUs. In *SIGHPC '15*, pages 17:1–17:12. ACM, 2015. `doi:10.1145/2807591.2807606`.

19 X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. In *TPDS '16*, 2016.

20 Multi-process service. Online at `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

21 NVIDIA. Embedded systems developer kits and modules. Online at `http://www.nvidia.com/object/embedded-systemsdev-kits-modules.html`.

22 NVIDIA. Best practices guide. Online at `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, 2017.

23 NVIDIA. CUDA toolkit documentation v9.1.85. Online at `http://docs.nvidia.com/cuda/`, 2018.

24 N. Otterness, V. Miller, M. Yang, J. Anderson, F.D. Smith, and S. Wang. GPU sharing for image processing in embedded real-time systems. In *OSPERT '16*, 2016.

25 N. Otterness, M. Yang, T. Amert, J. Anderson, and F.D. Smith. Inferring the scheduling policies of an embedded CUDA GPU. In *OSPERT '17*, 2017.

26 N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F.D. Smith, A. Berg, and S. Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *RTAS '17*, pages 353–364, 2017. `doi:10.1109/RTAS.2017.3`.

27 U. Verner, A. Mendelson, and A. Schuster. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *SYSTOR '12*, page 7. ACM, 2012. `doi:10.1145/2367589.2367596`.

28 U. Verner, A. Mendelson, and A. Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. In *ICDCN '14*, volume 8314 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2014. `doi:10.1007/978-3-642-45249-9_23`.

29 U. Verner, A. Mendelson, and A. Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *ICCCN '14*, pages 1–8. IEEE, 2014. URL: `https://doi.org/10.1109/ICCCN.2014.6911778`, `doi:10.1109/ICCCN.2014.6911778`.

30 H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS '10*, pages 235–246. IEEE Computer Society, 2010. `doi:10.1109/ISPASS.2010.5452013`.

31 Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian. Scheduling tasks with mixed timing constraints in GPU-powered real-time systems. In *ICS '16*, pages 30:1–30:13. ACM, 2016. `doi:10.1145/2925426.2926265`.

32 J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25:1522–1532, 2014.

33 H. Zhou, G. Tong, and C. Liu. GPES: A preemptive execution system for GPGPU computing. In *RTAS '15*, pages 87–97. IEEE Computer Society, 2015. `doi:10.1109/RTAS.2015.7108420`.

Instruction Caches in Static WCET Analysis of Artificially Diversified Software

Joachim Fellmuth
Technical University of Berlin, Berlin, Germany
joachim.fellmuth@tu-berlin.de

Thomas Göthel
Technical University of Berlin, Berlin, Germany
thomas.goethel@tu-berlin.de

Sabine Glesner
Technical University of Berlin, Berlin, Germany
sabine.glesner@tu-berlin.de

Abstract

Artificial Software Diversity is a well-established method to increase security of computer systems by thwarting code-reuse attacks, which is particularly beneficial in safety-critical real-time systems. However, static worst-case execution time (WCET) analysis on complex hardware involving caches only delivers sound results for single versions of the program, as it relies on absolute addresses for all instructions. To overcome this problem, we present an abstract interpretation based instruction cache analysis that provides a safe yet precise upper bound for the execution of all variants of a program. We achieve this by integrating uncertainties in the absolute and relative positioning of code fragments when updating the abstract cache state during the analysis. We demonstrate the effectiveness of our approach in an in-depth evaluation and provide an overview of the impact of different diversity techniques on the WCET estimations.

2012 ACM Subject Classification Software and its engineering → Real-time systems software

Keywords and phrases WCET, static analysis, abstract interpretation, artificial diversity, cache analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.21

1 Introduction

Cyber-Physical Systems (CPS) have an ever increasing impact on our life as more systems are controlled by computers, which are highly interconnected and even connected to the internet. Among these systems are hard real-time systems such as airbag or ABS controllers, where missing a deadline is considered a system failure. If such a functionality is safety-critical, e.g. the ignition of an airbag, the developer is required to provide guarantees on safety and timing properties. To provide timing guarantees for given system, the worst-case execution time (WCET) needs to be determined. Static WCET analyses deliver a safe upper bound of the execution time of a task. In contrast, dynamic analyses under-approximate the execution time and are thus not feasible to be used in safety-critical systems.

When safety-critical systems are exposed to potential attackers, assuring safety implies also dealing with security issues. In particular, control-flow attacks are a threat to CPS because approximately 82% of the systems are developed in unsafe languages [12]. Also, CPS are often deployed in hostile environments. Existing run-time countermeasures cannot be applied due to limited resources or limited operating system support. Recent events demonstrate that even in safety-critical applications, where strict regulations are in place, attacks are successfully mounted [28, 33].

Formal methods to statically prove the absence of vulnerabilities (e.g. Astrée [29] or CPAChecker [3]) often impose high usage effort and cost or considerable limitations in language and development possibilities, which keep them from being widely adapted in areas where it is not enforced by regulations. Also, defensive techniques that specifically target known attacks and vulnerabilities are often circumvented by new kinds of attacks.

Artificial software diversity [17, 23, 32] is an established way to enhance security in general purpose computing systems by thwarting code-reuse attacks such as return-oriented programming [6, 35]. The basic idea is to hide the memory layout from the attacker by compiling or loading semantically equivalent versions of the program with varying memory layout. Without detailed knowledge of the memory layout, it is considerably harder to mount a successful attack using existing code. Hiding the memory layout does not prevent attacks entirely, but it can lower the probability of success so that the attack becomes infeasible. Diversity also copes very well with new kinds of attack, provided the attack relies on knowledge of the memory layout, and, once introduced into the tool chain, does not require additional actions by the developer. In addition, diversity enables redundant systems, where independent replicas show the same intended behavior, but react differently to code-reuse attacks. As long as an attack on a replica does not interfere with the timing of other replicas, the system can tolerate a subset of the replicas to be compromised.

A WCET of a task in a diverse system has to be an upper bound for *all* variants of the program because the timing guarantees are only sound if they are guaranteed for any variant at any time. Existing static WCET analyses that incorporate instruction caches perform a detailed micro-architectural analysis that relies on absolute fixed instruction addresses. Using the diversification techniques we consider, the code is split into a fixed set of code parts, whose order is varied among the variants (we refer to these parts as fragments). Thereby, the absolute positions of fragments and their relative distances are unknown to the WCET analysis. So far, due to this contradiction, state-of-the-art static WCET cache analysis is not applicable to diverse systems, as it cannot guarantee an upper bound for all variants.

To overcome this problem, we introduce an instruction cache analysis, which is based on the abstract interpretation-based approach originally proposed by Ferdinand [15], and later improved by Ballabriga [1]. Our key idea is as follows: To ensure soundness, we assume that all instructions are possibly located in any location in a cache block, and we apply the worst-case cache behavior to all sets of blocks with unknown relative distance to the current basic block. Together, this ensures that our analysis neither relies on absolute instruction addresses nor on their relative distances, which both may be changed by diversification. To still be able to calculate tight upper bounds on the WCET, we retain all relative positioning within a fragment, we consider all possible absolute addresses of a fragment and we tightly limit the impact of cache accesses for cache contents of other fragments to the worst-case cache access.

Our approach universally supports all regular instruction cache architectures. In our experiments with small caches, the WCET estimates are tight, with an average over-approximation of 8.6%, compared to the highest WCET obtained by the non-diverse analysis applied to a number of variants. The estimates are a considerable improvement over an analysis without caches (assuming *all miss* for every memory access). Our benchmark results average at only 39.8% of the WCET without considering a cache.

The rest of this paper is structured as follows: In Section 2, we give an overview of artificial diversity techniques, and we introduce the basic concepts of current instruction cache analyses. We introduce our approach in three steps: First, in Section 3, we discuss the impact of changes in absolute and relative position of code fragments on the cache behavior and analysis. Second, in Section 4, we present our analysis approach. Third, we introduce our worst-case cache hit classification in Section 5. Section 6 contains a detailed evaluation of our approach using well-known WCET benchmark programs. Section 7 contains a discussion of related work. And, finally, in Section 8, we conclude with a discussion of our findings and future work.

2 Background

In this section, we first briefly introduce artificial diversity, which serves as a basis for the different types of diversity we use in our evaluation. Then, we introduce the WCET instruction cache analyses our work is based on in Section 2.2.

2.1 Artificial Diversity

Artificial software diversity techniques [17,23] are run-time countermeasures against control-flow attacks that are based on hiding information from the attacker by automatically creating many variants of the same program. The concept is based on the fact that the attacker needs information such as the detailed layout of parts of the memory to mount certain attacks successfully. Diversity is most useful against code-reuse attacks [4,7,34] on systems, where code injection is prevented using data execution prevention (DEP). The diversification can be introduced into every stage of the software development life cycle. Comprehensive overviews of control flow attacks and their countermeasures can be found in [35,38]. It was demonstrated (e.g., [6]) that code-reuse attacks are a threat to CPS, many of which are safety-critical real-time systems.

The most prominent example of artificial diversity is address space layout randomization (ASLR) [5,32]. In ASLR, the base addresses of some or all segments of the virtual memory of a process are randomized. ASLR is part of standard desktop operating systems such as Windows and Linux.

In addition to the segment-level diversity of ASLR, many other variations have been proposed, such as the substitution of instructions or small sequences with equivalent ones, garbage code insertion, function and function variable reordering, basic block level code shuffling, instruction-level diversity [23]. In earlier work [14], we have proposed a way to apply block-level diversity to safety-critical real-time systems.

In this paper, we concentrate on diversity techniques whose transformations are limited to relocating and reordering fragments of the code without changes in the control flow, code size, and instructions. These can be applied to the entire instruction memory, and enable us to precisely predict the WCET of all tasks of the executable. More specifically, we support the following kinds of diversity:

- **Segment-level diversity**: Similarly to ASLR, the entire text segment is relocated to a random position in memory, assuming a (virtual) memory space that is considerably larger than the program. In contrast to ASLR, segment-level diversity does not have to be aligned to memory pages. The segment (only one fragment segment-level diversity) can be located at any address, which includes addresses that are mapped to any offset in a cache line.
- **Function-level diversity** [22]: Just as the compiler is free to choose the order of functions and global data in the final executable, a variant can contain the functions in random order without any semantic difference to other variants. This enables a much larger number of possible variants than in segment-level diversity. The number of fragments equals the number of functions in the code.

- **Block-level diversity** [14]: The code is split into movable instruction sequences (MIS), which form the fragments, whose last instruction is an unconditional jump (e.g. `jmp`, `ret`). These fragments contain at least one basic block (BB) of the control flow graph (CFG).

The diversity techniques we consider can - for instruction cache analysis - be characterized by the diversity alignment and the fragmentation. We define the diversity alignment as the set of *offsets* $O = \{o_1, ..., o_K\}$ within a cache line an instruction can be located at. The number of offsets is equal to the cache line size divided by the alignment. For example, if a cache line has 16 Bytes, and the placement of code fragments is aligned to 4-Byte instructions (e.g. in ARM binaries), there are $K = 4$ different offsets a basic block can have relative to a cache line. If the alignment is greater or equal to the cache lines (e.g. 4 kByte pages in ASLR), there is only one offset $K = 1$. The number of fragments N, or, more specifically, the mapping of instructions and basic blocks to fragments depends on the diversification techniques mentioned above. We assume this information is given during our analysis using a function *frag:* $I \to F$, where I is the set of instructions the program consists of and $F = \{f_1, ..., f_N\}$ the set of fragments.

2.2 Worst-Case Execution Time Analysis

Static WCET analyses typically consist of three parts: First, control flow and data flow analyses are used to create a model of the program in form of a control flow graph, flow facts such as loop bounds, and variable assignments or ranges. In a second phase, the micro-architectural analysis determines local timings, taking into account the actual timing behavior of the processor. Finally, the execution time is maximized over all control-flow paths, usually by representing the findings of the other phases as constraints in a linear program that can be solved by a linear program solver. This technique is called *implicit path enumeration technique (IPET)* [36]. While phase one and three are independent of the actual memory layout of the program on the target machine, phase two depends on the actual hardware and on the granularity of the analysis.

Instruction caches exploit the spatial and temporal proximity of executed instructions in the memory [27]. They are constructed so that instructions that are located close to each other are not conflicting. Therefore, the behavior of caches directly depends on the absolute position of each instruction (or basic block) and on the relative distance of code fragments that are executed on the same path. Any change in the program location or the order of code fragments will directly affect the WCET analysis result.

2.3 Instruction Cache Analysis

The state-of-the-art technique to represent caches in WCET analyses is based on Ferdinand et. al. [16]. There, an abstract interpretation based [8] separate cache analysis was introduced, which classifies memory accesses before the global WCET maximization phase. In this section, we briefly introduce the non-diverse analysis. Note that, while non-diverse analyses typically map a cache state from cache to memory, we define it the other way around. This helps in defining our analysis in a more compact way in Section 4.

2.3.1 Cache Definition

Caches are small buffer memories with shorter access times than the main memory. They are used to avoid long waiting times when accessing memory locations multiple times. They are mainly defined by the following values: The line size S_L defines the number of bytes that is cached together (cache block), i.e. the size of the portion of main memory that is loaded on a cache miss. The associativity A characterizes the number of locations, into which a memory block can be loaded. The capacity S_C is the total number of bytes in the cache, with $n = \frac{S_C}{S_L}$ blocks in the cache. A set consists of all memory locations that can be loaded into a cache line, and the number of different sets equals $\frac{n}{A}$. A cache with $A = 1$ is called direct-mapped, and a cache with $A = n$ is called a fully-associative cache.

As a replacement strategy, we focus on the *least recently used (LRU)* as it is most predictable, and we leave the investigation of other replacement strategies to future work. As we are using instruction caches and we assume that no write accesses can be made, write strategies are of no importance to our analysis.

2.3.2 Concrete Cache State

The cache itself is constructed as follows: A cache set is a sequence of cache lines $s_x = \{l_{x_1}, ..., l_{x_A}\}$, with x denoting the index of the set. Note that the order of the lines does not correspond to their location in memory. Instead, the index depicts the age of the line content in the LRU replacement, with 1 being the youngest. The whole cache is the union of all sets, $C = \bigcup\{s_1, ..., s_{\frac{n}{A}}\}$. We also assume a special cache line $\{l_\perp\}$, denoting the cache line that all memory blocks map to that are not currently in cache. The main memory is defined as a sequence of cache blocks in memory $M = \{m_1, ..., m_k\}$, with $k * S_L$ as the total program size. A cache state c is a function of each cache block to a cache line:

$$c : M \to C \cup \{l_\perp\}$$

The set of all cache states is denoted $\hat{C}$.

A *concrete cache state (CCS)* fulfills the property that at most one cache block can be mapped to each cache line:

$$c(m_1) = c(m_2) \to (m_1 = m_2 \vee c(m_1) = c(m_2) = l_\perp) \tag{1}$$

We use auxiliary functions: $set(m) : M \to \mathbb{N}$ maps cache blocks to cache set indices. And $age : \hat{C} \times M \to \mathbb{N}$ delivers the age of a cache block in the cache, or ∞ if not cached.

$$age(c, m) = \begin{cases} \infty & |c(m) = l_\perp \\ i & |c(m) = l_{s_i} \end{cases}$$

We define the empty cache as a function that maps all cache blocks to $l_\perp$:

$$c_\perp(m) = l_\perp$$

An update of the cache state c at a memory reference m using the LRU replacement strategy is described by an update function: $U : \hat{C} \times M \to \hat{C}$ that updates the mapping of all cache blocks m' in a cache state c, resulting in a new cache state. The currently accessed cache block m is in the first line of its set, l_{s_1}, as it is (most recently) accessed or loaded. If it was not in cache, all cache blocks $m' \neq m$ move one cache line “down" in age (degrade from $l_{s_{a'}}$ to $l_{s_{(a'+1)}}$), with the oldest one (with $a' = A$) being evicted (degraded from their previous cache line $l_{s_{a'}}$ to $l_\perp$). If the access to m is a hit, only the cache blocks of the same

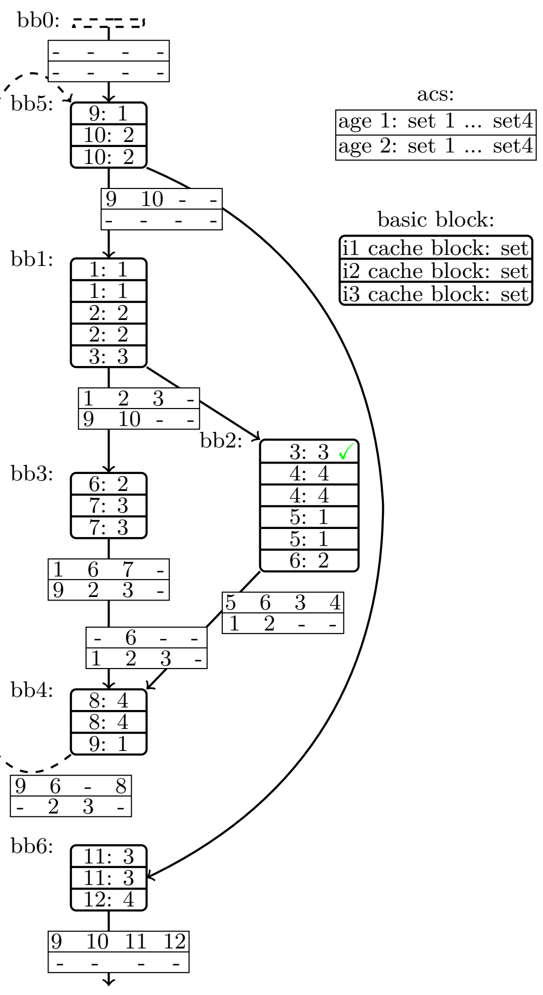


Figure 1 Must analysis example.

set, which were cached more recently ($a > a'$), are degraded. In any case, cache blocks of other sets are not affected.

$$U(c,m) = U^h(c,m,set(m),age(c,m))$$

$$U^h(c,m,s,a)(m') = \begin{cases} l_{s_1} & |m' = m \\ l_{s_{(a'+1)}} & |m' \neq m \wedge c(m') = l_{s_{a'}} \wedge a' < A \wedge a > a' \\ l_{\perp} & |m' \neq m \wedge c(m') = l_{s_{a'}} \wedge (a' = A \wedge a = \infty) \\ c(m') & |otherwise \end{cases}$$

The concrete cache state after executing a path that contains a sequence of memory references $P = \langle m_1, ..., m_y \rangle$ is given as $c_P = U(...U(U(c_{\perp}, m_1), m_2)..., m_y)$.

Note that we define the cache states for the whole cache at once, although the behavior of the different sets is independent. This makes it easier for us to explain our own analysis later on.

2.3.3 Must Analysis

In an *abstract cache state* (ACS), more than one cache block can be mapped to a cache line, i.e. property (1) may not hold. From the cache perspective, each cache line can be associated with a set of cache blocks. With these abstract cache states, cache information of all program paths leading to the basic block can be accumulated. Using the theory of abstract interpretation and data flow analysis [9], the abstract cache states of the complete control flow graph are determined using a fix point algorithm.

The *LRU must analysis* is used to identify which cache blocks are *all hit (AH)*, i.e. cache blocks which never generate a cache penalty. The must analysis creates abstract cache states, where each memory block maps to the cache line that corresponds to its oldest possible age in the cache at the given program point.

An update of a cache block m of an ACS in the must analysis U_{must} is the same as the update U of a CCS: Cached blocks of younger age are degraded, and blocks of age A are evicted if there was a miss. An important difference to the CCS is that the ACS might contain cache blocks of the same age as m. These are not degraded, because the ACS holds the oldest possible age, so those blocks are in fact either younger than m, where degradation would lead to an age not older than m's age, or older, and therefore do not need to be degraded. Given the incoming ACS, the cache behavior of the basic block is independent of the path that is currently executed. The relevant change is the introduction of the *join*-function that is used to merge two abstract cache states at program points where different paths join, i.e. at the start of basic blocks with at least two incoming edges. Here, for each memory block, the most pessimistic cache line is chosen, i.e. the line with the oldest associated age or $l_\perp$ if the memory block is not in the cache in at least one of the outgoing ACS of the basic blocks at the source of the incoming edges.

$$U_{must}(c, m) = U^h(c, m, set(m), age(c, m))$$

$$J_{must}(c_1, c_2)(m) = \begin{cases} l_\perp & |c_1(m) = l_\perp \lor c_2(m) = l_\perp \\ c_2(m) & |age(c_1, m) < age(c_2, m) \land c_1(m) \neq l_\perp \\ c_1(m) & |age(c_1, m) \geq age(c_2, m) \land c_2(m) \neq l_\perp \end{cases}$$

Figure 1 gives an example of the results of the must analysis. It contains a CFG of a short example program, and the corresponding ACS after the fix point of the analysis was reached (assuming `bb0` is the entry node and the cache is empty at start). The cache that is used here is a 2-way associative cache with four sets and the block size S_L is set so that it contains two fixed-size instructions. The boxes with rounded corners are basic blocks, and they are split horizontally into one part per instruction. The first number depicted in each instruction is the cache block it belongs to and the second is its corresponding cache set. For example, the first two instructions of `bb1` are in cache block 1, which belongs to cache set 1. Along the edges of the CFG there are the ACS. The ACS tables contain a row for each age of cache contents with the youngest on top. The cache contents are arranged by set, starting at set 1 from the left. This way conflicting cache blocks are in the same column (e.g. cache blocks 1, 5 and 9 are conflicting in set 1). When a basic block is executed, the update function U_{must} is applied to the incoming ACS, resulting in the ACS depicted at the outgoing edges. For example, in `bb3` the cache blocks 6 and 7 are accessed, which corresponds to changes in sets 2 and 3. After the update, cache blocks 6 and 7 are in the ACS at age 1. The conflicting cache blocks 2 and 3 are degraded from age 1 to age 2, and cache block 10 is evicted because it was already at age 2. The entry ACS of `bb4` shows an example application of the J_{must}: The oldest age of cache block 1 is age=2, therefore this is its resulting age. Cache block 7 is only cached after `bb3`, and not after `bb2`, therefore it does not appear in the resulting ACS. The check mark marks the memory block access that is identified as a hit by the must analysis of this example.

2.3.4 May and persistence analysis

The may analysis is used to classify cache blocks as *all miss (AM)*. It is similar to the must analysis, with the difference that it associates each memory block with the cache line of the youngest age the memory block may have. Another important analysis is the persistence analysis to classify cache blocks as *first miss (FM)* [30], causing a cache penalty exactly once in a specific scope. The first abstract interpretation based analysis by Ferdinand [16] uses a converted may analysis, whose ACS contains the oldest possible age of a cache block up to age $A + 1$. Any block that cannot have reached age $A + 1$ at a program point is persistent. This analysis does not perform well on nested loops. Ballabriga [1] proposed multi-level persistence: A persistence level is determined for each loop block m is part of. This is achieved using a stack of persistence ACS, where a new empty element is pushed on loop entry. Using this loop context information, the outermost loop in which the block is persistent can be determined.

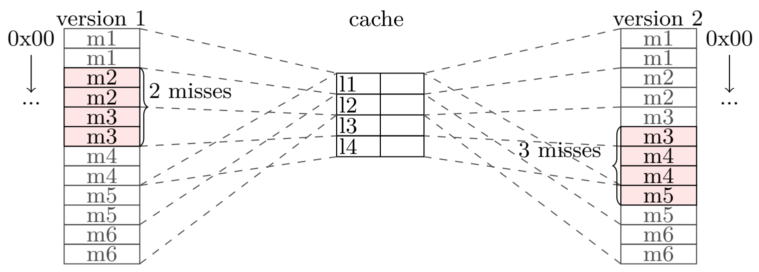
Cache blocks that cannot be identified by any of the analyses are classified as non-classified (NC), which, in a WCET analysis, is equal to AM.

3 Impact of Diversity on Caches

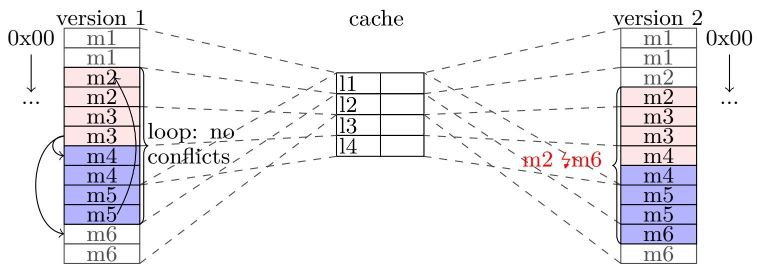
Instruction caches exploit the temporal and spatial proximity of the instructions executed alongside a path. Artificial diversity (relocating and reordering) impairs spatial proximity. Therefore, a negative impact on caching behavior and thus on the average performance can be expected. Moreover, static instruction cache analysis depends on absolute and relative addresses. Diversity worsens the predictability of the cache behavior because for every uncertainty, the worst case has to be assumed.

To clarify what the impact on the instruction cache analysis is, we discuss illustrating examples. Figure 2a and 2b show examples of the different caching behavior of the same basic blocks with different absolute positions. Figure 2a shows a basic block that in version 1 covers two cache blocks (m2, m3) and can therefore cause at worst two cache misses. In version 2, the same basic block is moved to another position where it covers 3 cache blocks (m3, m4, m5). Similar to that, Figure 2b shows a loop that just fits into the cache. Moving it by half a cache line, as in version 2, creates a conflict between the first and the last block. These examples show that the impact of absolute address changes of a basic block corresponds to different offsets of its instructions in the cache line. This is the case for every cache associativity.

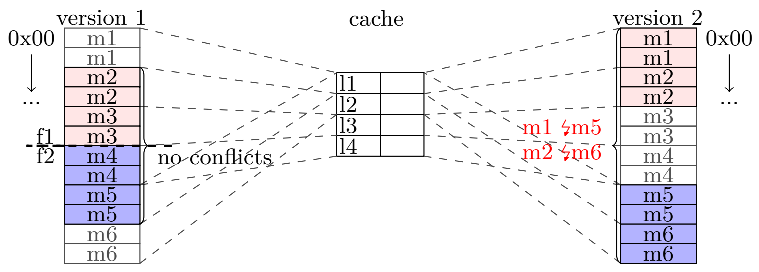
Figure 2c shows two basic blocks whose relative distance differs in the two versions. Now the number of cache blocks per basic block is the same, but the two basic blocks cover conflicting cache blocks. In combination with the different possible offsets this means that every instruction of a basic block in fragment $f1$ can be in conflict with every instruction in fragment $f2$, located at every possible offset. The impact on relative addressing is a concern for caches with $A < n$, because changing distances of cache blocks might also change their set associativity.



(a) Absolute addressing of single basic block.



(b) Absolute addressing of loop.



(c) Relative distance of two basic blocks.

Figure 2 Impact of basic block addresses on cache behavior.

The resulting insights of the impact of diversity are:

- Cache sets cannot be treated independently because every instruction of $f1$ may affect $f2$.
- Fragments can be located at different offsets relative to a cache line. As this may affect cache sets as well, the offsets cannot be handled separately as well.
- The number of cache blocks covered by a single basic block and the mapping of instructions to cache blocks differ in different versions. Therefore, a classification of cache blocks into the classes *AH*, *AM*, *FH* and *NC* is not feasible anymore.

4 Instruction Cache Analysis for Diversified Programs

Our instruction cache analysis is based on abstract interpretation similar to the analyses described in Section 2.3. We use the insights of Section 3 to define an ACS and its update and join functions so that we can cope with diversity. The key idea is as follows:

- For every fragment in F, we assume an own *virtual* memory in our ACS. This way we achieve independence of the fragments and the cache sets within and we get rid of overlapping cache blocks in adjacent basic blocks of different fragments. Note that the size of the memory representation is similar to the memory before, because of the smaller size of the fragments.

- We create an ACS for every possible offset within cache lines a fragment can start at. That way, we can find the worst-case timing for a basic block.
- At the transition of control flow from one fragment to another, every relative distance of addresses is possible. To address this uncertainty, the worst case (that depends on the analysis) is applied to ACS for *all* offsets of cache blocks of other fragments. That way, we do not have to create a tree of ACS in combination of all possible fragment offsets along a path.

These features allow us to adapt to diversity and at the same time to keep and use all the information available during updates in the abstract interpretation fix point algorithm: Relative positioning within a fragment is fully available. Relative intra-fragment positioning information and absolute positioning information are reduced to the worst case of spatial locality, which still allows us to exploit the temporal locality of instructions.

Our abstract cache model for the must analysis is as follows: The cache C is characterized as a set of cache lines (see Section 2.3). The memory model consist of N different virtual memories M consisting of cache blocks m, one for each fragment in F. To also represent all possibilities in absolute addressing (K offsets), the memory representation results in $O \times F \times M$. Our abstract cache state function is adjusted accordingly to map every cache block of the new memory representation to a cache line:

$$\tilde{c} : O \times F \times M \to C \cup \{l_\perp\}$$

$\tilde{C}$ is the set of all cache states, and $\tilde{c}_\perp$ with $\tilde{c}_\perp(o, f, m) = l_\perp$ represents the empty cache state. As described in Section 3, there is no direct affiliation of an instruction with a cache block in our abstract model, as it depends on an offset. In addition to the function $frag : I \to F$, we assume the function $block : O \times I \to M$, which delivers the cache block m of an instruction i within its fragment, given that it starts at an offset o. Given the premise that all cache blocks of a fragment f_1 are potentially in conflict with the cache blocks of another fragment f_2, the notion of cache sets is only valid within a fragment. Also, as only the conflicts between blocks (the distance) are of importance and not the actual set (the absolute set number or address), we set the start of all fragments to the current offset o, starting in the first set.

We use the following auxiliary functions, to deliver the set number and the age of a cache block in a cache state analogous to Section 2.3, along with a shorthand notation for the age of an instruction i at offset o fetched from cache:

$$\begin{aligned}
&\widetilde{set}(o, f, m) : M \to \mathbb{N} \\
&\widetilde{age} : \tilde{C} \times O \times F \times M \to \mathbb{N} \\
&\widetilde{age}_{inst} : \tilde{C} \times O \times I \to \mathbb{N} \\
&\widetilde{age}_{inst}(\tilde{c}, o, i) = \widetilde{age}(\tilde{c}, o, frag(i), block(o, i))
\end{aligned}$$

For better readability of the following definition of our must analysis, we define two additional functions: $deg : C \times \mathbb{N} \to C$ is used to update (degrade) a cache block mapping in a cache state. It selects a new cache line of the same set according to a given age a.

$$deg(l, a) = \begin{cases} l_\perp & | l = l_\perp \vee (a = \infty \wedge l = l_{s_A}) \\ l_{s_{(a'+1)}} & | l = l_{s_{a'}} \wedge a > a' \wedge a' < A \\ l_{s_{a'}} & | l = l_{s_{a'}} \wedge (a \geq 0 \wedge a \leq a') \end{cases}$$

$load_{wc} : \tilde{C} \times P(I) \rightarrow \mathbb{N}$ determines the oldest age of a hit over a set of instructions and all offsets, or a miss if any instruction misses at any offset. This resembles the worst-case cache degradation of cache blocks whose relative distance to the instructions in i is unknown.

$$load_{wc}(\tilde{c}, \langle i_1, ...i_x \rangle) = \max_{\substack{i \in \{i_1,...,i_x\} \\ o \in O}} (\tilde{age}_{inst}(\tilde{c}, o, i))$$

Our update function $\tilde{U}_{must_{BB}}$ is applied at basic block level in two steps: First, we perform the update of cache block mappings in the ACS that belong to the same fragment as the basic block ($\tilde{U}_{must}$), and in a second step we degrade the cache blocks of the other fragments by the worst-case $\tilde{D}_{must}$.

$$\tilde{U}_{must_{BB}}(\langle i_1, ..., i_x \rangle, \tilde{c}) = \tilde{D}_{must}(\tilde{U}_{must}(...(\tilde{U}_{must}(\tilde{c}, i_1), ..., i_x), \langle i_1, ..., i_x \rangle)$$

As stated earlier, our update function $\tilde{U}_{must}$ has to be applied to instructions rather than cache block references, because the mapping between those differs with different offsets. The update is therefore performed per instruction i, which together define the virtual memory space $O \times F \times M$. Cache block mappings for each offset o' in the same fragment are updated in the same way as in the regular must analysis: If they are in another set, they stay unchanged. Otherwise, they get degraded according to the previous cache state of i and o.

$$\tilde{U}_{must}(\tilde{c}, i)(o', f', m') = \begin{cases} deg(\tilde{c}(o', f', m'), a) & | f = frag(i) \wedge m = block(i, o') \\ & \quad \wedge f' = f \wedge \tilde{c}(o', f, m) = l_{s_a} \\ & \quad \wedge set(o', f, m) = set(o', f', m') \\ \tilde{c}(o', f', m') & | otherwise \end{cases}$$

All cache blocks of other fragments get degraded in $\tilde{D}_{must}$ by the worst-case cache access (oldest cache hit or eviction $load_{wc}$) that any of the instructions of that basic block may suffer at any offset, as they potentially conflict with that *worst-case instruction*. This pessimistic way to cope with the uncertainty in relative distances enables us to keep the analyses and the ACS local, combining all possible paths in the original CFG and all possible combinations of offsets of fragments in a memory model that does not grow exponentially with the number of fragments.

Note that this definition of the worst-case cache access assumes that no cache conflicts can occur during the execution of a basic blocks, as the worst case access is a single miss. This can be achieved by limiting the size of a basic block to $\frac{S_C}{A} - \frac{S_L}{2}$ in the preceding CFG construction. The definition uses the implicit property of the diversification that all instructions of a basic block are part of the same fragment.

$$\tilde{D}_{must}(\tilde{c}, \langle i_1, ..., i_x \rangle)(o', f', m') = \begin{cases} deg(\tilde{c}(o', f', m'), load_{wc}(\tilde{c}, \langle i_1, ..., i_x \rangle) & | f' \neq frag(i_1) \\ \tilde{c}(o', f', m') & | otherwise \end{cases}$$

The join function is adjusted to the new memory model in the ACS. No additional pessimism is required at this point, because the worst-case cache behavior is already applied during the updates.

$$\tilde{J}_{must}(\tilde{c}_1, \tilde{c}_2)(o', f,' m') = \begin{cases} l_\perp & | \tilde{c}_1(o', f', m') = l_\perp \vee \tilde{c}_2(o', f', m') = l_\perp \\ \tilde{c}_2(o', f', m') & | \tilde{age}(\tilde{c}_1, o', f', m') < \tilde{age}(\tilde{c}_2, o', f', m') \\ & \quad \wedge \tilde{c}_1(o', f', m') \neq l_\perp \\ \tilde{c}_1(o', f', m') & | \tilde{age}(\tilde{c}_1, o', f', m') \geq \tilde{age}(\tilde{c}_2, o', f', m') \\ & \quad \wedge \tilde{c}_2(o', f', m') \neq l_\perp \end{cases}$$

Figure 3 illustrates the must analysis for diverse software. It contains the same program and cache setting as Figure 1. The cache blocks for each instruction i are named $f.m$, where f refers to the fragment and m is the cache block index within this fragment. The instructions in the basic blocks (rounded rectangles) now contain the cache block every instruction maps to at the two different offsets, separated by slashes. The three dimensions of the ACS are displayed as tables with a column for each offset, and where the rows represent the worst-case cache ages for each cache block in cache, with the youngest at the top. In the update of `bb2`, we can see the regular must analysis within a fragment: In the offset `0` ACS, cache blocks `1.0` and `1.1` are degraded because they conflict with `1.4` and `1.5`. Cache block `1.2` stays in the same line because it gets accessed again. In the offset `0` ACS, `1.2` stays in the top line because there is no conflict with any of accessed blocks, whereas `1.0` and `1.1` are in conflict here as well. The update of the ACS of the different offsets is entirely independent. In `bb3` and `bb4`, we can see how a cache access in one fragment leads to updates of other fragments: the worst-case access to fragment 2 is a cache miss in both cases, and thus all cache blocks of fragment 1 get degraded once in `bb3` and `bb4` each.

The proposed analysis terminates because firstly, the number of memory blocks in the program and the number of blocks in the abstract cache state are both finite and secondly, the update and join functions are monotonous. We refrain from presenting a formal proof and a formal closure of the abstract interpretation, because our analysis differs from Ferdinand's must analysis only in the way the memory blocks are organized and in additional pessimism, resulting in faster evictions of cache blocks. The offsets enhance the ACS by another dimension, which increases the calculation effort, but does not affect the monotony.

4.1 May and Persistence Analysis

The *may analysis* and *persistence analysis* can be defined analogously to the *must analysis*. For space reasons, we keep our description brief and without the formal details, but refer the reader to Ferdinand [16] and Ballabriga [1].

In the may analysis, the cache states are updated so that cache blocks of other fragments are degraded using the youngest age of any cache access within the basic block, and the join function selects the youngest age for each cache block in $O \times F \times M$ out of the cache states.

Ballabriga [1] uses an optimization of the update function for efficiency: The persistence analysis is combined with the must analysis. If a block is accessed, that is not in the must ACS, the age of all other cache blocks is reduced.

Our persistence analysis applies the principles described for the must analysis to Ferdinand's and Ballabriga's persistence analysis: The memory is split into virtual memories M that represent the fragments, and duplicated for each offset in O. The cache model is extended by an additional cache line of age $A + 1$, which is reached if a cache block, once loaded, may have been evicted. The update and join functions resemble a reversed may analysis, where each cache block in the ACS maps to the oldest age that may have been reached (which is $A + 1$ if the block may have been evicted). Ballabriga [1] uses an optimization of the update function for efficiency, which we adapt: The persistence analysis is combined with the must analysis. If a block is accessed, that is not in the must ACS, the age of all other cache blocks is reduced. Just as in our new must analysis, the update function is applied as usual to all blocks within the same fragment as the current basic block. The age of all other blocks is degraded by the worst possible eviction of all accesses in the basic block (up to $A + 1$).

In our multi-level persistence, analogously to Ballabriga, an stack of the ACS described above is created, where each entry represents loop nesting level. The stack is initialized with a regular persistence analysis at the highest level outside any loop. With every loop entry, an

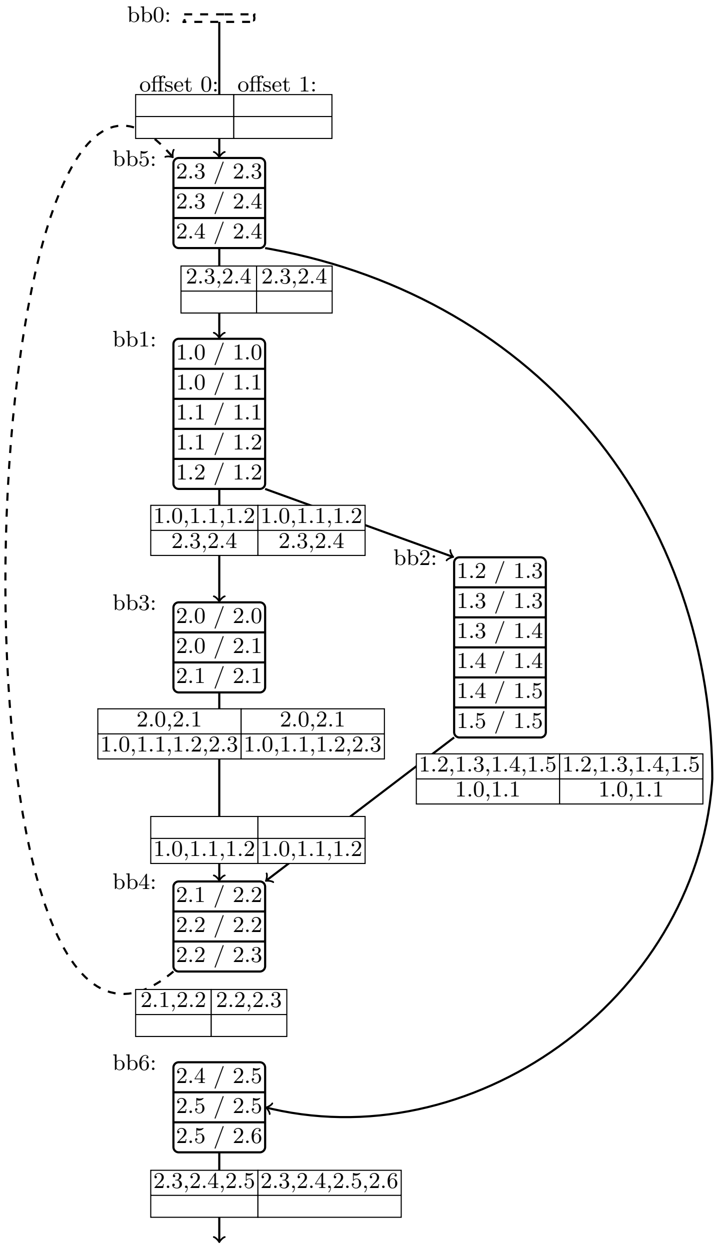


Figure 3 CFG with ACS (must analysis at fix point) of example program using diversity: Two different offsets and fragments.

empty ACS is pushed onto the stack. The entries in this ACS that are in the same fragment as the accessed block can only be replaced by cache blocks, which are accessed within the loop. A cache block that is persistent with respect to a loop is executed as often as the loop is entered, at the worst. In the update function of the multi-level persistence, the cache blocks belonging to other fragments are degraded by the worst case access of the current basic block over all ACS and all offsets.

With our *must* and *persistence* analyses, we have established an abstract representation of all possible cache states before and after the execution of a basic block. To extract a worst-case timing of the execution of the block, we cannot use a direct mapping between instructions and cache accesses. Instead, we accumulate the overall block timing using a classification algorithm, described in the next section.

5 Classification

Our memory model in the ACS causes every instruction to be represented K times, once for every offset, with possibly different cache classifications. These different representations need to be merged in order to be able to deduce a worst-case timing for the basic block. As stated in Section 3, the number of cache blocks may differ over different offsets, and instructions may belong to different cache blocks. Therefore, simply classifying the cache blocks is not feasible. Instead, we determine a worst-case number for each class at basic block level. We nevertheless use the term classification, analogously to previous publications [10, 39].

Finding the worst-case of cache misses AM after applying only the must analysis is straightforward: For each offset $o \in O$, we count the referenced cache blocks that are not classified as a hit ($c(o, f, m) \neq l_\perp$), and use their maximum to calculate the worst-case cache penalty. This penalty is added to the WCET as often as the block is executed.

In the multi-level persistence analysis, the cache block references are additionally classified as persistent, with respect to the outermost loop it is persistent in, adding the classes $P(L0), P(L1)$.... The cache penalty of a cache block classified as $P(L_x)$ is added as often as the outermost loop it is persistent in is executed. The worst-case classification over all offsets is more complicated than maximizing the blocks for each persistence level separately. We use Figure 3 as an example. Basic block `bb3` contains three instructions, hence two cache blocks in each offset. With offset 0, cache block `2.0` is a miss, because after being loaded, it could be evicted by `bb1` and `bb5` (and possibly `bb2`) in the second iteration before being accessed the next time. `2.1`, however, is loaded in `bb4`. In loop iterations other than the first, `bb3`, `2.1` was only degraded once by `bb1`, and is therefore persistent (*P(L0)*). When located at offset 1, `2.1` is not accessed in `bb4`, and therefore both cache blocks cause a miss. To sum it up, at offset 0, cache accesses are classified as $AM = 1, P(L0) = 1$ and at offset 1, $AM = 2, P(L0) = 0$. To determine how many cache loads are necessary at worst for `bb3`, we have to combine the results of all offsets. Now assuming the loop is executed three times, simply counting the worst case of each class, $AM = 2$ and $P(L0) = 1$ would result in $3 * AM + P(L0) = 7$ times the cache penalty, whereas the true worst case is two misses, causing six loads in total. This example did not make use of the fact that the second AM at offset 1, which is included in the combined worst-case, already covers the $P(L0)$ access at offset 0. We solve this by finding the worst case of all accesses for all offsets, instead of a separate worst case of each class.

Assuming a reducible CFG, the execution count of an inner loop is a multiple of the loop entry's execution count. Therefore, we can sort the classes, according to the loop nesting depth, by descending execution count: *AM,...,P(L1),P(L0)*. Our classification works as follows (we denote c_0 for the maximal number of misses and $c_i, (i > 0)$ as the maximal number of persistent blocks of each persistence level starting at c_1 as the persistence level of the inner loop where the block resides. a_{i_o} are the numbers of misses and persistent blocks in offset o):

$$c_i = \begin{cases} \max\limits_{o=1..K} (a_{0_o}) & |i = 0 \\ \max\limits_{o=1..K} (a_{i_o} - \sum_{k=1}^{i-1} (c_k - a_{k_o})) & |i > 0 \end{cases}$$

We find the access counts of each class in the worst case by reducing the count of cache blocks per offset that are in that class by the number of blocks already covered by classes with higher execution counts. In the following example, $max(P(L2))$ is 1 instead of 2, because in offset 1, one cache block is already covered by an extra miss that was identified for offset 0.

Offset	Miss	P(L2)	P(L1)	P(L0)
0	2	1	1	1
1	1	2	2	0
max	2	1	2	0

Our classification does not contain the classes *NC* and *AH*. The above maximum results for misses and persistence already contain a worst case for the timing behavior of the basic block, and thereby include the blocks before classified as *NC*. Note that we assume that the processor does not exhibit timing anomalies. We leave that investigation to future work.

The blocks classified as *AH* would resemble the "rest" of cache blocks, which are not included in the other classes. This differs between the offsets and its maximum does not represent useful information to the WCET analysis. If we assumed that hits also consume time, we would have to include the hits as an extra class of the above classification algorithm.

Using the preceding analyses and our classification we can generate additional ILP constraints for each class per basic block. These worst-case execution counts that represent a safe upper bound for the execution of all variants of the diverse program. In the following section, we present the evaluation we performed on our approach.

6 Evaluation

To evaluate our instruction cache analysis for diversified software, we implemented it in OTAWA, an open research framework for static WCET analysis, [2], and compared its original analyses [1] with our results. We implemented the *must*, *may* and *multi-level persistence analysis* as described in Section 4.

Our experiments were performed using the *Mälardalen benchmark suite* [18]. This set of small example programs is widely used for evaluating static WCET analyses, and therefore enables transparent evaluation results. Out of the Mälardalen benchmarks, nine programs were supported by OTAWA in the publicly available form, when applied to our ARM processor setup including cache analysis. Most of the other benchmarks were dismissed by OTAWA mainly because the ARM processor does not support floating point arithmetic, the flow facts (loop bounds, indirect branches) could not be deduced automatically, or external library functions were used, e.g. to emulate a division operation. In addition, OTAWA's original cache analysis, which we use to compare our results, does not support programs with basic blocks that are too large for the cache size we selected (basic blocks containing conflicting cache blocks cause unnecessary misses).

To increase the number of benchmarks, we slightly modified seven programs by inserting missing compiler functions, simplifying loop bounds and splitting the large basic blocks so that they fit the cache sets without conflict. There were no such fixes available for the remaining benchmark programs. However, the language features being exploited in the benchmarks is not the focus of this evaluation, as we are interested in the binary level. Also, we do not expect the additional pessimism of our analysis compared to state-of-the-art analyses to increase considerably in larger benchmarks, because larger code sections mostly introduce more independent sections with greater temporal distance, less relevant to caching.

The final set of benchmarks we used for this evaluation is summarized in Table 1. In addition to the benchmark name and its size in bytes, the table contains the number of fragments in function-level diversification (Functions) and in block-level diversification

Table 1 Evaluated Malärdalen benchmarks. (*) marks benchmarks with minor changes.

Benchmark	Size(B)	Functions	MIS	$WCET_{AM}$
bs	256	3	6	2445
bsort100	492	3	8	7411365
cnt	676	7	12	111195
crc	1092	4	11	889455
edn*	4104	10	29	2208345
expint*	1332	6	19	6660570
fac	192	2	4	5550
fdct*	2796	3	15	81795
fir*	1152	4	14	25809855
fibcall	192	2	3	8370
insertsort	324	2	4	25815
jfdctint*	2792	3	16	105675
matmult	676	6	11	5122410
ndes	3180	6	27	1440465
nsichneu*	30316	1	2	227130
ud*	2320	5	22	839430

(Movable instruction sequences - MIS). At last, the table shows the $WCET_{AM}$ that has to be assumed without our analysis, where every access is a miss (AM). The benchmark programs, which had to be modified for being supported are marked with an asterisk.

For our experiments, we used a simple 5-stage architecture, where each instruction consumes five cycles and where cache miss adds a latency of ten cycles. We selected a cache size so that the small benchmark programs do not fit entirely and caching effects are visible: The line size S_L is 16 Bytes, i.e. four instructions in the ARM instruction set. The diversity is aligned at the instruction size, resulting in *four offsets*. The number of sets (rows) is 16, and the associativity varies in the experiments. For our experiments, we created different variants of the benchmark programs using different offsets and random order of fragments: 30 for block-level diversity and 10 for function and segment level, respectively. To include the full range of possible offsets for all fragments, we added a random number of `NOP` instructions before the start of the program. The variants were created using the diversification program similar to the one we in introduced in [14].

Using the given setup, we analyzed each benchmark version using different cache dimensions. To obtain a comprehensive insight into the timing effects of our cache analysis, we needed to measure both the absolute timing improvements - compared to a system without caches - and the relative effect - compared to the cache analysis without diversity. We can measure the absolute effect by comparing to the WCET of a system without caches (*all miss* assumption) $WCET_{AM}$, as depicted in Table 1. The relative comparison is done using the WCET obtained by the original analysis without diversity, as implemented in the publicly available version of OTAWA. We calculated this value for each variant of each benchmark, together with the result of our analysis. We define $WCET_{max}$ the highest WCET obtained by that original analysis across all variants of a benchmark using the same cache setup. $WCET_{min}$, accordingly, is defined as the lowest WCET across such variants. Average values across all benchmarks are given as arithmetic mean. OTAWA also offers the possibility of using loop unrolling. We disabled that option in all our experiments.

One particularly interesting cache architecture for our evaluation is a cache with only one line (an instruction buffer). Diversity generally impacts spatial locality, because the execution order of the instructions remains intact and therefore subsequent executions of

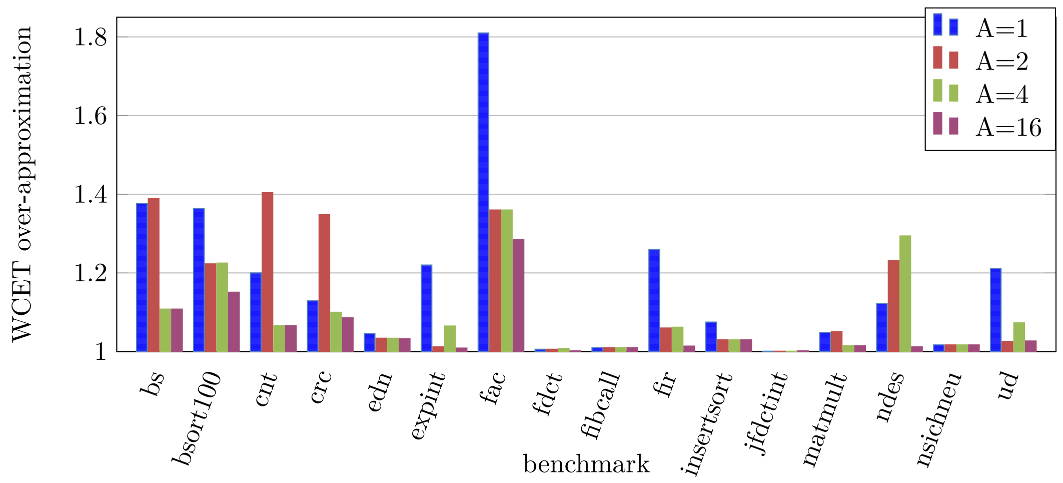


Figure 4 Diverse WCET pessimism with different associativity A with 16 Bytes per line and 16 sets.

instructions have the same temporal distance. In the case of subsequent instructions the temporal locality of the cache block of an instruction is impacted as well, because, depending on the current offset, two instructions may or may not cause a cache line to be accessed twice in a row. To evaluate the performance effect of subsequent accesses to the same cache line, we calculate $WCET_{buf}$, using our analysis on a one-line cache with four instructions (16B).

A necessary requirement for our analysis is that it delivers a safe upper bound that is the same for all versions of a program given a cache size. This requirement was fulfilled in all our experiments: The estimates were always "worse" than those of the original analysis ($WCET_{div} > WCET_{max}$), and they were equal across all versions in the same experiment.

Figure 4 shows the tightness of the estimates using different associativities A and block-level diversity. The WCET estimates $WCET_{div}$ are depicted relative to the corresponding $WCET_{max}$. The results show that the tightness of our analysis greatly depends on the associativity. The over-approximation is the highest for direct-mapped caches, and gets considerably tighter with $A > 2$. The average factor by which our analysis over-approximated in all experiments was 11.6% for basic-block level diversity, 4.3% for function-level, and 1.1% for segment-level diversity, which gives a total of 5.7%.

Figure 5 compares the results of our analysis with the range of non-diverse WCET estimates for all versions of a benchmark, using associativity $A = 2$ and basic block shuffling. There, $WCET_{max}$ and $WCET_{min}$ are depicted alongside the result of our analysis $WCET_{div}$. All results are shown relative to $WCET_{AM}$, the WCET using the *all miss* assumption that is the only choice for a static WCET analysis of diverse programs using the existing approaches. The results show that our analysis is a drastic improvement over assuming *all miss*: The estimates are on average at 44.8% of $WCET_{AM}$. Averaging the results of all experiments with all diversity types and associativities yields 40.8%. Furthermore, the diagram shows the impact of diversity on the cache analysis: $WCET_{max}$ is in average 5.3% higher than $WCET_{min}$. The diagram also shows $WCET_{buf}$, demonstrating the impact of immediate temporal locality. As expected, this effect is responsible for the bulk of the WCET gains, and in some benchmarks (such as `bs`) $WCET_{buf}$ is almost equal to $WCET_{div}$. However, this is not the case for most benchmarks, so that $WCET_{buf}$ averages at only 56% of $WCET_{AM}$.

Figure 6 shows the impact of fragmentation on the analysis result. The WCET estimates for associativity $A = 2$ are depicted for all diversity types, again with $WCET_{div}$ relative

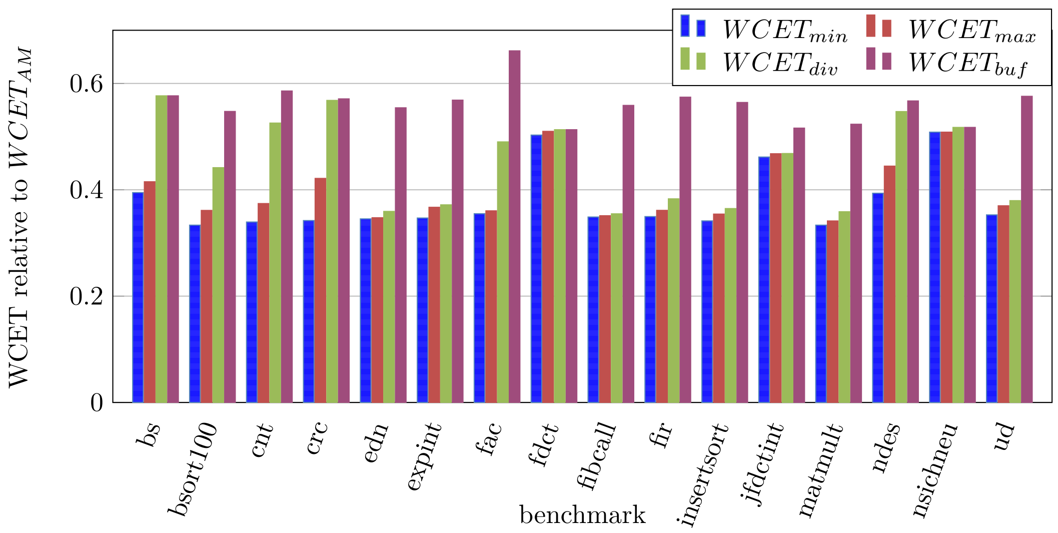


Figure 5 Comparison of WCET estimates using 2-associative cache with 16 Bytes per line and 16 sets.

to $WCET_{max}$. The results show that basic-block diversity produces the highest estimates (avg. 13.8%), while the segment-level diversity causes very low additional pessimism (avg. 1.2%). Function-level diversity averages at 6.0%.

Figure 7 shows the computation time compared to that of the original analysis. We measured the total execution time of each of the above experiments. The diagram shows the average factor by which the computation time exceeds that of the original analysis, in dependence of associativity A and the diversity type. We have expected an increase of computation time, as the ACS is a multiple of the size of the original analysis, because it contains a memory representation per offset. However, to deliver a sound upper bound, the original analysis would have to be executed for each possible version, which would take considerably longer depending on the number of fragments. The results for $A = 1$ show a factor eight in consumed time, while the number of offsets is $K = 4$. The time also increases with associativity A, but not proportionally. We have observed that it took more iterations for the fixpoint algorithm to terminate, and this increased with higher associativity. It can partly be explained with the fact that the cache sets are not independent in our analysis. The comparison of execution times of different diversity types delivers somewhat surprising results: The analysis time increase for segment-level diversity is the highest, while function level diversity performs best.

Note that in theory the performance analysis of OTAWA still has the bug referenced by Cullmann [10], which causes restrictions on the use of indirect branches. Our approach does not fix that, however, we think the demonstration of our approach using the proposed persistence analysis is representative as the error has a low impact, in particular on instruction caches. However, we plan to apply our solution to other persistence analyses as well.

To summarize our results, we can conclude that diversity is not prohibitive for WCET cache analysis. On the contrary, in many experiments our analysis results are very tight, and in all experiments they were far lower than the WCET without analyzing the cache.

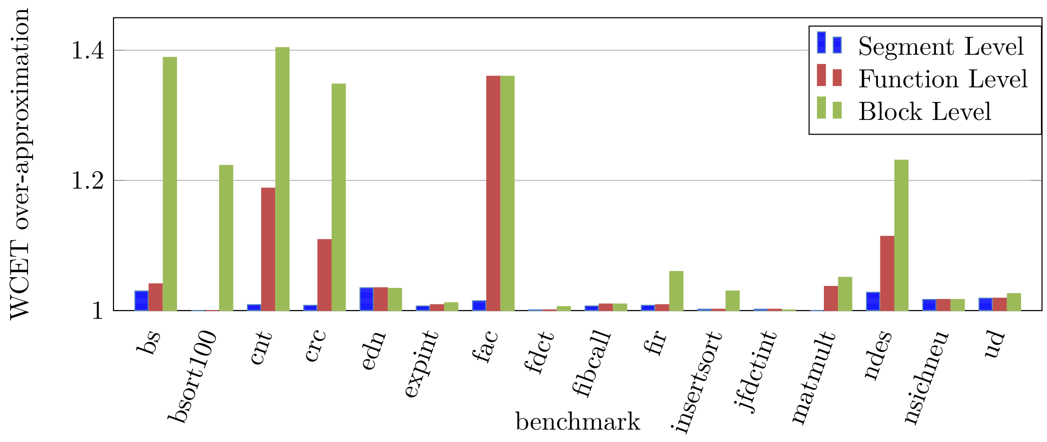


Figure 6 Diverse WCET pessimism with different types of diversity (number of fragments).

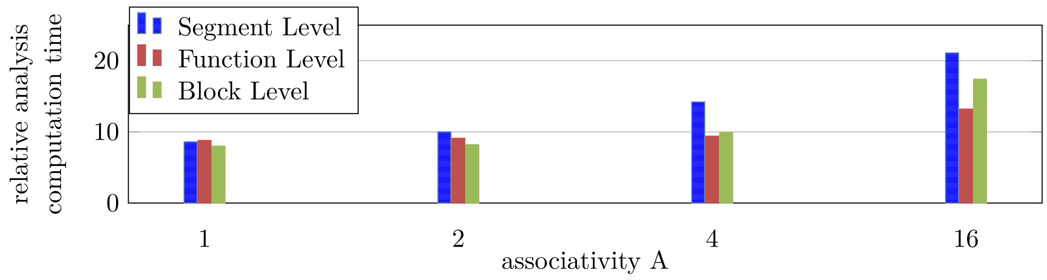


Figure 7 Relative WCET Computation time increase.

7 Related Work

To our knowledge, there is no approach of static WCET cache analysis that supports artificial software diversity. However, many of the existing approaches serve as a basis for our approach or may be complemented with our ideas.

7.1 WCET Analysis

Our approach is based on Ferdinand's abstract interpretation-based analyses [15, 16], as well as Ballabriga's extension, the multi-level persistence [1]. Other approaches improved the persistence analysis as well. Cullmann [10] proposed two analyses, one based on conflict counting and one that combines a regular may analysis with one that collects the oldest age. Huynh [21] collects a set of conflicting blocks that may be younger than a block m at its execution. If there are less than A blocks in this set, m is persistent. These analyses are based on abstract interpretation, thus we believe that our ideas are applicable there as well.

Apart from abstract interpretation there are other static cache analysis techniques, comprehensively surveyed in [27]. [24] introduces cache conflict graphs, explicitly modeling all concrete cache states. [26] uses model checking, which also, implicitly, analyzes all cache states. These approaches suffer from scalability issues, which would be worsened considerably by extending the state space with the uncertainty of diversity.

In [19] also investigates relative addressing of cache contents. However, they concentrate on data flow analyses for data caches, rather than implicit relations between instructions.

WCET-aware compiler optimizations such as code positioning [13] [25] are used to improve the WCET by varying compiler decisions similar to diversity. However, these approaches are based on heuristics and are not able to find a guaranteed upper or lower bound.

7.2 Artificial Diversity

Out of the many artificial software diversity approaches [23], we chose those that limit their transformations to reordering and relocating of fragments, for the reason that we need to be able to predict the WCET of all possible versions. This would be possible as well for narrow-scope in-place code transformations [31]. However, it would require a more detailed analysis of the instruction semantics and would further complicate the ACS construction.

There are also basic-block level approaches that propose splitting the code dynamically into blocks at random positions [11,37] for each variant. This is not supported as they change the CFG and thus the WCET impact over all variants would be unpredictable statically.

Instruction level diversity [20] uses higher fragmentation than block-level diversity. We expect its caching behavior to be very bad, and the analysis is futile as the relative distances of all instructions are unknown.

8 Conclusion

The instruction cache analysis we propose fills an important gap in the research of static WCET analysis, as it is the first to support artificially diversified programs. The key idea is to precisely represent the uncertainties in relative and absolute positioning, which are introduced by diversity, in the analysis, while still exploiting all relative information that is still available. Our analysis supports all artificial diversity approaches where diversification is achieved by reordering and relocation of code fragments. We have discussed that diversity has an impact on caching and its analysis, which is confirmed by our experimental results.

Our evaluation shows that our analysis delivers a safe upper bound for all versions, and that it is a major improvement over assuming *all miss* for the worst case, or not enabling the cache at all. In many cases, the estimations are even very close to the highest WCET that the non-diverse analysis may find. As we chose very small caches for being able to better observe the different effects, we can even expect better results for more common cache sizes.

In addition to systems using artificial software diversity, the analysis is also applicable to other areas, where code fragment positions are, at least partially, unavailable, such as dynamic libraries, or resources linked together from independent teams or vendors.

There are interesting aspects of the approach that deserve further attention. Applying the worst case of cache accesses per basic block is pessimistic considering that several successive basic blocks of the same fragment might not contain any conflicts. We will look into extending this scope to larger regions or sub-paths. We did also not investigate the actual structure of the code, with aspects of fragmented loops, sub-functions within loops and such. Note that we have also presented a WCET aware diversification approach in [14]. We plan to use our cache analysis to optimize fragmentation in this approach with respect to caching behavior. We also plan to consider other hardware features, such as certain kinds of branch prediction, multi-core, multi-level caches etc.

By enabling diversity in the instruction cache analysis of static WCET analysis, our approach delivers an important contribution to make artificial software diversity applicable to more systems. Thus, a critical group of CPS can be protected against code-reuse-attacks, making the systems they are controlling considerably more secure.

References

1 C. Ballabriga and H. Casse. Improving the first-miss computation in set-associative instruction caches. In *2008 Euromicro Conference on Real-Time Systems*, pages 341–350, July 2008. doi:10.1109/ECRTS.2008.34.

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

3 Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

4 Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011. doi:10.1145/1966913.1966919.

5 Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the Fourth ACM Conference on Wireless Network Security*, WiSec '11, pages 127–138, New York, NY, USA, 2011. ACM. doi:10.1145/1998412.1998434.

6 Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Computer and Communications Security*, pages 27–38, 2008. doi:10.1145/1455770.1455776.

7 Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *ACM Conf. on Computer and Communications Security*, pages 559–572, 2010. doi:10.1145/1866307.1866370.

8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

9 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi:10.1145/512950.512973.

10 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):40, 2013.

11 Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 299–310, 2013. doi:10.1145/2484313.2484351.

12 embedded.com EE Times. 2017 embedded market survey, 2017. URL: http://m.eet.com/media/1246048/2017-embedded-market-study.pdf.

13 Heiko Falk and Helena Kotthaus. Wcet-driven cache-aware code positioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 145–154, Taipei, Taiwan, oct 2011.

14 J. Fellmuth, P. Herber, T. F. Pfeffer, and S. Glesner. Securing real-time cyber-physical systems using wcet-aware artificial diversity. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 454–461, Nov 2017. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.88.

15 Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

16 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, 1999.

17 S. Forrest, A. Somayaji, and D.H. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, pages 67–72, 1997. doi:10.1109/HOTOS.1997.595185.

18 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, pages 137–147, 2010.

19 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 102–111. IEEE, 2012.

20 Jason Hiser, Anh Nguyen-Tuong, Michele Co, Mathew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *Security and Privacy*, pages 571–585, 2012.

21 Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 203–212. IEEE, 2011.

22 C. Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC*, 2006. doi:10.1109/ACSAC.2006.9.

23 P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014. doi:10.1109/SP.2014.25.

24 Y. T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *17th IEEE Real-Time Systems Symposium*, pages 254–263, Dec 1996. doi:10.1109/REAL.1996.563722.

25 P. Lokuciejewski, H. Falk, and P. Marwedel. Wcet-driven cache-based procedure positioning optimizations. In *2008 Euromicro Conference on Real-Time Systems*, pages 321–330, 2008. doi:10.1109/ECRTS.2008.20.

26 Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu, and Wang Yi. Mcait-a timing analyzer for multicore real-time software. In *ATVA*, pages 414–417. Springer, 2011.

27 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

28 Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

29 Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016. URL: https://hal.archives-ouvertes.fr/hal-01271552.

30 Frank Mueller, David B Whalley, and Marion Harmon. Predicting instruction cache behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

31 V. Pappas, M. Polychronakis, and A.D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP)*, pages 601–615, 2012. doi:10.1109/SP.2012.41.

32 PaX Team. PaX address space layout randomization (ASLR), 2003.

33 Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. Security and privacy challenges in industrial internet of things. In *Proceedings of the 52Nd Annual Design*

Automation Conference, DAC '15, pages 54:1–54:6, New York, NY, USA, 2015. ACM. doi:10.1145/2744769.2747942.

34 Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561, 2007.

35 L. Szekeres, M. Payer, Tao Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symp. on Security and Privacy (SP)*, pages 48–62, 2013. doi:10.1109/SP.2013.13.

36 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000. doi:10.1023/A:1008141130870.

37 Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Computer and communications security*, pages 157–168, 2012.

38 Yves Younan, Wouter Joosen, and Frank Piessens. Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs. *ACM Comput. Surv.*, 44:17:1–17:28, 2012. doi:10.1145/2187671.2187679.

39 Zhenkai Zhang and Xenofon Koutsoukos. Improving the precision of abstract interpretation based cache persistence analysis. *SIGPLAN Not.*, 50(5):10:1–10:10, 2015. doi:10.1145/2808704.2754967.

Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems

Kristin Krüger
Technische Universität Kaiserslautern
Kaiserslautern, Deutschland
krueger@eit.uni-kl.de
https://orcid.org/0000-0002-3201-5528

Marcus Völp[1]
SnT - Université du Luxembourg
Esch-sur-Alzette, Luxembourg
marcus.voelp@uni.lu
https://orcid.org/0000-0002-8020-4446

Gerhard Fohler
Technische Universität Kaiserslautern
Kaiserslautern, Deutschland
fohler@eit.uni-kl.de
https://orcid.org/0000-0001-6162-2653

Abstract

Much effort has been put into improving the predictability of real-time systems, especially in safety-critical environments, which provides designers with a rich set of methods and tools to attest safety in situations with no or a limited number of accidental faults. However, with increasing connectivity of real-time systems and a wide availability of increasingly sophisticated exploits, security and, in particular, the consequences of predictability on security become concerns of equal importance. Time-triggered scheduling with offline constructed tables provides determinism and simplifies timing inference, however, at the same time, time-triggered scheduling creates vulnerabilities by allowing attackers to target their attacks to specific, deterministically scheduled and possibly safety-critical tasks. In this paper, we analyze the severity of these vulnerabilities by assuming successful compromise of a subset of the tasks running in a real-time system and by investigating the attack potential that attackers gain from them. Moreover, we discuss two ways to mitigate direct attacks: slot-level online randomization of schedules, and offline schedule-diversification. We evaluate these mitigation strategies with a real-world case study to show their practicability for mitigating not only accidentally malicious behavior, but also malicious behavior triggered by attackers on purpose.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Scheduling, Security and privacy → Operating systems security

Keywords and phrases real-time systems, time-triggered systems, security, vulnerability

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.22

Acknowledgements We want to thank the reviewers for their helpful comments which greatly improved this paper. We are also gratefully indebted to Ali Syed, Florian Heilmann and Rodrigo

[1] supported by Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

Coelho from the Technical University of Kaiserslautern and Martina Maggio from Lund University for their comments on an earlier version of this paper. Their insights and expertise assisted research, however, any errors found are our own.

1 Introduction

Real-time systems used to be closed systems running on specialized hardware. Consequently, security had been given little thought, as no access from the outside to these systems was assumed. However, recent trends show the reuse of more and more components for real-time systems, e.g. the shift from federated avionics architectures to IMA (Integrated Modular Avionics) [24], a growing need for connectivity, especially in the area of IoT and networked systems, and a shift from single to multi- and manycore architectures. These trends lead to an increase in the complexity of real-time systems in general and in particular at the real-time application level. This increased complexity implies that real-time systems cannot be considered closed and inaccessible anymore but instead demands anticipating more vulnerabilities and in turn an increased risk of compromise. Security has to be considered during system design and deployment to prevent unauthorized information disclosure and exploitation of vulnerabilities by a potentially malicious, safety-threatening attacker [22]. This is especially true for systems in safety-critical environments, where real-time time-triggered systems are often used. Research on security in the real-time domain, especially for time-triggered systems, however, is still in its infancy [23].

Time-triggered real-time systems [12] provide highly predictable scheduling behavior to meet strict timing constraints. While real-time online scheduling provides *predictability*, i.e, guarantees that deadlines will be met, but not exact times of execution, time-triggered systems provide *determinism*, i.e., given schedule and time, the task executing is known. However, the very properties of determinism, periodicity, and timeliness can be exploited by an attacker. Reusing complex components in a networked environment inherits all classical security concerns and requires appropriate countermeasures. However, in addition, real-time systems enable a class of attacks specifically targeting the timing of applications and thereby the safety to which these tasks contribute. Security is therefore of high concern for safety-critical real-time time-triggered systems.

Having compromised a large enough set of co-scheduled non real-time or low safety-critical tasks, an attacker can make use of leaked scheduling-related information to fine-tune the compromised tasks' behavior such that they generate maximum interference on subsequently executing victim tasks. In order to stay undetected, an attacker could continue normal operation of the compromised tasks up to the point in time when one of the tasks is executed immediately before a safety-critical task. At this time, the compromised task exploits all of its accessible resources to create an access pattern that maximizes interference on the safety-critical task. For example, writing all accessible memory instead of just the locations accessed when executing as analyzed may result in a cache and/ or DRAM access pattern that maximizes cache-related delays of the subsequently executed safety-critical task. Alternatively, on a multicore system, the compromised task could issue the maximum number of allowed memory requests. If memory requests are not handled properly, this may lead to a deadline miss on another core competing for memory access.

Tools analyzing only the legitimate task behavior to determine, e.g., cache-related preemption delays, are not aware of such malicious behavior. Unless the system designer anticipates maximum preemption delays for all tasks, real-time schedules remain susceptible to such attacks. Furthermore, due to its predictability, time-triggered scheduling is inherently vulnerable to timing inference based attacks [25]. In this paper, we analyze inference-based vulnerabilities of time-triggered systems and investigate strategies to mitigate attacks based on exploiting these vulnerabilities without violating the very properties that make time-triggered systems attractive to system designers: timeliness and determinism.

Related Work. In literature, several security solutions for real-time systems exist. For example, Völp et al. [21] prevent timing leaks in fixed-priority schedulers by exploiting the idle task to mask early stops or blocks of a high priority task such that a low priority task always has the same view of the high priority task. Naturally, time-triggered systems do not require this modification since no two tasks coexist in the same time window on the same processor. In [16], Mohan et al. focus on the problem of information leakage over shared resources. They define security levels for tasks and prevent undesirable information flow between tasks of different security levels by flushing the resource. Further, they discuss the integration of security constraints into the design of fixed-priority schedulers. In contrast to [21] and [16], we do not focus on preventing timing channels or information leakage. In fact, we assume timing information may be infered.

Yoon et al. [25] introduce a schedule randomization protocol for task sets scheduled under Rate Monotonic which provides obfuscation against timing inference attacks. As long as deadline constraints are not violated, the next task is picked randomly from the ready queue. Each task has a defined budget of tolerated priority inversions which do not violate the tasks deadline constraints. In Section 3, we follow a similar approach for time-triggered systems.

Two examples for state-of-the-art research deal with security for time-triggered communication. In [19], Skopik et al. introduce a security architecture for time-triggered communication which adds device authentication, secure clock synchronization and application level security. Wasicek et al. [22] investigate the security of time-triggered transmission channels and shows how an authentication protocol secures these channels without violating timeliness properties. In our work, we do not consider intended communication channels for infering timing information, but instead focus on covert or side channels and the implication of attackers learning timing information to coordinate their attacks.

Wasicek [23] further presents a threat model for real-time systems, explores security and dependability in the Time-Triggered Architecture (TTA) in great depth and investigates how to enhance TTA for security. In contrast, we do not focus on a specific architecture for time-triggered systems. More precisely, we show how to mitigate directed attacks by randomizing or changing the schedule without violating the timing constraints of time-triggered schedules.

Contributions.

- We analyze vulnerabilities of time-triggered systems with regard to timing inference and malicious behaviour, and show possible attacks which exploit these vulnerabilities.
- We present two practical mitigation strategies for timing inference based attacks with low implementation complexity: an online job randomization algorithm which is able to preserve the timeliness and predictability properties of time-triggered systems, and offline schedule-diversification.
- We evaluate these mitigation strategies with a real-world case study to show that they have low runtime overhead and are practical.

Paper Structure. The remainder of this paper is organized as follows: In Section 2, we present the vulnerability analysis of time-triggered systems against directed timing inference based attacks. In Section 3, we explore mitigation strategies for directed timing attacks, and evaluate them in Section 4 with a real-world case study.

2 Threat Model and Vulnerabilities

In this section, we first describe our threat and system model, highlighting in particular the assumptions we make on the attacker and how he or she is constrained by time-triggered systems. After that, we analyze the vulnerabilities present in time-triggered systems.

2.1 Threat and System Model

For our vulnerability analysis we assume a time-triggered real-time system running on a single core or a single partition with an offline constructed schedule, e.g. in the form of a table. We assume the schedule has been validated and precautions (such as authenticated boot) are in place to ensure that the validated schedule is correctly deployed to the real-time system.

We assume attackers are able to successfully infiltrate the system through undetected vulnerabilities and will eventually exploit infiltrated outposts to attack further parts of the system. Less stringent evaluation requirements make non real-time tasks and low safety-critical tasks primary targets, but we also do not preclude penetrations of higher-critical tasks. Our concern is that attackers exploit these infiltration points to collect timing information about the system and to coordinate subsequent directed attacks against critical, replicated tasks. In particular, we assume that most critical tasks are sufficiently shielded against direct attacks to require attackers to find a pathway through less critical tasks. Firewalls and gateways in autonomous vehicles and planes support this assumption.

Even though we assume intrusion detection, hardening mechanisms and other defenses against the common attack vectors (e.g., DoS attacks) are in place, we acknowledge that these techniques are imperfect and compromises may go undetected. Of particular concern to us are stealthy attackers that continue normal operation of the compromised tasks until these tasks are executed in a manner where a directed attack is most effective, e.g., immediately before a safety-critical victim task is run. Possible targets of such attacks in time-triggered systems are the low-level control loops. Destabilizing these components (e.g., by increasing the dead time or by introducing jitter in the control cycle) may provoke critical failure modes and thus result in a continuing denial of service [23], or worse, unsafe control decisions.

The timing information required for coordinating such a stealthy attack can be infered via side channels constructed using shared resources like cache or memory, or through covert timing channels, such as the scheduling-covert-channel described by Boucher et al. [1].

While there exist mitigation strategies for closing side channels (for example in the real-time context, the works of Völp et al. [21] or Mohan et al. [16] on fixed-priority schedulers), they are incomplete. Additionally, systematically closing all side channels typically entails significant performance overheads, e.g. when flushing caches prior to scheduling a lower classified task [8]. Meltdown [15] and Spectre [10] are recent examples demonstrating the difficulty of identifying and closing such channels in sufficiently complex architectures. Exploiting non-architectural channels (cache allocation) as communication medium, Meltdown and Spectre extract confidential information from speculative processor state, breaking security on most Intel and many high-end ARM and AMD processors. While real-time systems traditionally avoid such complex hardware, we cannot exclude an integration of cores of this complexity in a real-time system on chip, e.g. for meeting the extended demand of autonomous driving functionalities. Fixing the security flaws of Meltdown and Spectre results in up to 21 percent performance decrease for Intel client systems [3] and up 25 percent for Intel data center systems [2].

We assume the real-time system features isolation mechanisms for enforcing the schedule of tasks and for limiting direct access to the memory of other tasks. Real-time operating systems (RTOS) that feature memory isolation support this assumption unless attackers are able to penetrate the operating system. For the purpose of this paper, we assume the deployed RTOS excludes this possibility.

One immediate consequence of this isolation assumption is that when the attacker has infiltrated the system, he or she is inherently constrained by properties of the system and its architecture for subsequent attacks on more critical tasks. In time-triggered systems, table-driven scheduling prevents influencing other tasks by manipulating the execution time of a compromised task. That is, in contrast to event-triggered scheduling, each task is confined to its execution window and thus the actual task execution time has no influence on subsequent tasks. Time-triggered systems therefore provide temporal isolation of CPU time irrespective of the actual behavior of tasks and without having to revert to timing leak transformations as described for example by Völp et al. [21]. Additionally, messages are only accepted during a certain time window, i.e., if they are timely.

Operating system enforced schedules combined with the assumed impenetrability of the OS ensure that the attacker can neither directly influence the scheduler nor can he read the offline defined scheduling tables. Instead, the attacker has to infer the current schedule from observations he or she makes about the system behavior. As we show in Section 2.2, schedules typically carry too little information to remain secure over extended periods of time even if this information is leaked only over low bandwidth channels. Furthermore, we assume that the global clock remains under exclusive control of the operating system and that it cannot be affected by the attacker.

Even though time-triggered systems eliminate CPU time as shared resource over which information can be leaked and through which other tasks may be influenced, other resources remain through which attackers may gain information and through which they can impact the timing behavior of other tasks. One prominent example of such a resource is the processor cache, which healthy tasks leave behind in a predictable state but which compromised tasks can put into a state that may not be anticipated when computing the worst-case execution time of subsequent tasks.

The use of time-triggered systems imposes further limitation on attackers. For example, side channels and covert channels can only be constructed over explicitly or implicitly shared resources, most of which time-triggered systems already multiplex with the table driven schedule in a manner that is agnostic to the behaviour of executing tasks. Access controls and partitioning techniques like cache [14] or bank coloring [26] further constrain the attacker. However, each such countermeasure negatively impacts system performance. Moreover, as we show in greater detail in Section 2.2, mitigating attacks may require cancelling tempting optimizations such as bounding the delay a task can impose through the cache by evaluating their execution patterns. Designers may be tempted to implement optimizations for the sake of increasing performance while neglecting security.

2.2 Vulnerabilities

One of the main vulnerabilities of a time-triggered system lies in its *deterministic behaviour*. The schedule is the same offline constructed schedule for every hyperperiod. For each point in time, the task executing is known. An attacker who listens to the schedule over a side channel is able to reconstruct the schedule in reasonable time even when the channel has low bandwidth. The schedule comprises only a few bytes of information, thus even with a very low channel bandwidth of, for example, 1 byte per second the schedule is found out in a matter of a few minutes. As we show in Section 4.4, an offline schedule of a real-world system can consist of just 52 bytes. Through the aforementioned channel, the attacker would know the schedule after one minute. Therefore, we reason that timing information can be infered and focus on mitigating directed attacks under this assumption.

Another vulnerability of real-time systems in general is that *worst case execution time (WCET) derivation* does not take malicious behaviour into account. WCET estimated through simulation of the expected behaviour of the system does not account for malicious behaviour. If a task is infiltrated at runtime and, for example, starts accessing the cache to create maximum interference for the next task execution, the tasks simulated worst case does not account for this malicious behaviour if this behaviour is not encountered during uncompromised execution. Prior research on abstract interpretation WCET derivation claims the assumption of cold caches is too pessimistic for a real system and shows methods to achieve tighter and less pessimistic WCET bounds [9], [5]. The assumption of cold caches would nullify the described attack of delaying a task through cache misses. We have to choose WCET estimates in a way that they also account for malicious behaviour and we have to check the impact of performance optimizations on security.

In the next section, we show mitigation strategies for directed attacks which prevent an attacker from exploiting the vulnerability that results if malicious behaviour has not been taken into account.

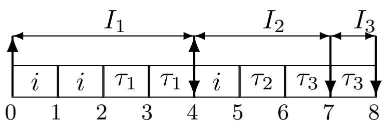
3 Mitigation Strategies

An attacker's goal is to predict as precisely as possible when a victim task gets scheduled immediately after a compromised task to then mount a directed attack. Our primary mitigation strategy is therefore to impede predictions about the point in time when the victim is executed. While we do not prevent timing inference, i.e. we assume the attacker may gain information about the schedule, we are able to counter predictions by changing the points in time when tasks are executed at runtime. For this purpose, we present two strategies to mitigate directed attacks in this section. The first strategy takes an offline constructed time-triggered schedule as input and randomizes the schedule online at job-level while maintaining deadline constraints. This approach is an extended version of the work presented in [13]. The second strategy comprises a set of offline precomputed schedules one of which is randomly chosen at the end of each hyperperiod.

3.1 Slot-level Online Randomization

This mitigation strategy impedes the ability of an attacker to make predictions by randomizing job execution in a time-triggered system at runtime. Schedules for time-triggered systems are typically constructed offline [4], where real-time constraints are resolved and represented in a scheduling table. If not handled properly, online randomization may violate deadline constraints. Therefore, our approach analyzes the scheduling table offline and maps timing constraints of jobs onto execution windows. Execution windows are time intervals defined by the earliest start time of a job and its deadline. Proper handling and, possibly, modification of execution windows solves precedence constraints. Additionally, if one of the goals of the system is to achieve low jitter, we can reduce the size of execution windows accordingly.

During runtime, we randomize job execution within their respective execution windows. While we confine jobs to their execution windows, they still share the same processor so we



τ_i	est_i	d_i	C_i
τ_1	0	4	2
τ_2	0	7	1
τ_3	4	8	2

Figure 1 Job set and capacity intervals derived from offline schedule.

also have to guarantee that their execution does not lead to a deadline miss of other jobs. Slot shifting is a scheduling algorithm which introduces the concept of spare capacities to ensure timely execution [6]. We adopt this concept to guarantee task execution within their respective execution windows even though the scheduling decision is randomized.

3.1.1 Background

Slot shifting uses a discrete time model [11], where the time interval which separates two successive events (i.e. the granularity of the system) is called a slot [18]. We analyze the time-triggered schedule and its task set offline to determine available leeway and unused resources in the schedule for subsequent online adjustment. In order to track the available leeway of jobs in each execution window, a capacity interval is created for each distinct deadline in the system. Jobs with the same deadline belong to the same capacity interval. The start of a capacity interval I_j, $start(I_j)$, is defined as the maximum of the earliest start time $est(I_j)$ of jobs τ_i in this interval and of the end of the previous capacity interval:

$$start(I_j) = max(end(I_{j-1}), est(I_j)) \text{ , with } est(I_j) = min(est(\tau_i)) \ \forall \tau_i \in I_j \tag{1}$$

The end of the capacity interval is determined by the common deadline of all $\tau_i \in I_j$. If needed, empty capacity intervals without assigned jobs are created to fill gaps between capacity intervals with assigned jobs. Figure 1 shows an example job set derived from an offline schedule with earliest start times est_i, worst case execution times C_i and deadlines d_i. We derive the presented schedule in Section 3.1.3. In the schedule on the left side of Figure 1, i denotes the idle task.

Three distinct deadlines exist for that job set, thus at least three capacity intervals have to be created. The first interval I_1 starts at 0 and ends at the deadline of its assigned set of jobs $\{\tau_1\}$, which is 4. The job assigned to next interval, τ_2, shares the earliest start time of τ_1, but according to Equation 1, a capacity interval is not allowed to start before the end of the previous interval. Note that capacity intervals do not overlap, while execution windows may. Thus, I_2 starts at 4 and ends at the deadline of its assigned set of jobs $\{\tau_2\}$, which is 7. We create interval I_3 accordingly. We show the resulting capacity intervals together with an exemplary schedule in Figure 1.

The spare capacity $sc(I_j)$ of a capacity interval I_j is equal to the amount of free slots in I_j. $sc(I_j)$ is defined as the interval length minus the sum of worst case execution times C_i of all its jobs τ_i minus slots borrowed from the succeeding interval (denoted as negative spare capacity), see Equation 2 below.

$$sc(I_j) = |I_j| - \sum_{\tau_i \in I_j} C_i + min(sc(I_{j+1}), 0) \tag{2}$$

Spare capacities are calculated starting from the latest capacity interval in the hyperperiod to the earliest. Borrowing occurs in those cases where the current capacity interval provides insufficient slots to accommodate all its jobs, which results in a negative spare capacity (I_3

in Figure 2). Capacity intervals with a negative spare capacity borrow the needed amount of slots from the preceding interval. Negative spare capacities do not necessarily imply infeasibility in the scheduling sense. Spare capacities are a means to track "free" slots in a capacity interval. We show the resulting offline calculated spare capacities (for time t=0) in Figure 2 of Section 3.1.3, where we present the spare capacity calculation.

If we have calculated all spare capacities, the first capacity interval has a non-negative spare capacity provided the task set is feasible, i.e. its utilization is equal to or less than one since we consider single core systems. Positive spare capacities represent the amount of unused resources and leeway [6] of an interval which can be given to other tasks with overlapping execution windows to adjust the schedule. Such adjustments may require updating spare capacities. At runtime, we update the spare capacities after each slot to reflect the impact of scheduling decisions on the availability of "free" slots.

We consider three different cases for spare capacity updates:

1. No job executes in a given slot. In this case we have to decrease the spare capacity of the current capacity interval by one.
2. A job executes which belongs to the current capacity interval. In this case the spare capacity of the current interval does not change because the WCET of this job is already considered.
3. A job executes which belongs to a later capacity interval. In this case the current interval's spare capacity needs to be decreased by one, but executing the job ahead of time frees resources in its assigned interval. We can therefore increase the spare capacity of the job's interval by one. If this capacity increase happened on a negative spare capacity (i.e., the job's interval is borrowing from another capacity interval), we also increase the spare capacity from the interval from which it borrows, as it needs to lend one slot less. Cascaded borrows are resolved recursively in a similar fashion.

The original slot shifting algorithm in [6] and [18] further integrates aperiodic tasks into a time-triggered schedule. In this paper, we only adopt the concept of capacity intervals and spare capacities to guarantee timely execution of periodic jobs within their execution windows without violating constraints of other jobs. Thus, our offline algorithm needs to create only one table with execution windows and a second one with intervals and their respective spare capacities. For our online randomizing scheduler, we update the spare capacities at runtime to keep track of scheduling decisions.

3.1.2 Slot-Level Randomization of Jobs

Our first attack mitigation strategy is to randomize job execution at runtime. Therefore, at the beginning of each slot, we invoke the online scheduler to select the next job from all tasks in the ready queue at random. We consider the idle task to be part of the ready queue in order to allow for more permutations of the schedule. Even though we select tasks randomly, we have to guarantee that no scheduled job violates the deadline constraints of other jobs. Thus, before taking a scheduling decision, we check if the spare capacity of the current capacity interval is greater than zero. If this condition is fulfilled, any job is allowed to run, as sufficient time remains in the current and later intervals such that no job misses its deadline. In other words, as long as the schedule has leeway, each ready job has the same probability of getting selected for a slot. Otherwise, if the spare capacity of the current interval drops to zero, there is no more leeway to schedule arbitrary jobs. However, because we have already considered jobs of the current capacity interval in the spare capacity computation and because all such jobs share the same deadline, we can still randomize their

time t	0	1	2	3	4	5	6	7	8
$sc(I_1)$	2	2	1	0	0	0	0	0	0
$sc(I_2)$	1	1	1	2	2	1	1	0	0
$sc(I_3)$	-1	-1	-1	-1	-1	-1	0	0	0

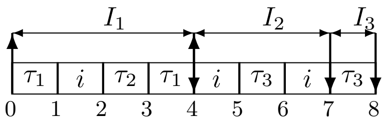


Figure 2 Left: Spare Capacities of I_1, I_2 and I_3 over time, Right: Randomized Schedule.

execution. That is, in the case of zero leeway, the online scheduler randomly selects among the jobs of the current capacity interval. After running a job, we update spare capacities as shown in Section 3.1.1.

Combining time-triggered scheduling with our slot-level randomization impedes online predictions about the schedule. Since the scheduler randomly selects the next job at runtime, predictions about which job runs next are not possible as long as execution windows allow for leeway. Furthermore, time-triggered scheduling inherently confines application-level leakage to shared resources which are held across slots [20]. An investigation of leakage countermeasures for such resources is out of the scope of this paper. While our randomization algorithm does not allow for slot-level determinism typical for time-triggered systems, it still allows for execution window determinism [7].

3.1.3 Example

Let us illustrate the proposed scheduling algorithm for our example jobset depicted in Figure 1. First, we have to calculate the initial spare capacities of the capacity intervals. Starting at the last capacity interval, I_3, its spare capacity is the difference between the interval length of 1 and the worst case execution time of its assigned jobs, here only τ_3, which results in a spare capacity of: -1. I_2 has an interval length of 3, from which we substract the worst case execution time of τ_2 (i.e., $C_2 = 1$) and the slots borrowed by the preceding interval I_3 (by adding $sc(I_3) = -1$), which results in a spare capacity of 1. We calculate the spare capacity of I_1 accordingly. Figure 2 shows the resulting spare capacities in the column for time $t = 0$.

At time $t = 0$, the scheduler sees that the spare capacity of the current interval I_1 is positive and picks τ_1 randomly for the first slot at $t = 0$ from the list of ready jobs τ_1, τ_2, plus the idle job (i). As τ_1 executes within its own interval, the current spare capacity does not change and remains positive. The idle job i is selected to execute during the next slot starting at $t = 1$, necessitating a decrease of the spare capacity by one. τ_2 is randomly selected for time $t = 2$. τ_2 does not execute within its own capacity interval, therefore we reduce $sc(I_1)$ by one and increase $sc(I_2)$ by one, since τ_2 belongs to interval I_2 and I_2 does not borrow from I_1. $sc(I_1) = 0$ at $t = 3$ constrains the online scheduler to select from the set of jobs $\{\tau_1\}$ that are assigned to I_1. At time $t = 4$, τ_3 becomes active and is selected to execute at time $t = 5$ after picking the idle thread to $t = 4$. This is valid, as $sc(I_2)$ is positive, and thus we reduce $sc(I_2)$ by one and increase the capacity interval of τ_3, I_3, by one. However, at this time, I_3 is still borrowing one slot from I_2. τ_3 executed prior to its own capacity interval, thus I_3 needs to borrow one slot less from I_2 and therefore we increase $sc(I_2)$ by one, resulting in no change of $sc(I_2)$. In summary, $sc(I_2)$ stays at 1 and $sc(I_3)$ is increased by one. We show further exemplary scheduling decisions and spare capacity updates in Figure 2.

3.2 Offline Schedule-Diversification

The second mitigation strategy we investigate in this work constructs multiple offline precomputed schedules and switches between them at hyperperiod boundaries. Resolving scheduling constraints offline ensures lower runtime overheads, but increases the chance of attackers to guess the schedule and launch directed attacks. For example, repeating the same offline computed schedule several times allows an attacker to deduce the schedule, as illustrated for example in [16], and to coordinate directed attacks from compromised tasks scheduled later in the same hyperperiod or in subsequent hyperperiods. To partially mitigate this threat vector, we randomly switch schedules at the end of each hyperperiod. As a consequence, even when the attacker is able to recognize different schedules and has enough memory available to store them, the more schedules have been generated, the harder it is for the attacker to recognize which schedule has been chosen for the current hyperperiod and the less time remains to launch a directed attack. In particular, if the attacker is not able to identify the current schedule in time for his attack, the attacker misses the opportunity to launch a directed attack.

We show in Section 4.5 that computing and storing all possible, feasible schedules in memory is impractical. However, in non-embedded systems (e.g., SCADA), we foresee the continuing generation of schedules in a non real-time subsystem (e.g., in a sufficiently protected external control station) and an update of the set of schedules downloaded to the real-time device. This way, once a new set of schedules has been produced (possibly by recombining precomputed and stored schedules), the real-time device can switch to the new set at the end of the hyperperiod. Double buffering, signing and encryption of schedules ensures that the current set of schedules remains valid while the system validates the confidentiality and integrity of the new schedules (e.g., in a background task). Irrespective of update possibilities, the selected subset of schedules out of the set of all feasible schedules for a given task set should impede directed attacks as much as possible. We present two criteria to select subsets that complicate directed attacks in addition to guaranteeing deadlines and respecting task precedence constraints. Carefully created execution windows solve deadline and precedence constraints.

Random Selection. For the sake of low implementation complexity, the subset can be selected randomly. That is, schedules are created randomly and checked to meet all scheduling constraints. The schedules fulfilling this requirement form the set of schedules for the system. Schedule creation is stopped after a certain number of feasible schedules has been constructed. We recommend this method for large subsets, when enough memory is provided to store a large number of different schedules. If the subset is large enough, the random selection process provides a set of schedules with a schedule entropy close to the set of all feasible schedules. Other criteria impose more constraints on the selection process and therefore increase its complexity.

Schedule Entropy. Another criterium for schedule selection is schedule entropy as presented in [25]. This measure makes use of the Shannon entropy and is used to quantify the difference, i.e. randomness, between schedules. A subset of feasible schedules is chosen in a way to maximize the schedule entropy for the number of chosen schedules. However, Yoon et al. [25] showed that calculating the schedule entropy has asymptotic exponential complexity because it requires the enumeration of all possible schedules. They provide an approximation of the schedule entropy over the sum of slot entropies called upper-approximated schedule entropy, which is calculated using the probability mass function of a task appearing at a certain slot

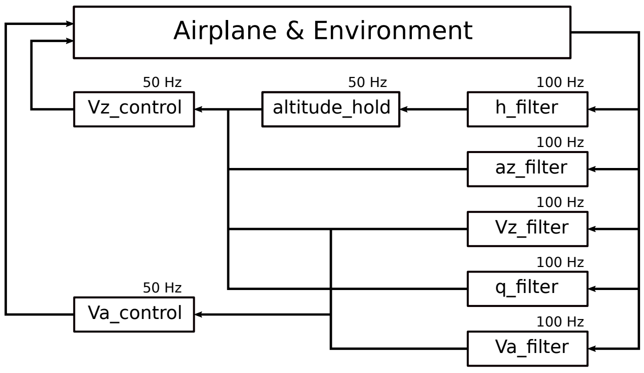


Figure 3 Longitudinal flight controller design.

in the schedule. However, finding a subset with n schedules with the global maximal schedule entropy for all subsets of size n also requires enumeration of all feasible schedules, which is impractical. Therefore, we can apply heuristics for local maxima or select schedules such that the entropy is above a tolerable threshold. For example, we first construct a subset of randomly chosen, feasible schedules with size significantly greater than n, from which we then select the smaller subset of size n with the highest entropy.

4 Evaluation

We evaluated our two directed attack mitigation strategies, which we presented in Section 3, with the ROSACE case study [17]. ROSACE is a practical, real-world example of a real-time system in a safety-critical environment: avionics. This section presents our results.

4.1 Case study: Flight Controller

Pagetti et al. [17] carried out a case study of a longitudinal flight controller of an aircraft. The longitudinal flight controller helps the pilot to accurately track altitude, vertical speed and airspeed of the aircraft. Pagetti et al. describe two control loops: the $Va\_control$ loop handles airspeed control by maintaining the desired airspeed Va; the second control loop — altitude control — combines $altitude\_hold$ and $Vz\_control$. First, $altitude\_hold$ translates altitude commands to vertical speed commands. Then, $Vz\_control$ tracks the vertical speed Vz of the aircraft. Both control loops are fed with filtered data: h, az and q for altitude, vertical acceleration and pitch rate, respectively. Vertical Vz and true airspeed Va are also inputs to the control loops. We show the design of the controller in Figure 3.

According to Pagetti et al. [17], the closed-loop system with continuous-time controllers can tolerate delays of up to roughly 1 second before destabilizing. To preserve stability as well as to increase performance, Pagetti et al. chose lower sampling periods of 50 Hz for the digitalization tasks of the three controller blocks and 100 Hz for the filter tasks which feed the data to the controller. Pagetti et al. derived worst case execution times of all tasks using a measurement-based approach by measuring the repeated execution of a task in isolation. The granularity the authors chose for the measuring clock was 100μs, thus the worst case execution times for the tasks shown are the same as they presumably finished execution in that granule. Table 1 shows the task set with implicit deadlines for the longitudinal flight controller. In this work, we do not consider environment simulation tasks as they are not part of the controller but only of the test environment.

Table 1 Flight controller task set[17].

Taskname	Frequency	WCET
Vz_control	50Hz	100μs
Va_control	50Hz	100μs
altitude_hold	50Hz	100μs
h_filter	100Hz	100μs
az_filter	100Hz	100μs
Vz_filter	100Hz	100μs
q_filter	100Hz	100μs
Va_filter	100Hz	100μs

Table 2 Execution windows.

Name	Start	End	WCET
h_filter	0	50	1
az_filter	0	50	1
Vz_filter	0	50	1
q_filter	0	50	1
Va_filter	0	50	1
h_filter	50	100	1
az_filter	50	100	1
Vz_filter	50	100	1
q_filter	50	100	1
Va_filter	50	100	1
altitude_hold	0	100	1
Vz_control	0	100	1
Va_control	0	100	1

We construct the execution windows of all tasks from the task set in Table 1. Schorr [18] suggests 200,000 clock cycles as slot shifting slot length. The processor cores in ROSACE run at 1.2GHz, which results in 167 μs for 200,000 clock cycles. We choose 200 μs as slot length to evenly divide the task periods into slots. Task execution is non-preemptive, as the worst case execution times are smaller than the slot length. Table 2 shows the resulting execution windows.

4.2 Runtime Overhead for Slot-Level Randomization

Our slot-level randomization algorithm is based on Schorr's [18] slot shifting algorithm. Schorr measured the runtime overhead of the unmodified slot shifting algorithm on a cycle-accurate ARM quadcore simulator — MPARM — with ARM7 cores running at 200 Mhz, 8kB 4-way set associative L1 cache, 8kB direct mapped L1 instruction cache, 1MB core-private memory and 1MB shared memory. Schorr provided minimum and maximum runtimes of all parts of the slot shifting algorithm for single core execution. Using the timing measurements of [18], shown in Table 3, we approximate the runtime overhead of slot-level randomization, when executed on the same processor.

Slot-level randomization invokes the same functions to update spare capacities and the ready list. The cost of the function to update spare capacities increases with the number of intervals due to cascaded borrowing in the worst case. However, according to the slot shifting

Table 3 Minimum and maximum runtime overhead for single core execution in ns [18].

Function	Min	Max
update spare capacity (up_{sc})	2,655	10,145
update ready list (up_{ready})	3,500	9,115
next job selection (sel)	1,850	2,350
ISR overhead (ISR)	2,560	3,120

Table 4 Minimum and maximum runtime overhead for single core execution in ns [18].

Function	Min	Max
next job selection (sel)	1,850	2,350
ISR overhead (ISR)	2,560	3,120

algorithm as explained in Section 3.1.1, only 2 intervals are created for the presented task set. Hence, the costs of both functions remain the same. The interrupt service routine (ISR) overhead is architectural and hence should not change for an implementation of slot-level randomization in the same operating system. Randomization is not part of slot shifting and as such not covered by the above measurements. As calculating random numbers for each slot is independent of parameters like the number of tasks or intervals, we assume a constant per slot overhead. Moreover, assuming an O(1) `get_length` implementation of the ready list, pruning random values to a list index remains a constant operation.

We calculate the maximum runtime overhead as:

$$t_{ov,rand,max} = rand_{max} + up_{sc,max} + up_{ready,max} + sel_{max} + ISR_{max} \tag{3}$$

Accordingly, the minimum runtime overhead results in:

$$t_{ov,rand,min} = rand_{min} + up_{sc,min} + up_{ready,min} + sel_{min} + ISR_{min} \tag{4}$$

Using the measurements from Table 3 for equation 3 and assuming $rand_{max} = 5,000ns$, the maximum runtime overhead results in $t_{ov,rand,max} = 29,730ns$, which is around 3 percent of the assigned slot size of 1ms in [18]. Keeping in mind that ROSACE uses 6 times faster cores than [18] and that execution time does not scale exactly linear with processor speed, we can approximate the runtime overhead for ROSACE. Therefore, we divide these values by 5 for a core with 1.2 Ghz and approximate the maximum runtime overhead for ROSACE to be $t_{ov,rand,max} = 6,000ns$.

Under the assumption that $rand_{min} = 2,000ns$, the minimum runtime overhead results in $t_{ov,rand,min} = 12,565ns$, which is around 1.3 percent of the slot size in [18]. Dividing these values by 5 as explained earlier, we approximate the minimum runtime overhead for ROSACE to be $t_{ov,rand,min} = 2,500ns$.

4.3 Runtime Overhead for Offline Precomputed Schedules

The runtime overhead for offline precomputed schedules is lower than that of scheduling algorithms which have to take more complex decisions online, which we also prove in this section. Again we can make use of the overhead measurements done in [18], which we show in Table 4.

At runtime, the scheduler performs a table lookup to select the next job after each slot. In constrast to the slot-level randomization scheduling algorithm, the overhead only consists of the next job selection and the interrupt service routine. At the end of the hyperperiod, we

Table 5 Exemplary precomputed time-triggered schedule for ROSACE.

ID	Start	End	WCET
0	1	2	1
1	8	9	1
2	22	23	1
3	33	34	1
4	35	36	1
0	51	52	1
1	58	59	1
2	66	67	1
3	67	68	1
4	71	72	1
5	80	81	1
6	88	89	1
7	94	95	1

select the next offline precomputed schedule randomly. We calculate best and worst case runtime overhead for selecting a precomputed schedule in MPARM as shown below.

$$t_{ov,prec,max} = rand_{max} + sel_{max} + ISR_{max} = 10470ns \tag{5}$$

$$t_{ov,prec,min} = rand_{min} + sel_{min} + ISR_{min} = 6410ns \tag{6}$$

Using the same estimation on the execution time of the randomization function for the ROSACE case study as in Section 4.2, best and worst case approximated overhead results in 1300 ns and 2100 ns, respectively. Thus, around 1 percent of the chosen slot size is used for scheduling for both ROSACE and on the ARM simulator MPARM.

4.4 Memory Cost for Offline Precomputed Schedules

Each precomputed schedule needs to be stored in memory. For ROSACE, we can build an offline schedule in the same way as Table 2 suggests. Each task has its own task ID, an entry for the start and end of the execution of its instance, and a fourth entry for its worst case execution time. The difference between start and end time must be equal to its worst case execution time and the execution windows for different jobs must not overlap. Table 5 shows an example for a precomputed time-triggered schedule.

Assuming each entry has the size of 1 byte, a single schedule with this information needs $13 * 4 = 52$ bytes of memory.

4.5 Discussion

Slot-level randomization proves to be practical, as the approximated overhead in Section 4.2 shows. In the worst case, slot-level randomization uses less than 3 percent of the slot for scheduling. Precomputing offline schedules can further reduce this overhead to roughly 1 percent of the slot size, but physical memory capacity limits the number of offline precomputed schedules that can be stored in a system. It is possible to offload scheduling tables to secondary storage by accepting an increase of scheduling overhead while loading the selected scheduling table from this memory.

Even for side channels with low bandwith as we mentioned in Section 2.2, an attacker might identify a small number of schedules after several minutes or a few hours. In order to show how many possible schedules slot-level randomization covers, we calculate the total number of possible feasible schedules for the task set presented in Table 2. For each execution window, the binomial coefficient $\binom{n}{k}$ calculates the number of possibilities to execute the task in different slots, where n is the window size and k the worst case execution time, both quantified in slots. The binomial coefficients of neighbouring and overlapping execution windows are multiplied with each other. If execution windows overlap, we subtract the worst case execution time of tasks belonging to execution windows whose binomial coefficients are already accounted for in the equation ("preceding" binomial coefficients) from the window size. Thus, we calculate the number of possible feasible schedules for the presented task set as shown below. On the left side of the equation, the binomial coefficients of the five tasks with periods of 50 slots are calculated two times, because the hyperperiod results in 100 slots. Their combined worst case execution time of 10 slots is then substracted from the execution window sizes of the tasks with a period of 100 slots.

$$\left[\binom{50}{1}\binom{49}{1}\binom{48}{1}\binom{47}{1}\binom{46}{1}\right]^2 \times \binom{90}{1}\binom{89}{1}\binom{88}{1} = 4.56 \times 10^{22} \tag{7}$$

4.56×10^{22} schedules with 52 bytes require 2^{81} bytes of storage, so we can safely conclude that it is infeasible to track or store all possible schedules in terms of memory space and computation time needed. Positive spare capacities, i.e. leeway in the schedule, are key for a high number of distinct feasible schedules.

Even under the assumption that the attacker is able to store a huge number of schedules, the higher the number of precomputed schedules, the longer it takes the attacker to be sure which schedule is used. Updating the stored scheduling tables partially mitigates the threat that the attacker might eventually identify the schedule in time. The threat is fully mitigated with slot-level randomization, which we recommend in general, due to the comparable overhead, and for systems with strict memory constraints.

5 Conclusion

In this paper we described vulnerabilities of time-triggered systems to timing inference based directed attacks and presented two mitigation strategies. The deterministic behaviour of time-triggered systems allows attackers to infer timing information over side channels and precisely target victim tasks. Worst case execution time assumptions, on which schedules are based, do not take malicious behaviour into account. As the schedule of a time-triggered system comprises only a few bytes, it can be infered by an attacker. In order to prevent attackers from predictions about the point in time when a certain task is executed, we presented two mitigation strategies for directed attacks. First, we introduced slot-level randomization, which impedes predictions about the schedule by selecting the next job at random. We employ concepts of slot shifting to allow randomization of a time-triggered schedule without violating deadlines. Secondly, we proposed online selection of offline precomputed schedules for mitigation of directed attacks. At runtime, a schedule from a precomputed set of schedules is randomly selected at the end of each hyperperiod. We evaluated both mitigation strategies with respect to overhead and memory cost with a practical, real-world case study of a safety-critical flight controller. Slot-level randomization has a runtime overhead of around 3 percent in the worst case, which makes it suitable for practical use. Scheduling precomputed schedules reduces the worst case runtime overhead to around 1 percent of the slot size, but is more costly in terms of memory. A single schedule for the case study has a size of 52 bytes, but the total number of feasible schedules lies in the magnitude of 10^{22}. We proved both mitigation strategies to be practical. An attacker could still try to launch undirected attacks, but he or she will be easier to detect this way.

For future work, offline schedulers may be enhanced to consider entropy during schedule creation. Moreover, imperfect randomization leaves a residual side channel. Therefore, we are interested in a simulated attack measuring the influence a compromised task has against its victim using our mitigation strategies and to further examine if there exist attack vectors particularly effective against our approach. Lastly, we intend to integrate our approach into a multicore system with partitioned scheduling.

References

1 Peter K. Boucher, Raymond K. Clark, Ira B. Greenberg, E. Douglas Jensen, and Douglas M. Wells. *Toward a Multilevel-Secure, Best-Effort Real-Time Scheduler*, pages 49–68. Springer Vienna, Vienna, 1995. `doi:10.1007/978-3-7091-9396-9_8`.

2 Intel Corporation. Firmware Updates and Initial Performance Data for Data Center Systems. accessed on 26/01/2017. URL: `https://newsroom.intel.com/news/firmware-updates-and-initial-performance-data-for-data-center-systems/`.

3 Intel Corporation. Intel Security Issue Update: Initial Performance Data Results for Client Systems. accessed on 26/01/2017. URL: `https://newsroom.intel.com/editorials/intel-security-issue-update-initial-performance-data-results-client/`.

4 Silviu S. Craciunas and Ramon Serna Oliver. SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 45:45–45:54, New York, NY, USA, 2014. ACM. `doi:10.1145/2659787.2659812`.

5 Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2):131–181, Nov 1999. `doi:10.1023/A:1008186323068`.

6 G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 152–161, Dec 1995. `doi:10.1109/REAL.1995.495205`.

7 Gerhard Fohler. *Advances in Real-Time Systems, Chapter Predictably Flexible Real-time Scheduling*. SPRINGER, 2012.

8 W. M. Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–61, May 1992. `doi:10.1109/RISP.1992.213271`.

9 B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-Aware Data Cache Analysis for WCET Estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, April 2011. `doi:10.1109/RTAS.2011.27`.

10 Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, 2018. `arXiv:1801.01203`.

11 H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 460–467, Jun 1992. `doi:10.1109/ICDCS.1992.235008`.

12 H. Kopetz and G. Grünsteidl. TTP-a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan 1994. `doi:10.1109/2.248873`.

13 Kristin Krüger, Marcus Völp, and Gerhard Fohler. Improving Security for Time-Triggered Real-Time Systems against Timing Inference Based Attacks by Schedule Obfuscation. In

29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Work-in-Progress Proceedings, pages 4–6, 2017.

14 J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, Jun 1997. doi:10.1109/RTTAS.1997.601360.

15 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, jan 2018. arXiv:1801.01207.

16 Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh B Bobba. Integrating security constraints into fixed priority real-time schedulers. *Real-Time Systems*, pages 1–31, 2016.

17 C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE case study: From Simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318, April 2014. Open Source avionics task set. doi:10.1109/RTAS.2014.6926012.

18 Stefan Schorr. *Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures*. PhD thesis, Technical University of Kaiserslautern, March 2015.

19 Florian Skopik, Albert Treytl, Arjan Geven, Bernd Hirschler, Thomas Bleier, Andreas Eckel, Christian El-Salloum, and Armin Wasicek. *Towards Secure Time-Triggered Systems*, pages 365–372. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-33675-1_33.

20 M. Völp, B. Engel, C. J. Hamann, and H. Härtig. On confidentiality-preserving real-time locking protocols. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013. doi:10.1109/RTAS.2013.6531088.

21 Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding Timing Channels in Fixed-priority Schedulers. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 44–55, New York, NY, USA, 2008. ACM. doi:10.1145/1368310.1368320.

22 A. Wasicek, C. El-Salloum, and H. Kopetz. Authentication in Time-Triggered Systems Using Time-Delayed Release of Keys. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 31–39, March 2011. doi:10.1109/ISORC.2011.14.

23 Armin Rudolf Wasicek. *Security in Time-Triggered Systems*. PhD thesis, Technische Universität Wien, 2011.

24 C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, pages 2.A.1–1–2.A.1–10, Oct 2007. doi:10.1109/DASC.2007.4391842.

25 M. K. Yoon, S. Mohan, C. Y. Chen, and L. Sha. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. doi:10.1109/RTAS.2016.7461362.

26 H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014. doi:10.1109/RTAS.2014.6925999.

Recovery Time Considerations in Real-Time Systems Employing Software Fault Tolerance

Anand Bhat
Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA
anandbha@andrew.cmu.edu
https://orcid.org/0000-0001-8703-7057

Soheil Samii
General Motors R&D, Warren, MI, USA and Linköping University, Sweden
soheil.samii@gm.com, soheil.samii@liu.se

Ragunathan (Raj) Rajkumar
Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA
rajkumar@andrew.cmu.edu

Abstract

Safety-critical real-time systems like modern automobiles with advanced driving-assist features must employ redundancy for crucial software tasks to tolerate permanent crash faults. This redundancy can be achieved by using techniques like active replication or the primary-backup approach. In such systems, the recovery time which is the amount of time it takes for a redundant task to take over execution on the failure of a primary task becomes a very important design parameter. The recovery time for a given task depends on various factors like task allocation, primary and redundant task priorities, system load and the scheduling policy. Each task can also have a different recovery time requirement (RTR). For example, in automobiles with automated driving features, safety-critical tasks like perception and steering control have strict RTRs, whereas such requirements are more relaxed in the case of tasks like heating control and mission planning. In this paper, we analyze the recovery time for software tasks in a real-time system employing Rate-Monotonic Scheduling (RMS). We derive bounds on the recovery times for different redundant task options and propose techniques to determine the redundant-task type for a task to satisfy its RTR. We also address the fault-tolerant task allocation problem, with the additional constraint of satisfying the RTR of each task in the system. Given that the problem of assigning tasks to processors is a well-known NP-hard bin-packing problem we propose computationally-efficient heuristics to find a feasible allocation of tasks and their redundant copies. We also apply the simulated annealing method to the fault-tolerant task allocation problem with RTR constraints and compare against our heuristics.

2012 ACM Subject Classification Software and its engineering → Software fault tolerance, Software and its engineering → Real-time systems software, Computer systems organization → Real-time systems

Keywords and phrases fault tolerance, real-time embedded systems, recovery time, real-time schedulability

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.23

1 Introduction

Advances in sensing, machine learning and semiconductor technology have resulted in a dramatic increase in the amount and complexity of computational resources used in real-time systems. Many of these systems, such as industrial control, aviation and automobiles

[35], are also safety-critical. The increasing complexity of these computational aspects has compounded the need for these systems to remain dependable [9]. For such systems, the ability to tolerate permanent crash faults is integral to their dependable operation.

Conventionally, fault tolerance is achieved by replicating hardware and often using a voting mechanism to determine the output [32]. Unfortunately, this approach is extremely inefficient in terms of cost, weight, space and energy needs for many applications. This is especially true for the automotive industry, where the system reliability requirements can be diverse and the cost constraints are stringent. For example, consider self-driving cars. Several levels of automation (1 to 5) have been defined in [31] to characterize the spectrum of self-driving features. To put such systems in context with redundancy requirements, consider a Level 2 system active on highways only. In such a system, although the driver is not in direct control of the vehicle motion, the driver still plays a supervisory role: the driver will be requested to take over control in case of any subsystem or component failure. In such a system, perhaps only a small subset of all software tasks need redundancy (e.g., braking and steering control, but not propulsion). Now, consider a Level-4 system active in the same operational domain (highways), where the vehicle itself is responsible to bring the system to a safe state in case of failures. Such high levels of automation impose more stringent fault-tolerance requirements in terms of the number of task replicas (or backups).

Similarly, other diverse needs are also evident from the fact that different tasks running in an automobile have different levels of safety criticality. For example, the braking control task is far more safety-critical than (say) a music playback task. This motivates the need for adaptive cost-optimized fault-tolerance solutions to reduce overall resource utilization. Hence, software fault-tolerance techniques like *active replication* [37, 16] or the *primary-backup approach* [6, 2] using hot and cold standbys are more applicable to systems like automobiles [19, 4].

The diversity in the reliability requirements for tasks in a system using software fault tolerance is captured by the *recovery time requirement* (RTR) of each task. The RTR specifies the number of consecutive deadlines of the primary task that a redundant task can afford to miss without the system being considered to have failed. The recovery time requirement for a task varies depending on its safety criticality. Tasks that are safety-critical have a strict (and very low) upper bound on RTR, while others can afford more relaxed values. The *recovery time* is the time a redundant task takes to successfully take over execution on primary task failure. It is influenced by a number of factors like redundant-task type, redundant task priority and network delays. The goal of this paper is to analyze the recovery times achieved by different types of redundant tasks (active/passive) used in software fault-tolerance techniques for real-time systems. The major contributions of this paper are as follows:

1. We derive the bounds on the recovery time of different types of redundancies, i.e, active or passive, used in software fault-tolerance techniques for real-time systems.
2. We derive conditions to map the recovery time requirements of a task to a redundant-task-type assignment.
3. We propose heuristics to determine redundant-task-type assignments and allocate these tasks to different nodes satisfying the recovery time requirements of all tasks while attempting to optimize resource utilization.
4. We apply the Simulated Annealing method to the fault-tolerant task allocation problem and compare its performance to the heuristics proposed.

The rest of this paper is organized as follows. In Section 2, we describe related work. In Section 3, we define our system model and fault model, and describe different types of redundant tasks we consider for our analysis. In Section 4, we quantify recovery time and derive bounds on the recovery time for each redundancy type. In Section 5, we derive conditions to assign a redundancy type to a task given its recovery-time requirements. In Section 6, we present heuristics to assign tasks to nodes satisfying the recovery time requirements of each task. We also apply the simulated annealing method to the fault-tolerant task allocation problem and compare its performance against the proposed heuristics. We summarize and conclude our findings in Section 8.

2 Related Work

The problem of supporting fault tolerance at the level of task scheduling has been widely studied in the literature. A number of real-time task allocation algorithms in order to tackle this problem in a distributed real-time system [8, 13, 27] have been described in the literature. In [26], Oh et al. present an online allocation heuristic to assign replicas to a minimum number of processors such that all replicas guarantee that task deadlines are met. They also derive the bound on the number of processors required to feasibly schedule a task set using their heuristic. These approaches focus only on active replication, where the redundant software executes regardless of failure modes. The resource consumption of such approaches is impractical for resource-constrained systems like cars, especially as the level of automation increases and multiple failures need to be tolerated. In contrast, we also focus on the primary-backup approach which enables fault-tolerance solutions with optimized resource usage by activating some backups only when failures occur.

Fault-tolerant task allocation using a combination of active replication and the primary-backup approach has been studied in [15] and [3]. Both techniques introduce phasing delays to support backup overlapping and backup deallocation techniques. Neither technique leverages the lower run-time utilization of different types of passive backups to optimize the number of processors used for deployment. Also, all techniques mentioned so far attempt to meet immediate recovery-time requirements. In this paper, we allow each task to specify its own configurable recovery time requirement.

In our previous work [4, 19], we discussed the fault-tolerant task allocation problem which states that no task should be placed on the same node as its primary or another redundant task. We proposed the Tiered Placement Constrained Decreasing (TPCD) and TPCD with cold standby (TPCDC) heuristics to produce allocations respecting this fault-tolerant placement constraint. Both these heuristics assume the type of redundant task to be used as inputs. In this paper, we attempt to determine this parameter given the recovery time requirement of each task. To this end, we present a recovery-time analysis framework, along with an extension to the TPCDC heuristic, TPCDC+R, and two new heuristics to produce a fault-tolerant allocation satisfying the recovery-time requirement of each task.

3 System Model and Problem Definition

3.1 Computation Model

In this paper, we consider a distributed system consisting of N computational nodes, where each node can communicate with every other node in the system by sending messages. We assume a set of n tasks, $(\tau_1, \tau_2, ..., \tau_n)$, where each task is assigned a unique priority based on the Rate-Monotonic Scheduling (RMS) [23] Policy. We assume that the tasks are ordered in non-increasing order of priorities. We assume that a higher-priority task can immediately preempt a lower-priority task. Each task τ_i is assumed to have a worst-case execution time $(WCET)$ of C_i, a period of T_i and an implicit deadline $D_i = T_i$. The analysis can be adapted to other scheduling policies and deadline models (e.g., $D < T$), as long as a response-time analysis is available. Each task τ_i may be blocked by lower-priority tasks for at most B_i units of time as a result of the operation of a concurrency control protocol like the Priority Ceiling Protocol [33]. We assume that the worst-case release jitter, the worst-case time a task τ_i can spend waiting to be released after arrival, is J_i [36].

The schedulability of a task can be evaluated using the response-time analysis presented in [36].

$$\begin{aligned} r_i^{n+1} &= C_i + B_i + \sum_{j=1}^{i-1} \lceil (r_i^n + J_j)/T_j \rceil C_j \\ r_i^0 &= \sum_{j=1}^{i} C_j \end{aligned} \tag{1}$$

Equation (1) represents an iterative solution which starts at r_i^0 and terminates when either $R_i = r_i^{n+1} = r_n$ or $r_i^{n+1} > D_i$. We refer to R_i as the worst-case response time for task τ_i. R_i is measured from the instant the task is released to its completion. The worst-case time from arrival to completion of task i [36], also known as the worst-case completion time ($WCCT$), is given by,

$$WCCT_i = r_i + J_i \tag{2}$$

A task is said to be schedulable if its $WCCT_i \leq D_i$.

3.2 Fault Model

In our model, we are primarily concerned about permanent crash faults [9]. Hardware failures, operating system crashes and process crashes are some examples of crash faults. We assume that these crash faults are fail-silent [5]. In order to tolerate these crash faults, we employ fault tolerance by replication [14]. We consider three types of redundancies:

1. **Active Replica:** In active replication, all redundant copies are identical and treated uniformly. Each replica performs all operations, like accepting and processing application inputs, performing state calculations, performing application calculations and producing output. This implies that, under normal operation, the system needs to support duplicate suppression to filter out duplicate outputs.
2. **Hot Standby:** A hot standby is based on the primary-backup approach. It performs all the operations of the primary task except for producing outputs. On detection of primary failure, the hot standby is promoted to become a primary and begins to produce outputs. Unlike an active replica, a hot standby can run a degraded version of the primary to optimize resource consumption.
3. **Cold Standby:** A cold standby is also based on the primary-backup approach. It can be of two types depending on the type of application. If an application is *stateless*, the cold standby does not perform any operations until it detects primary failure. For applications with state, the cold standby accepts and logs application inputs but does not perform any other operations. It regularly accepts the state from the primary to maintain consistency. On detection of primary failure, the cold standby primes its state first and then begins to produce outputs.

Transient and intermittent faults can be overcome by techniques like simple re-execution, forward recovery [7], and recovery blocks [17]. The impact of these solutions can be accounted for by modifying the analysis of task response times to include additional fault-induced

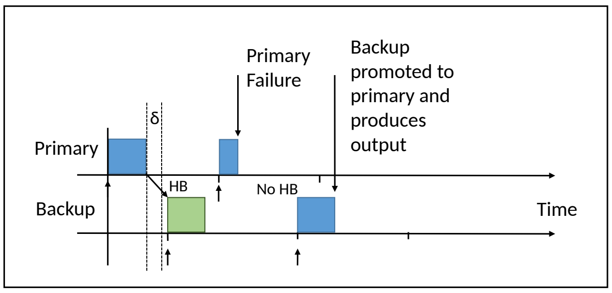


Figure 1 Detecting Primary Failure.

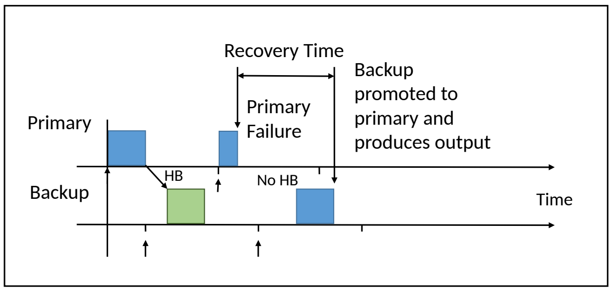


Figure 2 Defining Recovery Time.

processing requirements [7]. In this paper, we focus on permanent faults, though the analysis for transient faults from [7] can be incorporated into our framework. Similar fault models have also been used in the automotive sector [19] and [22].

In our model, a task and its replicas have the same period and a task can be assigned one or more replicas based on the application requirements. The system designer can decide which tasks are considered *critical* for the application and which are considered *non-critical*. In this paper, we assume that non-critical tasks do not have replicas and can be terminated in order to allow a cold standby to execute when a primary fails. For fault detection, we assume that the replicas monitor the status and health of the primary, for example, by using heartbeats and producing outputs when necessary [19, 4]. This is illustrated in Figure 1. We assume that the underlying communication framework is reliable[1], i.e., it guarantees that a message will either be delivered within a fixed message delivery bound δ or not be delivered at all. Common communications protocols like CAN/CAN-FD [10], FlexRay [28] and many variants of real-time Ethernet [1, 34] can support these guarantees. The successful reception of a heartbeat indicates to the replica that the primary is operational.

3.3 Problem Statement

▶ **Definition 1.** *Recovery time* (*RT*) is the time elapsed from the instant of primary failure to the instant when a redundant task is able to produce the desired output. This duration is shown in Figure 2.

[1] Safety-critical real-time systems must deal with communication failures. The communication layer can utilize solutions like redundant CANbus links, dual FlexRay configurations with built-in support for fault tolerance, and replicated ethernet switches. In the interests of brevity, we abstract away the details of such solutions with our assumption of a reliable communication layer in this paper.

The choice of the type of redundant task to be used has a major impact on a task's recovery time. For example, an active replica can virtually provide seamless recovery since it runs alongside the primary. The hot and cold standbys, on the other hand, have to first detect primary failure. In addition, the cold standby needs to then prime its state, which results in an even longer recovery time.

The number of redundant copies assigned to each task is also an important design parameter. Every task can be assigned m ($m \in \mathbf{N}$) redundant copies. It is important to note that different tasks can utilize different redundancy types (i.e., active, hot or cold). The number of replicas and their types are system parameters which are application-specific. Their choice determines the number of failures a given task can tolerate and how quickly a task can recover from a failure. The former is a system designer's choice and the latter can be captured by specifying a recovery-time requirement for each task.

▶ **Definition 2.** *Recovery time requirement* (RTR) is the maximum number of consecutive deadlines of the primary task that the system can afford to miss before the redundant task must recover in accordance with Definition 1.

We first determine which type of redundant task is appropriate for a given task to meet its recovery-time requirement. The benefit of using replicas is maximal when a task and any of its redundant copies obey the placement constraint of not being co-located on the same node. To this end in [20], Kim et al. defined the *Fault-tolerant Partitioned Scheduling* problem as one of assigning independent tasks to nodes where every member of a group, i.e., a primary task and its copies, would not be co-located on the same node. This ensures that, when nodes fail independently, they do not result in application failures. The bin-packing problem [25] of allocating fault-tolerant tasks is known to be NP-hard [18], and heuristics were proposed in [4] to address this problem. In this paper, we extend these heuristics to ensure that the recovery-time requirements of tasks are also satisfied.

We assume that task I/O dependencies[2] and ensuring input consistency between a primary and its redundant copies are considered by the system designer in assigning the RTR of each task in the system.

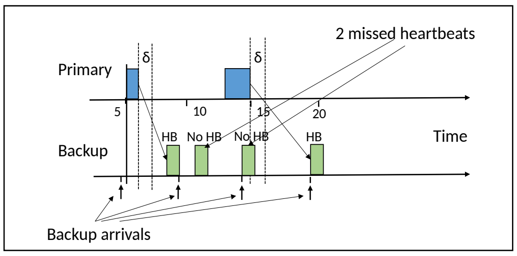
To summarize, the goals of this paper are as follows. Given N nodes, and a task set $\tau = \{T_1, T_2, \ldots, T_n\}$, where every task has an application-dependent recovery-time requirement RTR_i,

1. derive bounds on the recovery time for each redundant-task type,
2. decide a redundant-task type, i.e., active, hot or cold, and
3. find an allocation where all tasks satisfy their recovery-time requirements while minimizing the number of nodes used for allocation.

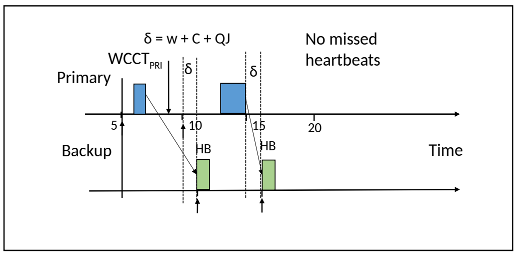
4 Recovery Time Analysis for Passive Backups

In the previous section, we saw that an active replica can be seamlessly recovered from, since other replicas are running in parallel. In this section, we derive the recovery time bounds for hot and cold standbys.

[2] Detailed task models capturing I/O dependencies are certainly needed, and will be part of our future work. For example, task I/O dependencies can be factored into our analysis by constructing composite (virtual) tasks formed by combining tasks with I/O dependencies.



(a) Backup not following the Primary.



(b) Backup following the Primary.

Figure 3 Motivation for Backup following the Primary.

4.1 Backup Following the Primary

Previous work [4] has shown that the bounds on the recovery time for passive backups can be reduced if the backup task execution follows the execution of the primary. The intuition for this can be seen in Figures 3a and 3b. As seen in Figure 3a, if the backup can execute at any time independent of its primary, it is possible for a backup to miss up to two heartbeats without primary failure. Hence, the backup must wait for three consecutive missed heartbeats to declare failure of the primary and initiate recovery, resulting in a longer recovery time. In contrast, when the backup follows the execution of the primary, it needs only a single missed heartbeat to detect primary failure.

For the backup to follow the primary, the following requirements must be satisfied:

1. *Global Time Synchronization*: To ensure that the backup follows the primary, the release time of the backup w.r.t that of the primary must be explicitly controlled. Since fault-tolerant task allocation requires primaries and replicas to run on distinct nodes, the nodes must be time-synchronized. This constraint can be relaxed in a system which allows tasks to be released with offsets at boot up and has negligible clock drift.
2. *Network Schedulability Analysis*: In order to calculate the optimal release instant for the backup, network delays must be characterized. [3] The worst-case network response time δ_m for message m can be represented as,

$$\delta_m = w_m + QJ_m + C_m \tag{3}$$

   where,
   - The queuing jitter QJ_m corresponds to the longest time between the initiating event and the message being queued, ready to be transmitted on the network.
   - The queuing delay w_m corresponds to the longest time that the message can remain in the device queue, before commencing successful transmission on the network.
   - The transmission time C_m corresponds to the longest time that the message can take to be transmitted. In the case of standbys, the transmission time depends on the standby type. Cold standbys need to accept state, and normally require longer transmission times than hot standbys.

4.2 Recovery Time Bounds for Hot Standbys

A hot standby produces an output immediately after it detects primary failure as described in Section 3 and shown in Figure 4. Let $WCCT_{pri}$ be the $WCCT$ for the primary. Let

[3] Popular automotive network technologies, like CAN [10] and FlexRay [28], have response-time analyses to bound the worst-case message delivery time.

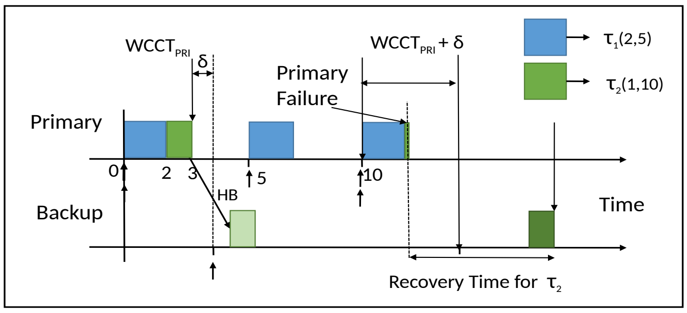


Figure 4 Recovery Time Bounds for Hot Standby τ_2.

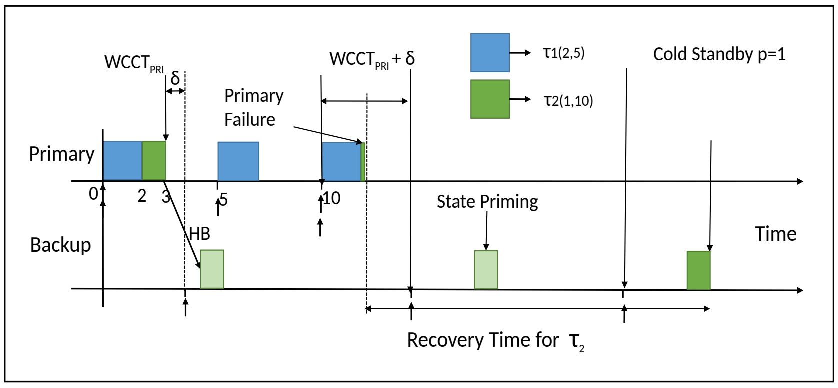


Figure 5 Recovery Time Bounds for Cold Standby τ_2.

$WCCT_{hot}$ be the completion time of the backup corresponding to the failure of the primary. The total time from the release of the primary to the execution of the backup would be $WCCT_{pri} + \delta_{hot} + WCCT_{hot}$. Hence, the recovery time is

$$RT_{Hot} = WCCT_{pri} + \delta_{hot} + WCCT_{hot} \tag{4}$$

4.3 Recovery Time Bounds for Cold Standbys

A cold standby takes longer to recover from a failed primary as described in Section 3, since it does not produce any state of its own, but instead receives regular state updates from the primary. This is illustrated in Figure 5. It only logs application inputs which it uses to prime state for future use. These logs can be cleared once a primary state is applied. Let p denote the number of periods the cold standby needs to prime state and produce output. A cold standby for a stateless application does not need to prime any state, and hence, in this case, $p = 0$. For applications with state, the value of p depends on two factors:

1. The frequency of state transfer from the primary to the standby: The higher the frequency of state transfer, the fresher is the state of the cold standby and hence lower is the number of periods required for state priming (i.e., a lower value for p).
2. Priming state is highly application-dependent. Some applications may make temporal corrections of the most recent state using appropriate extrapolations. Other applications may iterate through all the logged inputs between the last received state and the time instant the failure is detected, and, in each iteration, re-calculate the state to finally produce output based on fresh state. In this paper, we assume that, for applications with state, the value of p is provided by the application designer.

Thus, for a cold standby to recover from a primary failure, the recovery time would be

$$RT_{Cold} = WCCT_{pri} + \delta_{cold} + pT + WCCT_{cold} \tag{5}$$

5 Redundant-Task Type Assignment To Tasks

In this section, we identify the types of redundant task assignments that can satisfy a given RTR constraint. As described in Section 3.3 an active replica can be seamlessly recovered from, since other replicas are running in parallel, hence it can satisfy any RTR requirement.

5.1 Hot standby

5.1.1 $RTR = 0$

For a hot standby to recover from primary failure and maintain $RTR = 0$, the recovery time, RT_{hot}, should be less than or equal to T, i.e., the redundant task must recover before the deadline of its primary. Hence, from Equation (4) we have,

$$RT_{Hot} \leq T \quad \Rightarrow \quad WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq T \tag{6}$$

With the worst-case values for the terms in Equation (6),

$$T + \delta_{hot} + T \nleq T$$

Hence, with the worst-case values for $WCCT$, a hot standby cannot satisfy $RTR = 0$.

5.1.2 $RTR > 0$

Consider the case where $RTR = n$ and $n \in \mathbb{N}_{>0}$, allowing the task to tolerate up to n missed deadlines when the primary fails.

In the case of a hot standby, $RTR = n$ can be satisfied if $RT_{Hot} < (n+1)T$.
From Equation (4),

$$WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq (n+1)T \tag{7}$$

Considering $n \geq 2$ and the worst-case values for the terms in the above equation,

$$\begin{aligned} WCCT_{pri} + \delta + WCCT_{bkp} &\leq 3T \\ T + \delta + T &\leq 3T \end{aligned} \tag{8}$$

Assuming $\delta < T$, a hot standby can meet $RTR \geq 2$ (if it is schedulable).

5.2 Cold standby

5.2.1 $RTR = 0$

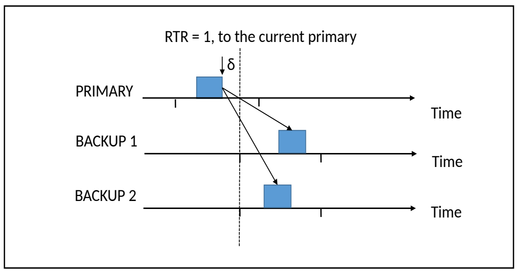
For a cold standby to satisfy $RTR = 0$, the recovery time should be less than or equal to T. From Equation (5),

$$WCCT_{pri} + \delta_{cold} + pT + WCCT_{cold} \leq T \tag{9}$$

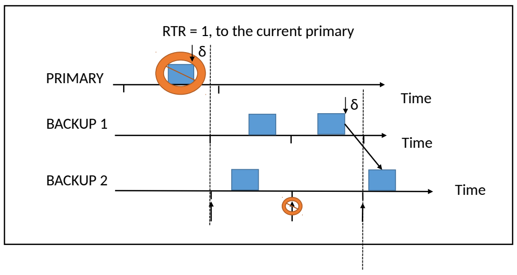
Equation (9) must be satisfied for a cold standby to meet $RTR = 0$. However, if $p \neq 0$, a cold standby cannot satisfy $RTR = 0$.

Table 1 Conditions for Redundant Task Selection.

Standby Selection		
RTR(n)	**Condition**	**Standby Assignment**
0	$WCCT_{pri} + \delta_{cold} + WCCT_{cold} \leq T$	Cold ($p = 0$)
0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq T$	Hot
0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} > T$	Active
> 0	$WCCT_{pri} + \delta + WCCT_{bkp} + pT \leq (n+1)T$	Cold
> 0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} \leq (n+1)T$	Hot
> 0	$WCCT_{pri} + \delta_{hot} + WCCT_{hot} > (n+1)T$	Active



(a) Multi-Level Backups.



(b) Release Time Correction.

Figure 6 Support for Multi-Level Backups.

5.2.2 $RTR > 0$

In the case of a cold standby, $RTR = n$ can be satisfied if $RT_{Cold} < (n+1)T$ in the worst case.

From Equation (5),

$$WCCT_{pri} + \delta + WCCT_{bkp} + pT \leq (n+1)T \tag{10}$$

Table 1 summarizes all the conditions above for standby selection. We see that, for certain conditions, multiple options are available for redundant-task type assignment. We describe our approach to redundant task selection in case of multiple available options in Section 6.

5.3 Multi-Level Backups

As shown in Figure 6a, a single primary can have more than one backup. Both backups in the figure are released such that they follow the primary to satisfy the primary's recovery-time requirement. We assume that the order of promotion to primary is statically configured (which in practice is easily achieved by the use of configuration parameters, or using node IDs). Suppose that the first backup in Figure 6a is designated to take over execution first after primary failure. On primary failure, it is not guaranteed that the current second backup would always satisfy the recovery time requirements of the first which would now become the new primary. In order for the second-level backup to now satisfy the RTR of the new primary, the release time of the task needs to be corrected and this is shown in Figure 6b. Also, since we are delaying the release time of the task, and deadlines are therefore correspondingly postponed, the deferred start does not affect the overall schedulability of the task set [30].

Algorithm 1 TPCDC+R.

```
1: procedure TPCDC+R ($\Gamma = \{\tau_0^0, \tau_0^1, ....\tau_1^0...\tau_n^0, ...\}$)    ▷ ($\tau_j^i : j \rightarrow TaskId, i \rightarrow TierOrder$)
2:     for each task $\tau_j$ in $\Gamma$ do
3:         $\Psi_i \leftarrow \tau_j^i$    ▷ Create tiers consisting of tasks with redundancies of the same order
4:     for each tier $\Psi_i$ in $\Psi$ do
5:         Sort tasks in descending order of their utilizations
6:         for each task $\tau_i$ in $\Psi_i$ do
7:             Check recovery time to primary and assign redundant-task type
8:             Task Assignment($\alpha$) $\leftarrow$ BFD-P($\tau_i$)
9:     Apply lower run-time utilizations for cold standbys
10:    Allocate the tasks that do not have redundancies
11:    return $\alpha$    ▷ Return the task set assignment
```

6 Task partitioning with Recovery Time Constraints

In Section 3, we presented the fault-tolerant task allocation problem. We now extend this problem to include the constraint that every backup task satisfies the recovery-time requirement of the primary. Given our focus on resource-constrained environments, we present heuristics to address this problem while trying to minimize the number of processors used for allocation. Based on the recovery-time bounds of Section 4.2, we derived conditions to determine the standby type in Section 5. In this section, we look at how the redundant-task type assignment can be incorporated in the task allocation scheme to satisfy the recovery time requirement of each primary.

The TPCD heuristic [4] produces an allocation satisfying the fault-tolerant placement constraint while attempting to minimize the number of nodes used. TPCD breaks the task set into tiers based on the backup order to place members of a replica group as far away from each other in the task order as possible. This reduces the chances of a task facing a placement conflict. In each tier, TPCD arranges tasks in descending order of utilization values, since, members of larger groups have a greater probability of running into a placement conflict. TPCD then allocates the tiers from the highest-order tier to the lowest-order tier. The TPCDC heuristic extends TPCD to leverage lower cold-standby utilizations. Any non-critical task can be terminated in order to allow a cold standby to execute when a primary fails. TPCDC initially treats all standbys as hot standbys from a utilization standpoint.

6.1 The TPCDC+R Heuristic

We now extend TPCDC by introducing an explicit check for RTR. This TPCDC+R heuristic is shown in Algorithm 1. Before assigning a task to a node, we ensure that every task (primary or copy) on that node satisfies RTR constraints. In order to determine the recovery time of a redundant task, we must first assign the redundant-task type using Table 1. Since cold standbys at run-time have very low utilization values, it allows for an optimization where non-safety critical tasks can be assigned to processors with cold standbys which can be terminated in case the cold standby needs to take over primary execution. Hence, if multiple redundant task options are available, we prioritize cold standbys over hot standbys and active replicas because they are the most resource-efficient. Next, hot standbys do not normally produce outputs. Hence, the overhead for duplicate suppression is avoided and hot

Algorithm 2 TRTI.

```
1: procedure TRTI ($\Gamma = \{\tau_0^0, \tau_0^1, ....\tau_1^0 ... \tau_n^0, ...\}$)    ▷ ($\tau_j^i : j \rightarrow TaskId, i \rightarrow TierOrder$)
2:     for each task $\tau_j$ in $\Gamma$ do
3:         $\Psi_i \leftarrow \tau_j^i$    ▷ Create tiers consisting of tasks with redundancies of the same order
4:     for each tier $\Psi_i$ in $\Psi$ do
5:         Sort tasks in ascending order of RTR constraints
6:     for each task $\tau_i$ in $\Psi_i$ do
7:         Check recovery time to primary and assign redundant-task type
8:         Task Assignment($\alpha$) $\leftarrow$ BFD-P($\tau_i$)
9: Apply lower run-time utilizations for cold standbys
10: Allocate the tasks that do not have redundancies
11: return $\alpha$    ▷ Return the task set assignment
```

Algorithm 3 RTT.

```
1: procedure RTT($\Gamma = \{\tau_0^0, \tau_0^1, ....\tau_1^0 ... \tau_n^0, ...\}$)
2:     for each task $\tau_j$ in $\Gamma$ do
3:         $\Psi_i \leftarrow \tau_j^i$    ▷ Create tiers consisting of tasks of same RTR
4:     for each tier $\Psi_i$ in $\Psi$ do
5:         TPCDC+R($\Psi_i$)
6:     return $\alpha$    ▷ Return the task set assignment
```

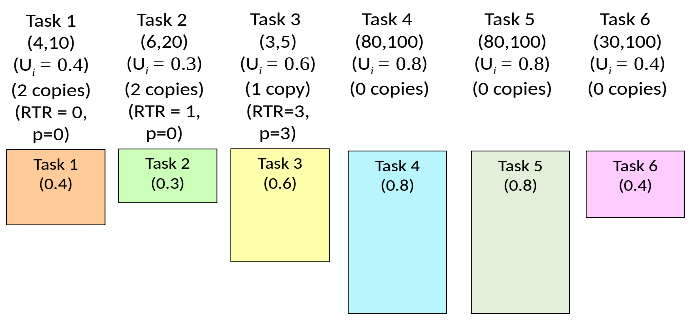
standbys can potentially run a degraded version of the primary with lower utilization values. However, they may have a scheduling penalty since they need to satisfy RTR constraints. Therefore, the heuristic first checks if the hot standby satisfies the RTR constraint of the task. If so, it assign a hot standby. Else, it chooses an active replica instead of opening a new node for assignment.

It must be noted that the choices among three redundant-task types would be different if the goal was different. For example, if communication bandwidth is constrained, the cold standby overheads for state transfer need to be factored in.[4]

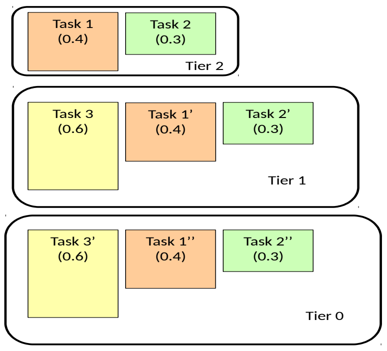
As stated before, we prioritize cold standbys over hot standbys or active replicas. Figure 8a shows the distribution of standby types produced by TPCDC+R. We plot the percentage of active, hot or cold redundant task assignments against the number of primary tasks in each task set. The results are averaged across 50,000 tasksets, where tasks are randomly generated. Each task is randomly assigned 0,1 or 2 redundancies, an RTR constraint from 0 to 5, and a value for p (i.e., periods for cold standby priming) from 0 to 5.

TPCDC+R prioritizes tasks with higher utilization values by assigning them first in the task allocation order for each tier. This introduces additional placement constraints for tasks which have tight RTR requirements. An example occurs when a task with low utilization with strict RTR requirements gets placed later in the allocation order. As a result, cold standbys may become unschedulable forcing the use of active replicas, which in turn can cause new nodes to be added.

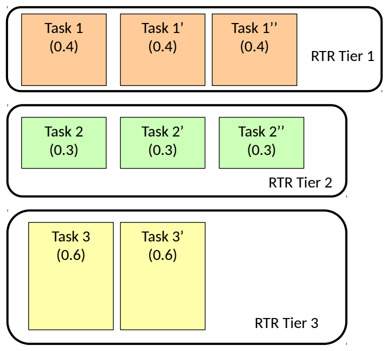
[4] We will consider this overall system resource optimization problem as part of our future work.


Task 1 (4,10) (U_i = 0.4) (2 copies) (RTR = 0, p=0)
Task 2 (6,20) (U_i = 0.3) (2 copies) (RTR = 1, p=0)
Task 3 (3,5) (U_i = 0.6) (1 copy) (RTR=3, p=3)
Task 4 (80,100) (U_i = 0.8) (0 copies)
Task 5 (80,100) (U_i = 0.8) (0 copies)
Task 6 (30,100) (U_i = 0.4) (0 copies)
Task 1 (0.4)
Task 2 (0.3)
Task 3 (0.6)
Task 4 (0.8)
Task 5 (0.8)
Task 6 (0.4)

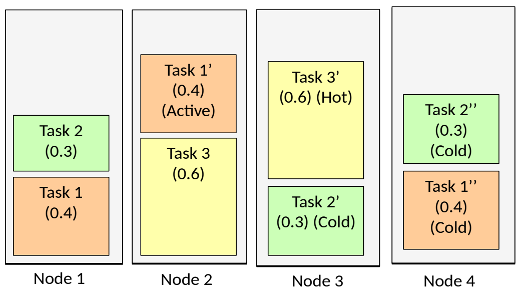
(a) Input Task Set.



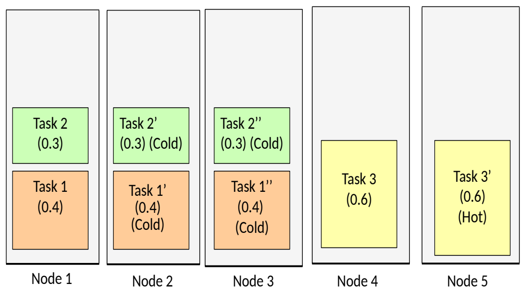
(b) TPCDC+R Tiering.



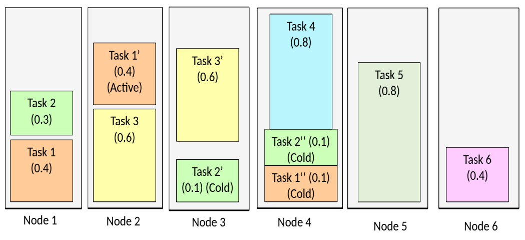
(c) RTT Tiering.



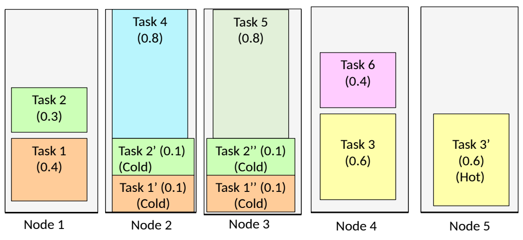
(d) TPCDC-R+ critical task allocation.



(e) RTT critical task allocation.

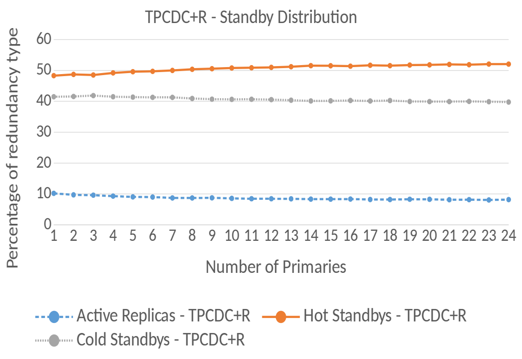


(f) TPCDC-R+ non-critical task allocation.

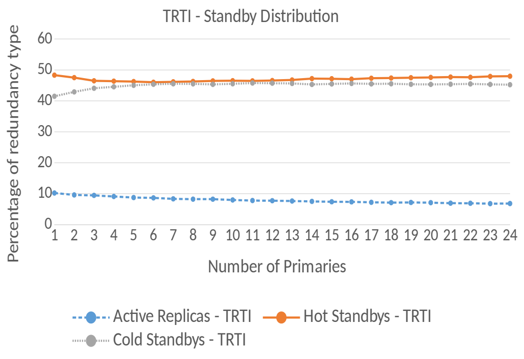


(g) RTT non-critical task allocation.

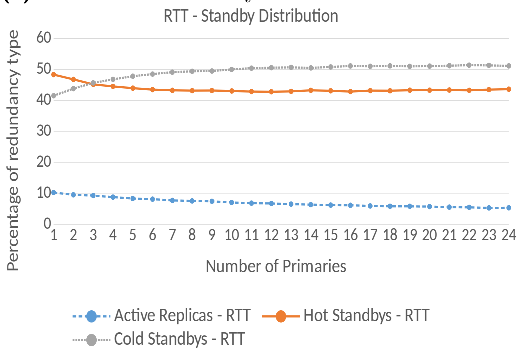
Figure 7 Example: TPCDC-R+ vs RTT (Best Viewed In Color).



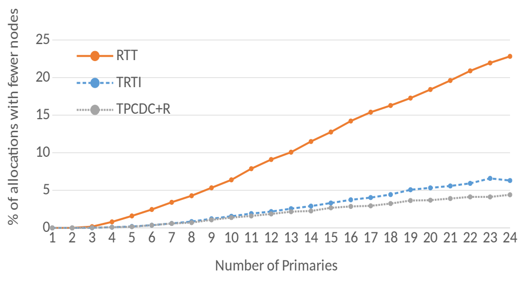
(a) TPCDC+R: Standby Distribution



(b) TRTI: Standby Distribution



(c) RTT: Standby Distribution



(d) Comparative Evaluation

Figure 8 Evaluation: RTT vs TRTI vs TPCDC+R

To address this problem, we introduce two new heuristics based on TPCDC+R that prioritize RTR constraints in the task allocation order.

1. In the first heuristic, we order tasks within each tier of TPCDC by their RTR requirements instead of utilization values. We refer to this extension as the *Tiered RTR constraint Increasing* (TRTI) heuristic. Algorithm 2 captures this TRTI heuristic.

2. In the second heuristic, we divide tasks into groups with different RTR requirements and allocate each group using the TPCDC heuristic separately. We refer to this as the *RTR-tiered* (RTT) heuristic. Algorithm 3 presents this heuristic.

Figure 7 depicts an example highlighting how prioritizing RTR constraints in the task allocation order can improve resource utilization by comparing the outputs of the TPCDC+R and the RTT heuristics for the input task set in Figure 7a. As shown in Figure 7b, TPCDC+R breaks the critical tasks into tiers based on the number of backups and orders tasks within a tier based on their utilization values. In contrast, RTT breaks the tasks into tiers based on their RTR constraints. Figures 7e and 7d show that the RTT heuristic allocates a greater number of cold standbys compared to the TPCDC-R heuristic. This, in turn, results in an allocation with fewer nodes as seen in Figures 7f and 7e. Notice that, when allocating the non-critical tasks, we consider the lower utilization values for the cold standbys.

Figures 8b and 8c show the standby distributions for TRTI and RTT heuristics. Both the heuristics result in a larger number of cold standby allocations than for the TPCDC+R heuristic.

6.2 Evaluation and Discussion

In this section, we evaluate and compare the performance of the TPCDC+R, TRTI and RTT heuristics. We also evaluate the impact of the increased cold standby allocation on the number of nodes used for allocations using the new heuristics. We plot the percentage of task sets for which a heuristic produces an allocation with fewer nodes, i.e., uses at least one node less for allocation compared to the other two heuristics. Figure 8d presents the results for 50,000 randomly-generated tasksets generated using Stafford's `Randfixedsum` algorithm [11] for total utilization values ranging from 0.1 to number of primaries and random period values ranging from 1 to 10^4. Each task is randomly assigned 0, 1 or 2 copies, an RTR constraint from 0 to 5, and a value for p (i.e., periods for cold standby priming) from 0 to 5. As the figure shows, RTT produces an allocation with fewer nodes on average when compared to the TRTI and TPCDC+R. For task sets with 24 primaries, it produces an allocation with fewer nodes than TRTI and TPCDC+R for almost 23% of the task sets. This is consistent with the intuition that increasing the number of cold standbys reduces CPU resource utilization. Also, as the number of primaries increase, this trend becomes more significant as we have more cold standby assignments to leverage. Moreover, both heuristics that prioritize the RTR constraints perform better than the TPCDC+R heuristic. It is important to note that increasing the number of cold standbys will result in additional network latencies since they need to have state information sent to them from their primaries. For the purpose of these experiments, we assume that the delays incurred for state transfer are short. For a network-constrained system, it may prove to be more advantageous to have a lower number of cold standbys.

7 Applying Simulated Annealing to the Fault-Tolerant Task Allocation Problem

In the previous section, we saw that the RTT heuristic on average produces a better solution than TPCDC+R and TRTI. In this section, we look at further improving on the RTT heuristic solution by utilizing the simulated annealing method to solve the fault-tolerant task allocation problem instead.

Simulated annealing is a general-purpose combinatorial optimization technique first proposed by Kirkpatrick et al. [21]. The fault-tolerant task assignment problem can be stated as an optimization problem as follows,

Given n tasks $(\tau_1, \tau_2, ..., \tau_n)$, with *utilization* $(u_1, u_2, ..., u_n)$, where $u_i \leq 1$, find the number of nodes M of size 1 that are needed to pack all tasks such that a primary task and its corresponding redundant copies obey the placement constraint of not being co-located on the same node and optimizing the following cost function [12]

$$cf = \text{Maximize} \quad \sum_{j=1}^{M} (\sum_{i \in k_j} u_i)^2 \tag{11}$$

where, k_j represents the set of tasks in bin j.

The simulated annealing algorithm for fault-tolerant task allocation is shown in Algorithm 4. The algorithm starts by using the RTT heuristic to create an initial allocation, α. We use this as the *initial state* of the system. To obtain a new state α' from the initial state we randomly perform one of the two operations described in Section 7.1. While performing either of these operations, we ensure that the placement constraints for all tasks remain satisfied. We also ensure that the new allocation is schedulable. Here, we apply a greedy

Algorithm 4 Simulated Annealing.

```
1:  procedure ANNEAL ($\Gamma = \{\tau_0^0, \tau_0^1, ....\tau_1^0 ... \tau_n^0, ...\}$)
2:      Task Assignment($\alpha$) = RTT($\Gamma$)
3:      $T \leftarrow T_\infty$
4:      while $T > T_0$ do
5:          repeat
6:              $\alpha$' = RANDOMLYMODIFYCURRENTSOLUTION($\alpha$)
7:              $\Delta C = cf(\alpha) - cf(\alpha')$                    ▷ From Eqn 11
8:              $\eta$ = RANDOM(0, 1)
9:              $P(\Delta C) = e^{(-\Delta C/T)}$
10:             if $\Delta C < 0$ or $P(\Delta C) > \eta$ then
11:                 $\alpha = \alpha'$
12:         until thermal equilibrium
13:         $T \leftarrow F(T)$
14:     return $\alpha$                         ▷ Return the task set assignment
```

optimization: if a valid operation results in an empty bin, we remove it from the allocation[5]. The value of the objective function is calculated for this new state. Let ΔC represent the change in the cost function, i.e, $\Delta C = cf(\alpha) - cf(\alpha')$. This state is unconditionally accepted if $\Delta C < 0$. If not, the Metropolis condition [24] is applied and the state is accepted with a probability according to the following acceptance function $P = e^{(-\Delta C/T)}$. We start with a large value for initial temperature $T = T_\infty$. When there is no appreciable change in the value of the cost function across a few chains of computation or a maximum number of iterations is reached, we lower the temperature. The annealing terminates when the temperature T reaches a low-enough value, T_o and the current best α is returned as the solution. We derive the values for T_∞ and T_o for the fault-tolerant task allocation problem in Section 7.2.

7.1 Generating Random Solutions

In order to create random solutions from a given solution, we apply the following two operations [29].

1. We randomly move a single task from a randomly-selected node k to another randomly selected node l.

   ▶ **Lemma 3.** *The maximum reduction ΔC_{max} for the cost function in Equation 11, for a system of two nodes, k and l, by moving a task from node k to l occurs when $U_k = 1$ and $U_l = 0$, where U_k and U_l are the total utilization values of the respective nodes.*

   Proof. Let u_t represent the utilization of the task that is moved from bin k to l. Let U'_k and U'_l be the transformed utilization values after a task is moved from node k to l. Hence, $U'_k = U_k - u_t$ and $U'_l = U_l + u_t$ and ΔC for this operation can be represented as,

$$\Delta C = {U_k}^2 + {U_l}^2 - {U'_k}^2 - {U'_l}^2 \quad \begin{aligned} &= {U_k}^2 + {U_l}^2 - (U_k - u_t)^2 - (U'_l + u_t)^2 \\ &= 2 * U_k * u_t - 2 * U_l * u_t - 2 * u_t^2 \end{aligned} \tag{12}$$

[5] In our experiments, we found no significant improvement in the quality of solutions obtained by retaining an empty bin

From Equation (12), ΔC is maximum when the positive terms are maximized and the negative terms are minimized. U_l only appears in the second term which is negative, and U_k appears only in the first term which is positive. Hence ΔC is maximized when $U_k = 1$ and $U_l = 0$ corresponding to their maximum and minimum possible values. ◀

For the fault-tolerant task allocation problem, moving a task from one bin to another can result in a different redundant-task-type assignment resulting in different run-time utilizations. Let the factor s capture this utilization change. The associated change in the cost function for this operation is given by,

$$\begin{aligned}\Delta C =&U_k^2 + U_l^2 - [(U_k - u_t)^2 + (U_l + s * u_t)^2] \\ =&2 * U_k * u_t - u_t^2 - 2 * U_l * s * u_t - (s * u_t)^2\end{aligned} \tag{13}$$

From Lemma 3, the maximum value of ΔC, which represents the largest reduction in the cost function, occurs when a task is moved from a completely-packed node to a completely-empty node. Since we apply a greedy optimization of removing empty bins, we consider $U_l = \epsilon$. Hence,

$$\Delta C_{max1} \approxeq 2 * u_t - u_t^2 - (s * u_t)^2 \tag{14}$$

2. We randomly select two tasks currently located in two different bins and swap them.
▶ **Lemma 4.** *The maximum reduction ΔC_{max} for the cost function in Equation 11, for a system of two nodes, k and l, by swapping two tasks occurs when one of the nodes has $U = 1$ and the other has $U = \epsilon$.*

Proof. Let U_k and U_l be the total utilization values of the respective nodes. Let u_{t1} represent the utilization of the task that is moved from bin k to l and u_{t2} represent the utilization of the task that is moved from bin l to k. Let U'_k and U'_l be the transformed utilization values after the tasks are swapped. Hence, $U'_k = U_k - u_{t1} + u_{t2}$, $U'_l = U_l + u_{t1} - u_{t2}$ and ΔC for this operation can be represented as,

$$\begin{aligned}\Delta C &= {U_k}^2 + {U_l}^2 - {U'_k}^2 - {U'_l}^2 \\ &= {U_k}^2 + {U_l}^2 - (U_k - u_{t1} + u_{t2})^2 - (U_l + u_{t1} - u_{t2})^2 \\ &= 2 * U_k * u_{t1} - 2 * U_k * u_{t2} + 2 * u_{t1} * u_{t2} - {u_{t1}}^2 - {u_{t2}}^2 \\ &+ 2 * U_l * u_{t2} - 2 * U_l * u_{t1} + 2 * u_{t1} * u_{t2} - {u_{t1}}^2 - {u_{t2}}^2 \\ &= 2 * U_k * (u_{t1} - u_{t2}) - 2 * U_l * (u_{t1} - u_{t2}) - 2 * (u_{t1} - u_{t2})^2 \\ &= 2 * (U_k - U_l) * (u_{t1} - u_{t2}) - 2 * (u_{t1} - u_{t2})^2\end{aligned} \tag{15}$$

From Equation (15), ΔC is maximum when $U_k - U_l \approxeq 1$, since $0 < U_k, U_l \leq 1$. Since we are swapping tasks between two nodes, a node cannot be empty. Hence, ΔC is maximized when one node has $U = 1$ and the other $U = \epsilon$. ◀

For our fault-tolerant task allocation problem, let the factors s_{t1} and s_{t2} capture the utilization changes after the swap. The associated change in the cost function for this operation is given by,

$$\begin{aligned}\Delta C =&U_k^2 + U_l^2 - [(U_k - u_{t1} + s_{t2} * u_{t2})^2 + (U_l + s_{t1} * u_{t1} - u_{t2})^2] \\ =&2 * U_k * u_{t1} - 2 * U_k * s_{t2} * u_{t2} + 2 * u_{t1} * s_{t2} * u_{t2} - u_{t1}^2 - (s_{t2} * u_{t2})^2 + \\ &2 * U_l * u_{t2} - 2 * U_l * s_{t1} * u_{t1} + 2 * u_{t2} * s_{t1} * u_{t1} - u_{t2}^2 - (s_{t1} * u_{t1})^2\end{aligned} \tag{16}$$

From Lemma 4, the cost function is maximized when one bin has $U = 1$ and the other has $U = \epsilon$. Hence,

$$\begin{aligned}\Delta C_{max2} \approxeq & 2 * u_{t1} - 2 * s_{t2} * u_{t2} + 2 * u_{t1} * s_{t2} * u_{t2} - u_{t1}^2 - (s_{t2} * u_{t2})^2 + \\ & + 2 * u_{t2} * s_{t1} * u_{t1} - u_{t2}^2 - (s_{t1} * u_{t1})^2 \rightarrow U_k = 1, U_l = \epsilon\end{aligned} \tag{17}$$

Given a task set, the value of $\Delta C_{max} = max(\Delta C_{max1}, \Delta C_{max2})$ can be easily calculated by substituting actual values into Equations (14) and (17) for all combinations of tasks.

7.2 Selecting an Annealing Schedule

The annealing schedule is described by quantitative choices for the three parameters: the starting value of the temperature, T_∞, the stopping value of the temperature T_o, and the decrement function $F(T)$ which determines the profile of the temperature from the beginning till the end of the annealing process.

The starting temperature, T_∞, for a good annealing schedule, is usually determined by monitoring the acceptance ratio at each temperature. The upper bound for acceptance ratio a_h (the fraction of generated states that are accepted), is arbitrarily fixed at some high value such as 0.9 and the temperature is increased to a value where this acceptance ratio is achieved [29]. Given that we can calculate ΔC_{max} for a given task set, we can calculate the value T_∞, which can accommodate even the largest reduction in the cost function at high temperatures, as follows.

$$a_h = e^{(-\Delta C_{max}/T_\infty)} \Rightarrow ln(1/a_h) = \Delta C_{max}/T_\infty \Rightarrow T_\infty = \Delta C_{max}/ln(1/a_h) \tag{18}$$

Similarly, T_o can be calculated for the lower bound of the acceptance ratio a_l.

$$T_o = \Delta C_{max}/ln(1/a_l) \tag{19}$$

In our experiments, we also found that $F(T) = 0.9 * T$ works well for the problem at hand.

7.3 Evaluation

In this section, we compare the performance of the simulated annealing approach with that of the RTT heuristic. We plot the execution time of the simulation annealing approach and the RTT heuristic against the number of the primaries in the task set. Figure 9 presents the results averaged across 5000 randomly-generated task sets. Each task is randomly assigned 0, 1 or 2 redundancies, an RTR constraint from 0 to 5, and a value for p (i.e., periods for cold standby priming) from 0 to 5. Note that the Y-axis is in log scale. Our heuristics are faster than the simulated annealing approach by more than 2 orders of magnitude. We also plot the number of nodes utilized by each technique per iteration against the number of primaries in the task set. Figure 10 presents the results for 5,000 randomly-generated tasksets generated using Stafford's `Randfixedsum` algorithm [11] for total utilization values ranging from 0.1 to number of primaries and random period ranges from 1 to 10^4. As Figures 9 and 10 show, though the simulated annealing algorithm takes longer to complete, it produces an allocation with fewer nodes on average when compared to RTT. This approach can be used for generating offline static allocations and in other non-time-sensitive contexts. In contrast, our heuristics can be used for run-time admission control and other environments that are time-sensitive.

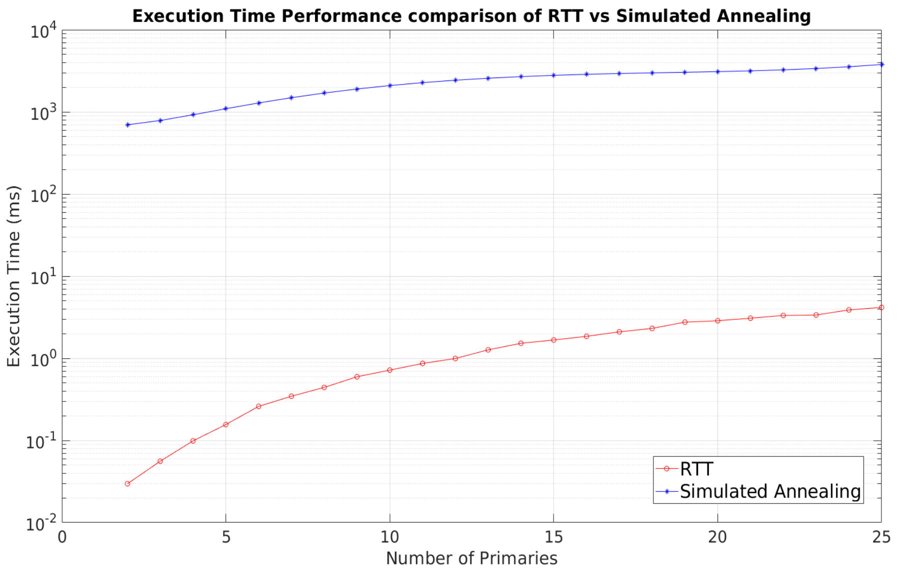


Figure 9 Execution-Time Evaluation.

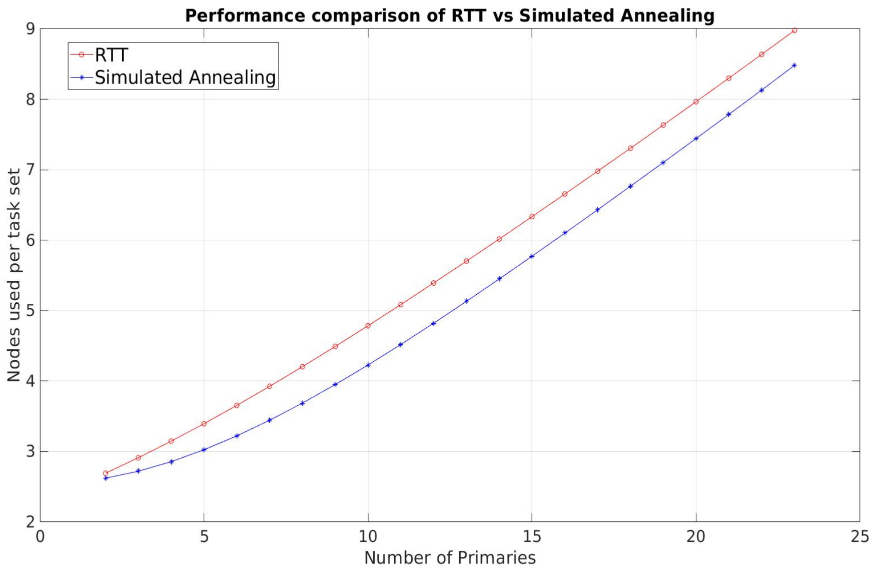


Figure 10 Resource Utilization Evaluation.

8 Concluding Remarks

In this paper, we considered software fault-tolerance techniques for safety-critical real-time systems and derived the bounds on the recovery time of different types of redundant tasks: active replication and primary backups with hot and cold standbys. We also derived conditions to map the recovery time requirements (RTR) of a task to a specific assignment of a redundant-task type. We extended the fault-tolerant task allocation problem to include these RTR constraints, and proposed the TPCDC+R heuristic to satisfy these constraints. Finding a core weakness in TPCDC+R, we then presented two additional heuristics called Recovery-Time Tiered (RTT) and Tiered Recovery-Time Constraint Increasing (TRTI) which prioritize the RTR constraints in the task allocation sequence. These two heuristics on average produce allocations with fewer nodes than the TPCDC+R heuristic because they yield more assignments of resource-efficient cold standbys. Overall, the RTT heuristic, which tiers tasks based on their RTR values to prioritize the allocation of tasks with strict RTR requirements first, performs the best. Finally, we used the simulated annealing method to solve the fault-tolerant task allocation optimization problem and showed that it produces allocations utilizing fewer computing resources than the proposed heuristics, at the cost of substantial run-time.

References

1 IEEE802.1cb-frame replication and elimination for reliability, howpublished = `http://www.ieee802.org/1/pages/802.1cb.html`, note = Accessed: 2018-01-12.

2 KapDae Ahn, Jong Kim, and SungJe Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 98–103, Dec 1997. `doi:10.1109/PRFTS.1997.640132`.

3 A. A. Bertossi, L. V. Mancini, and A. Menapace. Scheduling hard-real-time tasks with backup phasing delay. In *2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 107–118, Oct 2006. `doi:10.1109/DS-RT.2006.33`.

4 A. Bhat, S. Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 87–98, April 2017. `doi:10.1109/RTAS.2017.33`.

5 F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao. Implementing fail-silent nodes for distributed systems. *IEEE Transactions on Computers*, 45(11):1226–1238, Nov 1996. `doi:10.1109/12.544479`.

6 Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed Systems (2Nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. URL: `http://dl.acm.org/citation.cfm?id=302430.302438`.

7 A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 29–33, Jun 1996. `doi:10.1109/EMWRTS.1996.557785`.

8 J. J. Chen, C. Y. Yang, T. W. Kuo, and S. Y. Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 249–258, April 2007. `doi:10.1109/RTAS.2007.30`.

9 Jean claude Laprie and Brian Randell. Fundamental concepts of computer systems dependability. In *In Proceedings of the 3rd IEEE Information Survivability, Boston, Massachusetts, USA, October 2000*, pages 24–26, 2001.

10 Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, Apr 2007. `doi:10.1007/s11241-007-9012-7`.

11 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

12 Krzysztof Fleszar and Khalil S. Hindi. New heuristics for one-dimensional bin-packing. *Comput. Oper. Res.*, 29(7):821–839, 2002. `doi:10.1016/S0305-0548(00)00082-4`.

13 S. Gopalakrishnan and M. Caccamo. Task partitioning with replication upon heterogeneous multiprocessor systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 199–207, April 2006. `doi:10.1109/RTAS.2006.43`.

14 Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies — Ada-Europe '96*, pages 38–57, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

15 P. Guo and Z. Xue. Improved task partition based fault-tolerant rate-monotonic scheduling algorithm. In *2016 International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC)*, pages 1–5, July 2016. `doi:10.1109/SSIC.2016.7571812`.

16 K Hasimoto, Tatsuhiro Tsuchiya, and T Kikuno. Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems. *IEICE TRANSACTIONS on Information and Systems*, E85-D:525–534, 03 2002.

17 J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. *A Program Structure for Error Detection and Recovery*, pages 53–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985. `doi:10.1007/978-3-642-82470-8_7`.

18 David Johnson. Near-optimal bin packing algorithms. *Ph.D. Dissertation, MIT, MA*, 08 2010.

19 J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 227–236, Dec 2012. `doi:10.1109/RTSS.2012.74`.

20 J. Kim, K. Lakshmanan, and R. Rajkumar. R-batch: Task partitioning for fault-tolerant multiprocessor real-time systems. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1872–1879, June 2010. `doi:10.1109/CIT.2010.321`.

21 S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

22 Kay Klobedanz, Jan Jatzkowski, Achim Rettberg, and Wolfgang Mueller. Fault-tolerant deployment of real-time software in autosar ecu networks. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro C. Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification*, pages 238–249, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

23 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. `doi:10.1145/321738.321743`.

24 N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *jcp*, 21:1087–1092, jun 1953. `doi:10.1063/1.1699114`.

25 Dong-Ik Oh and T.P. Bakker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192, Sep 1998. `doi:10.1023/A:1008098013753`.

26 Yingfeng Oh and Sang H. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems*, 7(3):315–329, Nov 1994. `doi:10.1007/BF01088524`.

27 C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Fault-tolerant distributed deployment of embedded control software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):906–919, May 2008. `doi:10.1109/TCAD.2008.917971`.

28 Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1):205–235, Aug 2008. `doi:10.1007/s11241-007-9040-3`.

29 R.L. Rao and S.S. Iyengar. Bin-packing by simulated annealing. *Computers and Mathematics with Applications*, 27(5):71–82, 1994. `doi:10.1016/0898-1221(94)90077-9`.

30 Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, Mar 2004. `doi:10.1023/B:TIME.0000016129.97430.c6`.

31 Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems., .

32 C. Schonfeld. Redundancy approaches in spacecraft computers. In *28th Israel Annual Conference on Aviation and Astronautics*, pages 148–156, 1986.

33 L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990. `doi:10.1109/12.57058`.

34 D. Thiele, P. Axer, and R. Ernst. Improving formal timing analysis of switched ethernet by exploiting fifo scheduling. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015. `doi:10.1145/2744769.2744854`.

35 Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William "Red" Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. *Autonomous Driving in Urban Environments: Boss and the Urban Challenge*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. `doi:10.1007/978-3-642-03991-1_1`.

36 A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292(8), September 1993. URL: `http://digital-library.theiet.org/content/journals/10.1049/sej.1993.0034`.

37 Thomas Wolf and Alfred Strohmeier. Fault tolerance by transparent replication for distributed ada 95. In Michael González Harbour and Juan A. de la Puente, editors, *Reliable Software Technologies — Ada-Europe' 99*, pages 412–424, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems

Peter Wägemann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Christian Dietrich
Leibniz Universität Hannover (LUH), Germany

Tobias Distler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Ulbrich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Although internal devices (e.g., memory, timers) and external devices (e.g., transceivers, sensors) significantly contribute to the energy consumption of an embedded real-time system, their impact on the worst-case response energy consumption (WCRE) of tasks is usually not adequately taken into account. Most WCRE analysis techniques, for example, only focus on the processor and therefore do not consider the energy consumption of other hardware units. Apart from that, the typical approach for dealing with devices is to assume that all of them are always activated, which leads to high WCRE overestimations in the general case where a system switches off the devices that are currently not needed in order to minimize energy consumption.

In this paper, we present SysWCEC, an approach that addresses these problems by enabling static WCRE analysis for entire real-time systems, including internal as well as external devices. For this purpose, SysWCEC introduces a novel abstraction, the power-state–transition graph, which contains information about the worst-case energy consumption of all possible execution paths. To construct the graph, SysWCEC decomposes the analyzed real-time system into blocks during which the set of active devices in the system does not change and is consequently able to precisely handle devices being dynamically activated or deactivated.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases energy-constrained real-time systems, worst-case energy consumption (WCEC), worst-case response energy consumption (WCRE), static whole-system analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.24

Supplement Material ECRTS Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.4.2.7`, Source code of *SysWCEC* available at `https://gitlab.cs.fau.de/syswcec`

Acknowledgements This work is supported by the German Research Foundation (DFG), in part by Research Unit FOR 1508 under grant no. SCHR 603/14-2 (BATS), Research Grant no. SCHR 603/13-1 (Power-Aware Critical Sections), the CRC/TRR 89 Project C1 (Invasive Computing), and the Bavarian Ministry of State for Economics under grant no. 0704/883 25.

We thank Simon Schuster for the insightful discussions about Platin, Niklas Duda and Stefan Erhardt for the support with the energy-measurement setups, and Robert Burlacu for the help with improving the ILP solving.

1 Introduction

Energy-constrained real-time systems must not only ensure that tasks meet their timing deadlines, but also that there is enough energy to execute the tasks to completion [70, 72, 73]. Therefore, it is essential for energy-aware schedulers to consider both an upper bound for the execution time of a task as well as its worst-case response energy consumption (WCRE), that is, the maximum amount of energy required by the system to fully execute the task once it has been started [26, 70]. For systems where a task can be interrupted or preempted by other tasks with higher priorities, this means that a task's WCRE covers both the worst-case energy consumption (WCEC) of the task itself as well as the WCECs of all interrupt service routines and tasks that might be executed while the task is running[1].

Obtaining worst-case execution times can be regarded a solved problem for embedded, single-threaded real-time systems [5, 74] with multiple timing-analysis tools being commercially available [2, 53, 66]. Determining upper bounds for energy consumption, on the other hand, is still an open issue for systems in which devices and peripherals contribute to power consumption. Although energy profilers exist that are able to measure the energy consumption of systems including devices [62], so far there is no analyzer that provides reliable upper bounds for an entire system. Existing approaches to determine worst-case energy consumptions so far are usually limited to an analysis of the influence of a system's processor [35, 71]. Unfortunately, this strategy provides only a partial view of the problem, because in many embedded systems the processor is just one of several energy consumers besides internal devices (e.g., memory, timers) and external devices (e.g., peripherals such as WiFi transceivers, analog-to-digital converters, accelerometers, or LEDs). As illustrated in Table 1 by example of the NXP KL46z platform [23, 24] (ARM Cortex-M0+), a typical hardware for a small battery-operated real-time system, these devices in general significantly contribute to the system's overall power consumption. In some cases, for example, transceivers or LEDs, the power consumption of the device even exceeds the power consumption of the processor. Consequently, in order to obtain reliable results, it is crucial to take the impact of devices into account when analyzing a system's energy consumption.

The common approach to prevent WCRE underestimations for systems with devices is to assume that all the devices are active the entire time [43] and to include their combined power consumption into the analysis. Although this technique has the benefit of being sound, it also comes with the major drawback of usually leading to significant overestimations. These overestimations are caused by the fact that in many systems in practice devices and peripherals are disabled most of the time in order to save energy, and only temporarily switched on while their services are actually required, for example, to broadcast a message via a transceiver. As a consequence, WCRE analyses that assume all the devices to be always on often provide energy-consumption estimates that are much higher than the actual WCECs, which possibly leads to systems stopping execution unnecessarily early or to the system's lifetime being greatly underestimated by the pessimism of the analysis.

[1] We use the terms WCEC and WCRE analogous to timing analysis where the worst-case execution time (WCET) refers to a task in isolation and the worst-case response time (WCRT) to the timespan from the start of a task until its completion, including all possible interferences (e.g., preemptions).

Table 1 Power consumers in energy-constrained systems [23, 25, 67].

Hardware Unit	Power Consumption [@3.3V]
MCU (run mode)	5.6 mA
MCU (low-power run mode)	0.7 mA
MCU (stop mode)	0.3 mA
Accelerometer	1.7 mA
Analog–to-digital converter	0.4 mA
External memory (FRAM)	0.2 mA
LED	4.6 mA
WiFi transceiver	87.6 mA

The main reason why existing approaches deal with the impact of devices at a coarse-grained level is that WCRE analysis is inherently difficult in the context of devices that are dynamically switched on and off. Precisely determining the WCRE of a task requires knowledge about the entire system, including the WCECs of interrupt service routines and tasks with higher priorities. Additionally, the (de-)activation of devices, especially the activation of timers for running the CPU at a higher frequency, causes significant latencies that lead to energy-consumption penalties [8, 9]. In the absence of devices, obtaining the necessary WCECs is straightforward as the individual WCECs of all relevant routines and tasks can be analyzed in isolation from each other. However, in systems with devices this is not possible because, as we will show in detail, the WCEC of a task not only depends on the work performed by the task itself but also on the actions (i.e., device activations/deactivations) taken by other tasks, in some cases even tasks with lower priorities.

This paper presents *SysWCEC*, a static analysis approach that addresses the problem of determining WCREs in real-time systems with devices by taking the entire system into account. For this analysis purpose, *SysWCEC* first constructs and then leverages a novel data structure called the *power-state–transition graph*, which contains knowledge about the worst-case energy consumption of the analyzed system for all possible execution paths.

To construct the power-state–transition graph, *SysWCEC* in a first step searches for locations in the system code at which the power state of a device is changed and then logically decomposes the code into blocks of instructions during which the power states of devices remain constant. Next, *SysWCEC* identifies all possible interactions between the discovered blocks and combines this knowledge with additional information about the blocks' power consumptions and worst-case execution times. In the last step, this enables *SysWCEC* to determine all possible states the system might be in while it is running; in addition, for each of these states, this allows *SysWCEC* to compute the maximum amount of energy the system will consume while executing the instruction block associated with the state.

Decomposing the overall system into smaller blocks with constant device power states offers the key benefit of allowing us to perform large parts of the WCRE analysis without having to deal with varying power consumption while still being able to account for dynamic device (de-)activations. Apart from that, the context-sensitive analysis of both synchronous task interactions as well as asynchronous interrupts enables us to individually determine the WCEC of each task even for systems in which a task's WCEC cannot be analyzed in isolation as it might depend on the behavior of other tasks.

The *SysWCEC* approach presented in this paper borrows ideas from previous work on whole-system response-time analysis of fixed-priority real-time systems [19]. However, although lessons learned from timing analysis are helpful for energy-consumption analysis, in general, it is not possible to directly reuse existing techniques due to substantial differences between both domains. As we will show in this paper, energy-consumption analysis requires a more extensive system analysis that considers tasks of all priorities, addresses device (de-)activation penalties, and tracks power states of devices across all possible system states. Solving these problems forced us to develop a new approach to structuring real-time systems and their devices' power consumption to determine safe and accurate WCREs.

In summary, this paper makes the following contributions: (1) It presents our whole-system approach to WCRE analysis for real-time systems with internal and external devices and provides details on *SysWCEC*'s central data structure: the power-state–transition graph. (2) It gives insights into the open-source *SysWCEC* prototype, which supports the fully-automatic processing of OSEK-compliant (i.e., ECC1 [49]) real-time systems. (3) It discusses our evaluation of two different hardware platforms, which shows that *SysWCEC* is able to significantly reduce WCRE overestimations compared with the approach of assuming all devices to be always active.

2 Problem Statement

In this section, we first provide details on *SysWCEC*'s underlying system model and then discuss two open challenges that so far remain with regard to WCRE analysis: (1) precisely accounting for the fact that devices and peripherals in practical systems are dynamically switched on and off, and (2) determining task WCECs that depend on overall system state.

2.1 Hard- & Software System Model

SysWCEC targets embedded real-time platforms for which energy is a scarce resource. In such systems, the processor usually has a single processing core, a small predictable instruction cache, no data cache, and few pipeline stages [3, 4]. Due to the limited complexity, determining worst-case execution times based on the cycle costs of instructions in isolation is a feasible approach and achieves low overestimations [61]. Typically, the software running on such platforms consists of less than a dozen tasks that have fixed priorities and possibly depend on each other. A task is either synchronously activated by another task or a periodic alarm, or asynchronously activated as the result of a hardware interrupt. Interrupts always preempt the task currently running and can be released with a minimum inter-arrival time p_i, that is, there is an upper bound for the frequency with which interrupts are triggered.

Apart from the processor, systems in the targeted domain typically have numerous internal and external devices that significantly contribute to overall power consumption. While simple devices can only assume two different power states (i.e., `on` and `off`), more complex devices may comprise additional power modes, for example, to offer different tradeoffs between performance and power consumption. In each power mode, a device has a (mode-specific) maximum power consumption. Consequently, an upper energy bound E for an interference-free code sequence can be determined based on the worst-case execution time $WCET$ of the code using $E = WCET \cdot P_{max}$, with P_{max} being the total maximum power consumption of all hardware components in their current power modes. How to create a sound worst-case energy model to compute P_{max} is outside the scope of this paper. In general, the necessary information can be obtained from hardware analyses [50] and/or documentation [24].

In the targeted systems, transitions between power modes are initiated by the operating system as the result of a system call invoked by a task or an interrupt service routine; in this paper, we refer to such calls as *device system calls*, or *device syscalls* for short. Once invoked, a device syscall only returns after the requested power-mode switch is complete. All

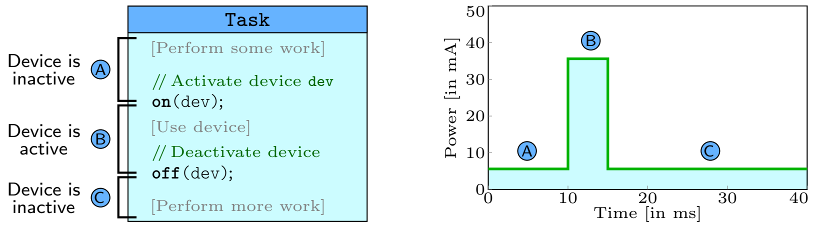


Figure 1 Effect of a temporarily activated device on power consumption.

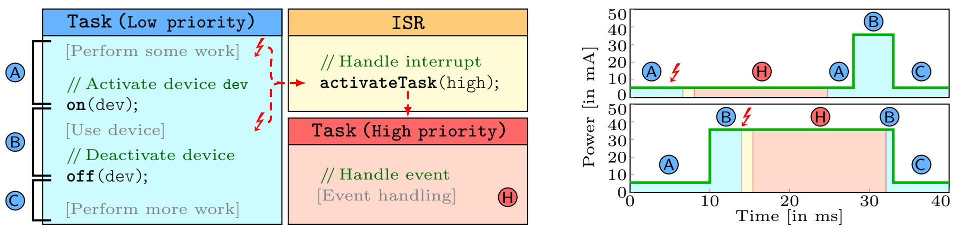
dynamic power management is explicitly controlled by the application via device system calls and passes the operating-system kernel. That is, as it is common for embedded platforms, the hardware does not initiate power-mode changes itself. To determine upper bounds for energy consumption for the whole system, the analysis cannot be limited to the user level but requires a whole-system view that includes both user application and the operating system.

2.2 Challenge #1: Modeling Temporarily Activated Devices

With devices being a decisive factor in an embedded system's power consumption, energy-constrained systems are usually designed to only keep a device active as long as its services are actually required. While on the one hand, this approach enables such systems to greatly extend their lifetimes, on the other hand, it also complicates static energy-consumption analysis because the power consumption of the system no longer only depends on the instructions executed but instead is also affected by the set of devices currently active. Figure 1 illustrates this problem for a task consisting of three parts: a first part in which the task performs some processing without using any devices (Part Ⓐ), a second portion in which the task temporarily activates and accesses a device (Part Ⓑ), and a third part that continues processing (Part Ⓒ). In the absence of devices, the worst-case energy consumption of a task can be statically determined based on the energy costs of individual instructions [35, 71]. However, for the example task this is not possible because the system calls to activate and deactivate the device, despite consuming only a small amount of energy themselves, significantly change overall power consumption due to modifying system state. Consequently, the energy consumption of the system for Part Ⓑ to a large extent is not a result of the processor executing certain instructions but of the fact that the device is active during this period of time. Our example therefore shows that for systems with devices it is insufficient to limit the energy-consumption analysis to the instructions executed and it explains why techniques that focus on the processor [35, 71] usually underestimate the worst-case energy consumption. Furthermore, many real-time scheduling approaches using DVFS disregard the latencies to switch on the timer devices for running on a higher frequency [8], which can be in the range of one millisecond [9].

In the context of worst-case energy-consumption analysis, the common approach to deal with devices is to prevent underestimations by modeling all devices in a system to be always on [43]. As our example in Figure 1 illustrates, this generally leads to significant overestimations due to assuming an increased power consumption that most of the time (e.g., during Parts Ⓐ and Ⓒ) is much higher than the consumption actually possible in practice.

Our Approach: To properly account for the energy consumption of internal and external devices, we identify parts of the system code during which the set of active devices does not change, starting a new part whenever a device is activated or deactivated. Performing our analysis at this granularity level allows us to minimize analysis complexity without losing the ability to model the impact of temporarily activated devices.



```
Task (Low priority)
  [Perform some work]
  // Activate device dev
  on(dev);
  [Use device]
  // Deactivate device
  off(dev);
  [Perform more work]
```

```
ISR
  // Handle interrupt
  activateTask(high);
```

```
Task (High priority)
  // Handle event
  [Event handling]
```

Figure 2 A task's WCEC may depend on another lower-priority task.

2.3 Challenge #2: System-State – dependent Task WCECs

While worst-case energy consumption (WCEC) analysis for systems with devices and peripherals is already challenging for a single task, the problem becomes even more difficult when entire task sets are involved. To illustrate this, we extend the example of Section 2.2: As depicted in Figure 2, we now assume that the task can be interrupted and that, as reaction to an interrupt, the interrupt service routine (ISR) activates a task with a higher priority, which executes a Part Ⓗ. In the following, we focus on discussing the difficulties associated with determining the WCEC of this higher-priority task.

As illustrated by the graphs in Figure 2, the power consumption of the high-priority task depends on the point in time at which the interrupt is triggered. That is, if the interrupt arrives while the system executes Part Ⓑ of the low-priority task, the high-priority task consumes much more power compared to the case in which the interrupt arrives during Part Ⓐ when the device is still inactive. However, in both cases the high-priority task executes exactly the same instructions (i.e., Part Ⓗ), which shows that the WCEC of the task does not only depend on the actions taken by the task itself but also on the state the system is in when the task starts executing, which is a result of previous actions taken by other tasks (i.e., device activations and deactivations). As shown by the example, this may even include actions taken by other tasks with lower priorities.

The fact that in systems with devices the worst-case costs of a task may depend on other tasks constitutes a major difference between timing analysis and energy-consumption analysis: To determine the worst-case execution time of the high-priority task, it is sufficient to analyze the task in isolation. Consequently, to compute an upper bound for the response time of a task (i.e., WCET plus potential interferences), an analysis only needs to consider the task itself as well as all interrupt handlers and tasks that might be executed while the task is running [5, 68]. In contrast, an analysis of worst-case (response) energy consumptions requires a comprehensive analysis of the entire task set, which means that existing timing-analysis approaches cannot be directly applied to analyze energy consumption in systems with devices.

Our Approach: Using a context-sensitive analysis that covers both synchronous task activations and asynchronous interrupts, we identify all possible states the analyzed system might reach during execution. By also analyzing the transitions between these states, we are able to determine the set of active devices for each of the states, which consequently allows us to precisely compute the WCECs of individual tasks.

3 The SysWCEC Approach

In the following, we present *SysWCEC*, a whole-system analysis approach to worst-case response energy consumption (WCRE). We tackle the challenges mentioned in Section 2 and tighten WCRE estimates by eliminating infeasible combinations of system-wide execution paths and energy states and thus abandon the all-always-on approach for external devices.

Overview

In a nutshell, *SysWCEC* leverages knowledge about device usage and operating-system semantics for a context-sensitive system-wide control-flow analysis that, in particular, incorporates state-dependent power consumptions. Conceptually, *SysWCEC* is based on the inference of the system's possible dynamic behavior and consequently all states the system might take during execution. By that, we mean the *system state* consisting of (a) active tasks and their priorities, (b) interrupt masks, (c) resource occupancy and ceiling priorities, and (d) power states of external devices. This knowledge allows for fine-grained modeling of extrinsic energy costs that can neither be attributed to individual instructions nor tasks but must be assessed in a system context, which is a fundamental advance of traditional techniques.

To infer the system states and thus disclose all energy-relevant interactions, our whole-system static analysis requires the following three steps, which we briefly outline next before immersing in further details in Sections 3.1 to 3.3.

1. **Abstraction and Decomposition:** In a first step, *SysWCEC* derives control-flow graphs of the entire system from the source code. To keep this step feasible, we take advantage of the fact that only system calls, or syscalls for short, can alter the system state and thus the power state and set of active devices. Consequently, our approach is to decompose the code into coarse-grained blocks that span between syscalls and thus are atomic from a system-state perspective: this coarsening facilitates the subsequent state enumeration and allows *SysWCEC* to perform large parts of the WCRE analysis efficiently without losing precision of dynamic device activations and deactivations.
2. **Power-Aware System-State Enumeration:** In the second step, *SysWCEC* explicitly enumerates all possible block-to-block transitions considering the priorities of tasks, synchronous task activations, and asynchronous interrupts. The result of this symbolic state enumeration is a state graph that incorporates the operating-system semantics and thus the possible dynamic behavior of the system. This, in particular, links the code blocks from the previous step with state-dependent power states and device activities.
3. **ILP Formulation & WCRE Determination:** In the last step, *SysWCEC* determines the worst-case energy consumption of each state-graph node based on the worst-case execution time of the associated code block and the respective power states of active devices. Furthermore, it constructs an integer-linear program (ILP) to eventually derive the WCRE.

3.1 Abstraction and Decomposition

Recalling our goal of a fine-grained, state-dependent modeling of energy consumption, we first need a global control-flow graph that, in particular, incorporates inter-task dependencies as well as the operating system. The canonical approach to this would be a full path analysis on a basic-block level. This granularity is, however, too fine and infeasible for the vast number of possible program paths through an entire system [10, 38].

Nevertheless, to determine the WCRE, besides scheduling events, we are only interested in energy-relevant events, that is, spots in the control flow that have the potential to change the power consumption. In other words, we can abstract from sequences of instructions that share a particular power and system state if executed uninterrupted.

We consequently based *SysWCEC*'s analysis on previous work on the concept of *atomic basic blocks* (*ABBs*) [22, 55] to abstract from the code's microstructure and decompose the system. An ABB is a control-flow superstructure that subsumes one or more basic blocks and conceptually spans between syscalls. Each ABB has exactly one entry and one exit

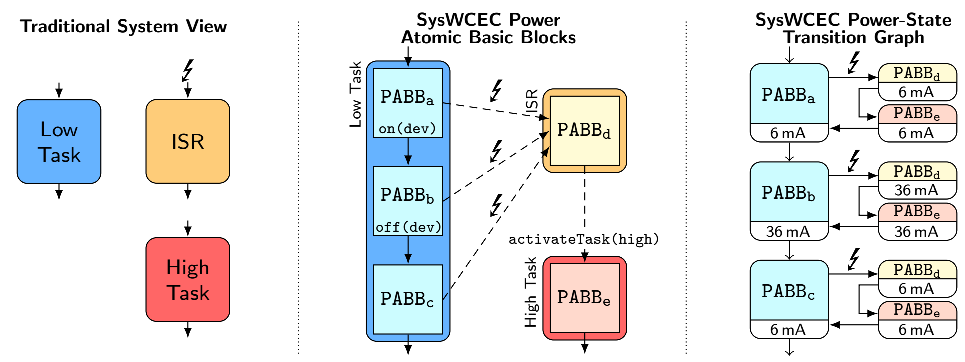


Figure 3 Illustration of *SysWCEC*'s first two analysis steps (decomposition & state enumeration).

block, which typically is the delimiting syscalls, forming a single-entry single-exit region. As long as the result complies with this rule, ABBs may be split arbitrarily for optimization reasons. These construction rules imply that an ABB executes *atomically* from a scheduling perspective. Still, there is no correlation between ABBs and power states.

Building on this foundation, for WCRE analysis we therefore developed the concept of *power atomic basic blocks* (*PABBs*) with the additional property that the set of active devices and their power states does not change within a block. With the operating system being the governor of power states and devices, this boils down to an extended analysis and decomposition of the implementation: any device reconfiguration is considered as a dedicated syscall, a device syscall (see Section 2.1). Consequently, a PABB is atomically executed from both the scheduling and power perspective. In the resulting PABB graph (i.e., coarse-grained global control-flow graph), changes in the system state (i.e., operating-system and power states) are possible only at the edges between PABBs. Note that the PABB graph covers the entire system implementation and all machine instructions. By that, *SysWCEC* inherently considers overheads (e.g., context switch, syscall, and scheduling costs), which are often neglected in real-time scheduling approaches [15].

Figure 3 illustrates the decomposition into PABBs using the example system from Section 2.3. Following the construction rules, the low-priority task is split by the device syscalls (`on`/`off`) into three PABBs, which account for the actual power states and the utilization of the external device. Consequently, only $\texttt{PABB}_\texttt{b}$ is modeled with active power state, while the computation in $\texttt{PABB}_\texttt{a}$ and $\texttt{PABB}_\texttt{c}$ is assigned the correct inactive power state. Here, the state modification is associated with the edges between the three PABBs. Similarly, the effect of scheduling-related syscalls is handled as inter-task constraints between PABBs. For example, the activation of the high-priority task from the ISR ($\texttt{PABB}_\texttt{d}$ to $\texttt{PABB}_\texttt{e}$). We further discuss the handling of asynchronous interrupts in Section 3.2.

Overall the decomposition into PABBs on its own is already a significant improvement over the all-always-on assumption. Our approach allows for an independent analysis of implementation artifacts and states, which has three main advantages that highly benefit the subsequent steps: First, it substantially reduces analysis complexity and allows *SysWCEC* to examine an entire system by identifying all possible states, without the need of enumerating all possible program paths. Second, the fact that the power consumption does not change within a PABB greatly facilitates the problem of determining upper bounds for the block's energy consumption. Third, the single-entry single-exit property allows the reuse of previously developed whole-system and timing analysis techniques [17, 18, 19].

3.2 Power-Aware System-State Enumeration

So far, the PABB graph only captures a static view of the system structure as well as the interactions between application, operating system, and external devices. Therefore, *SysWCEC* leverages the PABB graph in a second analysis step to deduce the dynamic system behavior by an explicit enumeration of all feasible system states and all transitions between them. A key aspect of this step is to further enrich the analysis by a model of operating system behavior (i.e., fixed-priority scheduling and resource protocols), the system configuration (e.g., task priorities, minimal inter-arrival times, and deadlines), and an energy cost model of the external devices. The resulting power-state–transition graph (PSTG) ultimately exposes the aspired context-aware global execution paths, including synchronous and asynchronous preemptions (i.e., task switches and interrupts). Thereby, we are subsequently able to identify and eliminate infeasible combinations of system-wide execution paths and power states and thus further refine the input for the final step in Section 3.3. Overall, the PSTG holds all relevant information to safely formulate an ILP, whose solution yields the WCRE. In the following, we detail the elements of the PSTG as well as its construction and show how to incorporate the operating-system semantics and the energy costs of devices.

Basic Principle and Operating-System Semantics

We begin our elaboration of the PSTG with its underlying principles and the operating-system–aware identification of possible execution paths. The basic construction rule is, as mentioned, to enumerate all possible system states and all transitions between them. A system state node is defined to hold the following information: (a) operating-system parameters, including the set of tasks with their current status (i.e., ready, running, suspended), priority, acquired resources, and resumption point as well as the ceiling priority. (b) Interrupt-related information including their status (i.e., enabled, pending, acknowledged). Finally, each state comprises (c) exactly one PABB and thread of execution, accordingly.

The construction algorithm starts with a dedicated entry state that is set up by the boot code. From this initial state, the application logic, which is obtained from the PABB graph, is simulated on a model of the operating system. At this point, the system configuration comes into play, which is used to instantiate the model to fit the concrete implementation. Subsequently, all reachable states are enumerated while the operating-system scheduling semantics are employed to discover inter-thread transitions. For example, when multiple tasks are runnable, the algorithm selects the task with the highest priority, and the follow-up state node references the task's entry PABB. Reconsidering the example system and its PSTG (see right part of Figure 3), the only successor of the interrupt is the runnable high-priority task. A transition to the low priority task is not possible in the fixed-priority scheduling model within this context-sensitive PSTG node. Moreover, tasks do not necessarily have their configured static priority due to shared resources and the employed priority-ceiling protocol [7] with its priority inheritance. Thus, tasks can have a dynamic priority, which is context-sensitively recorded in the task parameters of each PSTG node. Consequently, a PABB can occur multiple times in the PSTG with varying system states.

Handling Interrupts

Although the scheduler treats PABBs as atomic units, asynchronous interrupts can be released within a PABB's execution. At runtime, the interrupt could occur after every instruction and thus multiple times during the PABB execution. To handle such asynchronous preemptions, the PSTG construction algorithm inserts transitions from interruptible PSTG nodes to the entry functions of the ISRs (see the transition from lower-priority task to ISR in Figure 3). On the PSTG level, a single interrupt transition facilitates the state enumeration and is sufficient to enable compliance with the construction rules. We show in Section 3.3.2 how to bound the actual number of occurring interrupts in the final ILP formulation with help of the interrupt's minimum inter-arrival time and its response time from entry to return [19].

Power States and Energy-Consumption Costs

Finally, the PSTG's most distinctive feature comes into play: the inference of the current power state and the set of active devices. The associated power consumptions are taken from the given energy cost model. This combined information is crucial for modeling the WCEC of the individual system states during the ILP construction (see Section 3.3).

The power states are determined as part of the state enumeration: the construction algorithms memorizes the last power state when following transitions and updates the state (i.e., set of active devices) whenever it encounters a device syscall. Consequently, all possible succeeding nodes obtain the updated power state. In the same way, energy penalties (e.g., caused by mode changes) are incorporated at node transitions. Figure 3 illustrates the resulting power states as state-dependent consumption data. In this example, the ISR and the high-priority task are penalized with the additional power consumption caused by the device activation in the low-priority task only if the interrupt occurs within the device's operation period (i.e., within $\texttt{PABB}_\texttt{b}$). Note that, as with scheduling, the PSTG construction only eliminates infeasible states. Still, it contains all feasible combinations of execution and power state. Thus, WCEC estimation is the responsibility of the following ILP step.

Overall, the final PSTG incorporates by construction all possible execution paths of the concrete system under consideration of operating-system locks, scheduling, and interrupts as well as device usage and energy penalties. This modeling approach, like the PABB graph, represents a genuine simplification since it allows for independent handling of the system state and thus does not bloat the following ILP formulation unnecessarily. In Section 3.4, we provide details on our analysis framework and on how the executable system is generated, which behaves identically to the PSTG's analysis model [16].

3.3 ILP Formulation & Determining the WCRE

In the following, we describe how *SysWCEC* formulates an ILP to determine the WCRE of the overall system based on the entire PSTG. Analyzing the WCRE of a particular task would require the same steps, but only consider a subgraph of the PSTG that spans from the task's release until its completion. Our approach is based on a sound extension [19] of the well-known and proven implicit path-enumeration technique [42, 52], which we adapt for whole-system worst-case energy-consumption analysis. Once formulated, the ILP can be solved with a mathematical optimizer to eventually compute the WCRE.

3.3.1 Integer Linear Program

The main idea behind the ILP produced by *SysWCEC* is to determine for each PSTG node v how often the system executes the node in the worst case and to connect this execution frequency $f(v)$ to the worst-case energy consumption $E(v)$ of the machine code corresponding to the node. For this purpose, we rely on the following objective function to maximize the

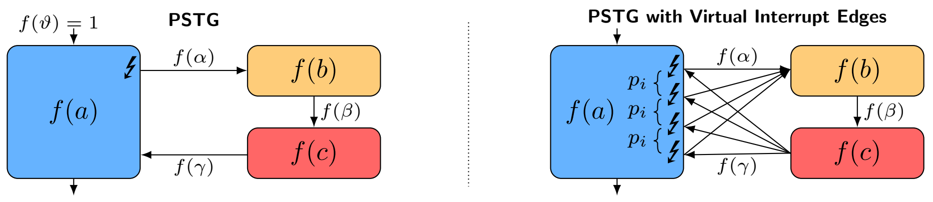


Figure 4 The ILP formulation derived from the PSTG accounts for the interrupts and potential switches to higher-priority tasks.

flow through the graph:

$$WCRE = max\Big(\underbrace{\big(\sum_{v \in V} E(v) \cdot f(v)\big)}_{\text{nodes}} + \underbrace{\big(\sum_{\varepsilon \in \mathcal{E}} E(\varepsilon) \cdot f(\varepsilon)\big)}_{\text{edges}}\Big)$$

Apart from nodes, the objective function also considers worst-case energy costs $E(\varepsilon)$ for edges $\varepsilon \in \mathcal{E}$ in the PSTG. This allows us to take energy costs into account that are caused by transitions between different power modes of a device and are a result of the fact that power-mode changes for some devices do not complete instantaneously.

To provide sound results, *SysWCEC* requires $E(v)$ and $E(\varepsilon)$ to be upper bounds of the energy consumptions of the PABB associated to node v and of the power-mode transition represented by edge ε, respectively. However, the *SysWCEC* approach does not make any assumptions on how these worst-case values are obtained which enables the reuse of existing WCEC analysis techniques [35, 50, 71]. One possibility to determine the WCEC for the PABB of a PSTG node v, for example, is to multiply the block's worst-case execution time $WCET(v)$ by $P_{max}(v)$, the maximum amount of power the system and its devices consume while the system is in the state represented by the PSTG node v; that is, $E(v) = WCET(v) \cdot P_{max}(v)$. As explained in Section 3.2, such knowledge about the power state of the system is part of the information maintained by *SysWCEC* in the PSTG and updated on each system-state transition. We discuss further refinements of this model in Section 6.

In addition to the objective function presented above, the ILP formulated by *SysWCEC* includes a set of constraints to specify dependencies between the execution frequencies $f(v)$ and $f(\varepsilon)$ of nodes and edges: (1) The entry and exit edge of the PSTG are each assigned a frequency of 1. (2) For each node v in the PSTG, the sum $f_{in}(v)$ of the execution frequencies of all incoming edges must be equal to the node's execution frequency $f(v)$ and must match the sum $f_{out}(v)$ of the execution frequencies of all outgoing edges; that is, $\forall v \in V : f(v) = f_{in}(v) = f_{out}(v)$. This constraint preserves the flow through the graph.

3.3.2 Handling Interrupts in the ILP

If an interrupt can occur within the execution of a PSTG node's PABB, the graph contains a single corresponding interrupt-transition edge. However, as the interrupt may be triggered more than once, in the ILP *SysWCEC* needs to consider the interrupt multiple times. Figure 4 illustrates this scenario for an example PSTG with three nodes: a node a depicting a low-priority task, a node b representing the asynchronous interrupt, and a node c referring to a high-priority task. With interrupts at most being released with a minimum inter-arrival time p_i (see Section 2.1), there is an upper bound $N \in \mathbb{N}_0$ for the number of interrupts that can occur during the execution of a system. In our example $N = 4$, which also represents the number of times the PABB of node a can be preempted and resumed. To bound N we use the following inequation based on T, the runtime of the system for the execution scenario in which the system achieves its worst-case response energy consumption:

$$p_i \cdot (N-1) \leq T$$

In a nutshell, this constraint expresses the fact that the longer the system runs (i.e., the larger T), the more interrupts N may be triggered. In the worst case, the first interrupt is released right at the start of system execution, with another interrupt following every p_i. To determine T, we combine the worst-case execution times $WCET(v)$ and $WCET(\varepsilon)$ of all PSTG nodes and edges and combine them with the execution frequencies $f(v)$ and $f(\varepsilon)$ of the ILP's objective function presented in Section 3.3.1:

$$T = \Big(\sum_{v \in V} WCET(v) \cdot f(v)\Big) + \Big(\sum_{\varepsilon \in \mathcal{E}} WCET(\varepsilon) \cdot f(\varepsilon)\Big)$$

Relying on the same execution frequencies that are used to compute the WCRE ensures that T actually represents the runtime of the scenario consuming the most energy.

Using the constraint presented above to bound the maximum number of interrupts, a solver is able to determine the WCRE of a system with interrupts, but may provide unnecessarily pessimistic results, as the following example based on Figure 4 shows: With the execution frequency of the graph's entry edge being $f(\vartheta) = 1$, the PABB of node a under any circumstance is only executed once; that is, $f(a) = 1$. However, applying the flow-preserving constraint, without further action, a solver would compute an execution frequency of $f(a) = f_{in}(a) = f(\vartheta) + f(\gamma) = 1 + N = 5$, accounting for the fact that the interrupt may resume up to N times. To prevent such overestimations, we differentiate between synchronous and asynchronous activations, which is knowledge that is already an attribute of the PSTG's transition edges (see Section 3.2). This approach allows us to consider and subtract the number of completed suspend-resume cycles when determining the execution frequency of a node v as follows: $f(v) = f_{in,sync}(v) + f_{in,async}(v) - f_{out,IRQ}(v)$, with $f_{in,sync}(v)$ and $f_{in,async}(v)$ being the execution frequencies of all incoming synchronous and asynchronous edges, respectively, and $f_{out,IRQ}(v)$ representing the execution frequency of all outgoing interrupt edges of the node. For the example system, this optimization reduces the execution frequency of node a to $f(a) = f(\vartheta) + f(\gamma) - f(\alpha) = 1 + N - N = 1$, and as a consequence correctly reflects the actual execution frequency of this node. In a similar way, *SysWCEC* is able to address interrupt preemptions that are not resumed, which can happen if the start and end point of the WCRE analysis are in different tasks. Note that in the example the described optimization only affects the execution frequency of node a. The execution frequencies of both other nodes still take the effects of multiple interrupts into account, resulting in frequencies of $f(b) = 0 + f(\alpha) - 0 = N$ and $f(c) = f(\beta) + 0 - 0 = N$.

3.4 Implementation

As shown in Figure 5, the *SysWCEC* toolchain relies on two main components: a modified version of the `dOSEK` framework [30] to construct the power-state–transition graph, and the `Platin` analysis toolkit [29, 51] to formulate the integer linear program necessary to determine the WCRE. Both the `dOSEK` system-analysis/-generation framework and the `Platin` timing-analysis toolkit are fundamentally based on the LLVM compiler infrastructure [40].

Provided with the specification of a real-time system and the implementation of tasks and interrupts, `dOSEK` is able to identify all possible system states and to automatically generate an executable and OSEK-compliant (i.e., ECC1 [49]) operating-system implementation. The conformance class ECC1 allows using prioritized, preemptible, self-suspending, and

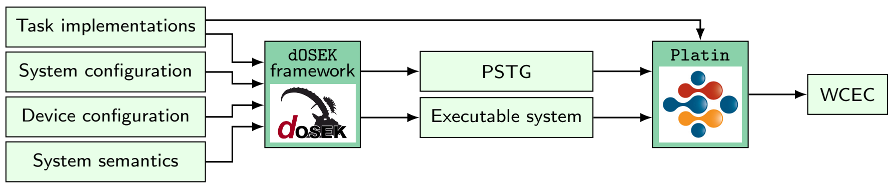


Figure 5 Workflow of the *SysWCEC* analyzer.

work-preserving tasks. Tasks can wait for specific events and can acquire resources, whereas a stack-based priority-ceiling protocol (PCP) avoids unbounded priority inversion [6]. `dOSEK` is able to perform the entire system-state enumeration considering also the dynamic priorities due to the PCP. For *SysWCEC*, we made extensive enhancements to implement the concepts of analysis, decomposition, and state enumeration described earlier. In particular, we introduced the notion of device syscalls, added means to supply `dOSEK` with information about the maximum power consumption of devices in different modes, and enabled the framework to track the modes of devices across different system-state transitions. As a result of our modifications, `dOSEK` now performs a whole-system state analysis that takes devices into account and puts out the results in the form of the power-state–transition graph.

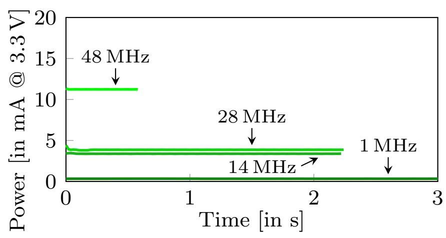
Providing `Platin` with the PSTG and the system implementation generated by `dOSEK`, we can use the toolkit to determine the WCET of PABBs. This allows us to compute the WCEC for each system state by multiplying the WCET of the associated block by the power-mode–specific maximum power consumption of all the devices that are active in the state. Based on this knowledge, `Platin` formulates the ILP for the WCRE bound that is then solved by the mathematical optimizer `Gurobi` [28].

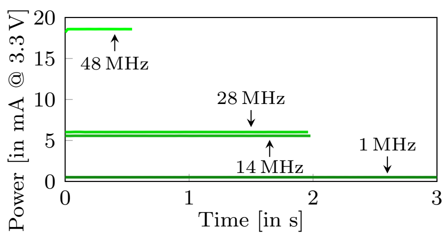
4 Evaluation

In this section, we experimentally evaluate the *SysWCEC* approach and its prototype. Our focus in this context does not lie on proving that the WCRE values determined by *SysWCEC* are actually upper bounds for response energy consumption. As discussed in Section 3.3.1, due to relying on proven analysis techniques *SysWCEC* delivers sound results by construction, provided that the worst-case energy model used for the analysis is accurate. Creating energy models with such properties is feasible [50] but outside the scope of this paper, which is why in our evaluation we concentrate on assessing the effectiveness of *SysWCEC* in comparison to existing analysis techniques. To obtain meaningful results, for this purpose we require an energy model that comprises realistic values for the power consumption of different hardware units, including devices and peripherals, which we can then use as input for *SysWCEC*. In Section 4.1 we describe how we compiled the energy model for our experiments. We do not claim this model to contain guaranteed upper power-consumption limits. Nevertheless, due to offering information on the characteristics of real-world hardware components, the model allows us to evaluate *SysWCEC*'s ability to deal with temporarily active devices (Section 4.2), its context-sensitive analysis (Section 4.3), as well as its scalability (Section 4.4).

4.1 Energy Model

In order to be able to evaluate *SysWCEC* with realistic power and energy consumption values of devices, peripherals, and processors, our energy model combines knowledge from different sources (e.g., manuals, measurements) and different hardware platforms.





Phase	Mean	Max. Upper Error	Standard Deviation (abs. / rel.)
CPU, 48 MHz	11.23 mA	0.01 mA	3.7 µA / 0.03 %
CPU, 28 MHz	3.86 mA	0.06 mA	14.61 µA / 0.38 %
CPU, 14 MHz	3.38 mA	0.08 mA	12.08 µA / 0.36 %
CPU, 1 MHz	0.33 mA	0.01 mA	10.19 µA / 3.09 %
Memory, 48 MHz	18.58 mA	4.32 µA	3.24 µA / 0.02 %
Memory, 28 MHz	6.03 mA	0.02 mA	3.20 µA / 0.05 %
Memory, 14 MHz	5.57 mA	3.69 µA	1.93 µA / 0.03 %
Memory, 1 MHz	0.51 mA	2.71 µA	1.04 µA / 0.20 %

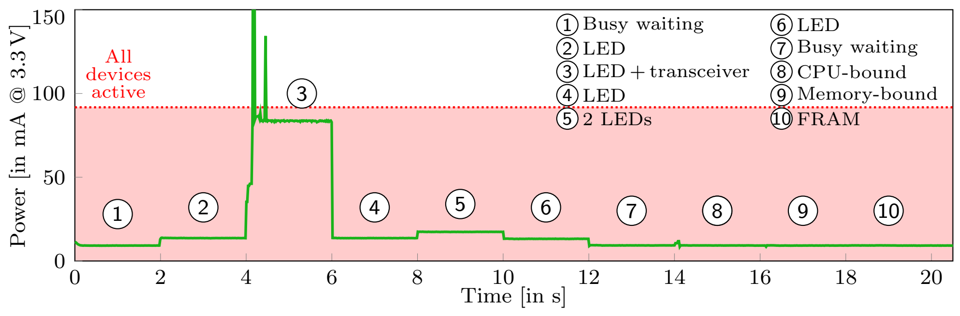
Figure 6 ARM Cortex-M4: Traces for CPU-bound (left) and memory-bound benchmarks (right).

Devices and Peripherals

Our first platform is an NXP FRDM KL46z evaluation board [23, 24] that features an ARM Cortex-M0+ core [4] with 256 KB of flash memory, 32 KB of SRAM, and a small cache (i.e., 4-way, 4-set program flash memory cache with a size of 64 B). We set up the evaluation board to run the execution pipeline at 48 MHz, the bus speed is 24 MHz. Apart from the processor, the board comprises a rich set of different devices including two LEDs, an accelerometer, a magnetometer, and an analog-to-digital converter. In addition, we attached an ESP8266 Wi-Fi module [67] as transceiver and an external ferroelectric RAM (FRAM) chip [25] as non-volatile storage. For most devices and peripherals, detailed documentation on the maximum power consumption is available (e.g., LEDs [46], accelerometer [48], magnetometer [47], analog-to-digital converter [24]). In all other cases, we obtain realistic power-consumption values by measurement relying on the source-measure unit Keithley 2612 [36], which is able to measure minimum currents of 100 fA and minimum voltages down to 100 nV at a temporal resolution of up to 20 µs. Using this source-measure unit circumvents the problem of potentially noisy power supplies and the problem of influencing the system under test with shunt-based measurement setups; both are known problems in the context of benchmarking low-power applications [21]. The results for our first platform are presented in Table 1.

Processor Power Modes

With our first platform's processor only offering a few power modes, we use a second platform with a more complex processor (i.e., an EFM32 Giant Gecko evaluation board [63, 65] with an ARM Cortex-M4) to examine the effects of different processor power modes. For measurements, in this case we rely on the board's integrated current-measurement circuitry, which is able to quantify currents from 0.1 µA to 50 mA and allows us to measure the power consumption of the microcontroller and correlate it to the code executed. As programs, we select a CPU-bound benchmark performing a prime-number calculation and a memory-bound benchmark repeatedly copying data, because these two categories represent the two ends



Phase	Mean	Max. Upper Error	Standard Deviation (abs. / rel.)
① Busy waiting	9.21 mA	0.09 mA	0.04 mA / 0.41 %
③ LED + transceiver	87.57 mA	220.84 mA	33.45 mA / 38.19 %
⑤ 2 LEDs	17.38 mA	0.12 mA	0.04 mA / 0.23 %
⑧ CPU-bound	11.25 mA	0.67 mA	0.59 mA / 5.25 %
⑨ Memory-bound	9.13 mA	0.28 mA	0.26 mA / 2.82 %
⑩ FRAM device	9.3 mA	0.09 mA	0.03 mA / 0.37 %

Figure 7 Measurement results for phases with different sets of active devices.

of the spectrum with regard to power consumption [12, 75]. Figure 6 shows the results and illustrates the impact of different processor power modes on execution time and power consumption. *SysWCEC* addresses this issue by differentiating processor power modes when analyzing a system, thereby modeling the processor in a conceptually similar way to devices.

Mode-Change Latencies

Hardware units not only consume energy while running in a certain power mode but also during the switch from one power mode to another. *SysWCEC* takes this fact into account by considering mode-change latencies and consequently attributing additional energy costs to power-mode switches. For most devices and peripherals of our two evaluation platforms, we found the associated mode-change overhead to be negligible, which is why in the following we focus on measurement results for the processor of our second platform. On the ARM Cortex-M4, switching from 28 MHz to 48 MHz, for example, takes 396 µs and comes with an energy overhead of 8.71 µJ. For comparison, the CPU-bound benchmark computing a 4-digit prime number on the same platform at 28 MHz requires 2 ms and 25.5 µJ. This shows that power-mode changes can have a significant impact on response time and energy consumption, although many energy-aware real-time scheduling approaches do not consider such overheads [8]. *SysWCEC*, on the other hand, includes time and energy costs for power-mode switches when analyzing worst-case (response) energy consumption.

4.2 WCRE Analysis for Temporarily Activated Devices

Using the energy model obtained in Section 4.1, in our first experiment we evaluate *SysWCEC* in the context of a system in which the set of active devices and peripherals constantly changes, as it is the case in a practical system that only activates devices temporarily in order to minimize energy consumption (see Section 2.2). The experiment runs on the Cortex-M0+ platform and, as shown in Figure 7, consists of phases with different sets of active devices, which results in varying overall power consumption. In each phase, we first execute a specific

Table 2 WCRE-estimate comparison between the all-always-on approach and *SysWCEC*.

Benchmark	WCRE All-Always-On	WCRE *SysWCEC*	Improvement
#1 Transceiver (w/o resource)	4,389.10 µJ	3,786.32 µJ	13.73 %
#2 Transceiver (w/ resource)	4,473.91 µJ	818.66 µJ	81.70 %
#3 Synchronous activation	2,266.28 µJ	1,236.15 µJ	45.45 %
#4 Asynchronous IRQ	399.88 µJ	335.41 µJ	16.12 %

event (e.g., a device activation) or a specific job (e.g., a computation) and subsequently delay execution for 2 ms. When the system activates the transceiver at the beginning of Phase ③, our results show power-consumption spikes that last for a short period of time. The spikes are an artifact of our current prototype hardware and in a practical system can be prevented by additional hardware circuitry [32]. In general, the power consumption within each phase behaves almost linear, as also confirmed by the small standard deviations (see Figure 7).

The measurements on the hardware platform for the Phases ② to ⑤ illustrate that activating and deactivating devices has a considerable impact on the power consumption of the whole system, especially in comparison to Phase ① when all devices are still switched off. This observation confirms that for an analysis of a system's (worst-case) energy consumption it is not sufficient to only take the processor into account, but crucial to consider all power consumers in the system. Although the power consumption of the processor varies depending on whether it executes a CPU-bound job (e.g., a prime-number calculation in Phase ⑧) or a memory-bound job (e.g., copy operations in Phase ⑨), the overall impact of the work performed by the processor is comparatively small.

Using the common approach to determine the WCRE for a system with devices, that is, to assume that all devices are always on (see Section 2.2), for the evaluated scenario results in a significant overestimation, as indicated by the red area in Figure 7. In contrast, by decomposing the system into parts during which devices do not change power modes, *SysWCEC* is able to provide much lower bounds, for example, due to being aware that the transceiver is only operating in Phase ③ and definitely remains inactive the rest of the time. For this experiment, the *SysWCEC* approach leads to a WCRE of 398.53 mJ, which is 79.25 % lower than the value determined by the all-always-on approach, representing the difference between the area under the green curve and the red area in Figure 7.

4.3 Exploiting Context-Sensitive Knowledge

In our next experiments, we focus on systems with multiple tasks and compare the WCRE values provided by *SysWCEC* with the WCRE values determined with the all-always-on approach. To obtain representative energy-consumption values, we combine the target platform, in this case the PATMOS research processor [58, 59], with our energy model. We configure the processor to run at 1 MHz and a static power consumption (i.e., all devices deactivated) of 10mA. As workload, we select a total of four benchmarks with different characteristics to be able to evaluate a wide spectrum of scenarios. Table 2 presents and compares the WCRE values determined by both methods evaluated. To compute the WCRE estimate for the all-always-on approach, we multiply the exact worst-case response time of the system by the amount of power the system consumes when all devices are switched on. In the following, we discuss the results of each benchmark in detail.

Benchmark #1 – Transceiver Benchmark

The structure of the first benchmark resembles the example in Figure 2: A low-priority task activates a device, in this case a WiFi transceiver, for example, to log the system status. During the execution of the task, an interrupt preempts the task and activates another higher-priority task, which is then dispatched after the interrupt service routine has terminated. For the worst-case scenario of the interrupt occurring while the transceiver is active, the low-priority task spends about 8 % of its overall response time before activating the device (i.e., Part Ⓐ in Figure 2), about 83 % of the time while the device is active (Part Ⓑ), and the remaining about 9 % of the time after having switched off the device (Part Ⓒ). Our results in Table 2 show that for this benchmark, the WCRE value determined by *SysWCEC* is 13.73 % lower than the WCRE value obtained with the all-always-on approach. This improvement is possible because due to its context-sensitive analysis of the system *SysWCEC* knows that it is impossible for the transceiver to be active during Part Ⓐ and Part Ⓒ of the low-priority task, which in combination represent about 17 % of the task's worst-case response time.

Benchmark #2 – Transceiver Benchmark with Resource

For our second benchmark, we modify the first benchmark and introduce a shared resource between the low-priority task and the high-priority task. In OSEK, shared resources are typically used to coordinate different tasks and can only be acquired by at most one task at a time. In our benchmark, the low-priority task acquires the resource right before activating the transceiver and releases it immediately after deactivating the transceiver. Applying the stack-based priority ceiling protocol [6], which is mandated by the OSEK standard to solve the problem of unbounded priority inversion, when the interrupt occurs while the low-priority task holds the resource, the execution of the high-priority task is deferred until the resource has been released. As a consequence, the high-priority task no longer has an influence on the transceiver's active time, independent of when the interrupt is actually triggered. In contrast to the all-always-on approach, *SysWCEC* is able to exploit this knowledge and consequently determines a 81.70 % lower WCRE value, accounting for the fact that even in the worst case the transceiver is only active during a small part of the low-priority task's response time.

Benchmark #3 – Synchronous Task Activation

In the next benchmark, a low-priority task synchronously activates two tasks with higher priorities, one prior to switching on a transceiver and the other one afterwards. As both high-priority tasks take the same time to run and due to their execution times dominating the response time of the low-priority task, the transceiver is activated about half way into the experiment. This leads to *SysWCEC* being able to provide a WCRE estimate that is 45.45 % lower than the all-always-on value.

Benchmark #4 – Asynchronous Events

The fourth benchmark consists of a task activating a transceiver and an interrupt service routine deactivating it again. Such a setting represents a textbook example of why context-sensitive WCRE analysis is conceptually different from worst-case response time analysis: To determine the worst-case response time, an analysis must consider the scenario in which the interrupt occurs as often as its minimum inter-arrival time allows; in this case, this results in a response time of 1,377 cycles. In contrast, WCRE analysis must focus on the scenario with the highest energy consumption, which for this benchmark is the interrupt being triggered

Table 3 Analysis runtimes and statistics for different benchmarks.

Benchmark	State Enumeration	States	Transitions	ILP Solving
Transceiver (w/o resource)	54.05 ms	63	71	534 ms
Transceiver (w/ resource)	69.23 ms	84	96	869 ms
Synchronous activation	40.44 ms	19	19	308 ms
Asynchronous IRQ	42.88 ms	35	40	136 ms
Multiple devices (2 tasks)	73.63 ms	119	135	0.91 s
Multiple devices (3 tasks)	535.22 ms	1,356	1,580	10.41 s
Multiple devices (4 tasks)	1.4 s	6,231	7,359	54.17 s
Multiple devices (5 tasks)	2.62 s	39,711	47,215	33.17 min

only once: at the very end of the task's computation; for this scenario, the response time is only 1,155 cycles. *SysWCEC* correctly identifies this scenario and determines a 16.12 % lower WCRE value than the all-always-on approach.

4.4 Scalability

In our last experiment, we evaluate the performance and scalability of the *SysWCEC* approach, thereby focusing on the two steps that contribute to the overall analysis runtime: the symbolic state enumeration performed by `dOSEK` and the solving of the ILP (see Section 3.3). All analyses run on a server (Intel Xeon E5, 80 cores, 132 GB RAM) and use Gurobi 7.5 for ILP solving. To reduce durations for ILP solving, we explored parameter-tuning strategies [28] and carried out optimizations in Gurobi, which eventually determines optimal bounds.

Apart from the results of the four benchmarks introduced in Section 4.3, we also present measurements gained from four additional benchmarks that use multiple devices and all share the following general structure: All of these benchmarks consist of a low-priority task whose WCRE is to be determined. Apart from this task, the benchmarks comprise a set of additional tasks that each possess a unique higher priority and are activated through dedicated asynchronous interrupt service routines; the minimum inter-arrival time between interrupts is 100 ms. All tasks in the system are assigned different devices. During execution, a task first switches on its device, then performs a computation, deactivates the device again, and finally terminates. To evaluate the impact of system complexity on analysis runtime, we rely on four different benchmarks whose task-set sizes range between 2 and 5. Note that due to the interfering interrupts and number of tasks, from a system-level perspective our multi-device benchmark comprising 5 tasks plus 4 task-activating interrupts has a comparable complexity as the real-world real-time benchmark DEBIE [31].

Table 3 compares the execution times for the two evaluated analysis steps and also presents the number of system states and transitions identified by *SysWCEC* for each benchmark. The results show that in general solving the integer linear program takes significantly more time than enumerating all system states and that the runtime of both analysis steps increases with the number of possible system states. In Section 6.2, we discuss how to further improve the performance and scalability of *SysWCEC* and its exponential state growth with increasingly complex systems. Nevertheless, even for systems with high complexity such as the multi-device benchmark with 5 tasks, preemptions through interrupts, and dependencies between tasks and interrupts, leading to 39,711 different system states and 47,215 transitions, *SysWCEC* can complete the entire analysis in around half an hour.

5 Related Work

The development of *SysWCEC* benefited from lessons learned in our previous work on system-wide timing analysis for fixed-priority real-time systems [19]. However, due to the substantial differences in objectives and requirements between timing analysis and energy-consumption analysis, we were not able to directly apply our existing analysis techniques to the problem of determining WCREs. Instead, it became necessary to develop a new approach, *SysWCEC*, that solves the specific problems associated with energy-consumption analysis. To the best of our knowledge, *SysWCEC* is the first work that enables determining WCEC bounds in peripheral-driven real-time systems that execute on various low-power modes.

Existing WCEC-analysis techniques [27, 35, 71] only consider the analysis of a single thread with a fixed set of active peripherals. In analogy to WCET tools, these analyzers follow the common approach of carrying out a hardware-independent path analysis that is combined with a hardware-dependent cost analysis. In contrast to these existing techniques, *SysWCEC* is able to analyze a whole embedded system with all attached or integrated power consumers. In addition to the awareness of all power-consuming devices and peripherals, the *SysWCEC* approach precisely addresses all non-hierarchical program flows such as synchronous task activations and asynchronous interrupts.

The integration of transceivers is common in the area of wireless sensor networks to estimate the overall lifetime of nodes [20, 39, 43]. Such lifetime-estimation tools are already featured by integrated development environments for battery-constrained devices [64]. Their basic principle is to multiply the maximum drawn power by the fixed duty cycle of periodically receiving/transmitting applications. In distinction to these approaches, *SysWCEC* is capable of modeling fixed-priority sporadic task sets with real-time constraints. An interleaved timing analysis determines the duration of activated devices and these costs are integrated into an overarching ILP formulation, which yields WCRE bounds.

Schneider pointed out that it is impossible to analyze timing constraints of applications without considering the semantics of the operating system and vice versa [56]. To solve this problem, he proposed to integrate fixed-priority scheduling semantics into timing analysis [57]. Following the idea of whole-system analysis, *SysWCEC* provides means to integrate multiple devices to determine WCRE bounds between two arbitrary program points in the system. The integration of operating-system standards [1, 49] into commercial, static analysis tools, such as Astrée, indicates the relevance of system semantics in analyzers [44].

In the context of real-time scheduling, a system's power consumption is often determined based on the frequency-aware power model [14, 33, 77, 78], which is then exploited by DVFS. This power model assumes that the system's dynamic power consumption only depends on the processor's frequency. However, when considering systems that use devices, such as sensors, this model is no longer applicable. *SysWCEC* addresses the integration of devices with knowledge of global system paths and their set of active devices.

Furthermore, many scheduling approaches using DVFS to minimize energy consumption while still guaranteeing timeliness neglect the overheads to switch the processor frequency [8]. As we found out for an ARM Cortex-M4 (EFM32), the latency to switch the frequency can be up to 396 µs. Rusu et al. and Zhang et al. discovered even greater latencies for the PowerPC 405LP [54] and a digital signal processor [76]. Since *SysWCEC*'s central data structure, the PSTG, contains all executed instructions and all possible paths in the whole system, it is inherently aware of these transition overheads.

In addition to the DVFS power model, a huge body of related work exists on energy-cost models for the processor's microarchitecture [13, 37, 41, 45, 50, 60, 61, 69]. However, considering the relations of power consumers in energy-constrained systems, processing cores only take a minor portion of the whole-system consumption. *SysWCEC* focuses on precisely integrating these consumers and determines WCRE bounds, but can profit from these advances on more fine-grained energy-consumption models.

6 Discussion & Future Work

In this section, we first discuss both improvements of our energy-consumption model and the scalability of the approach, and then outline our future work in the context of *SysWCEC*.

6.1 Improving Energy-Consumption Hardware Model

SysWCEC is a sound formulation of the path-analysis problem of whole-system WCRE analyses. In the current implementation, we use the maximum power consumption of a PSTG node and multiply it by its WCET to obtain an upper bound for the node's energy consumption. Our measurements in Section 4.2 show only small variations in power consumption within a power mode (i.e., around 5 %). This is especially true when these minor variations are put into relation with the orders of magnitude the whole system's power consumption varies when switching between power states. Consequently, we expect our method for determining a node's WCEC to result in only small overestimations. However, if necessary this model could be further improved with knowledge of the platform's microarchitecture [13, 37, 41, 45, 50, 60, 61, 69].

6.2 Further Improving Scalability

Although already providing reasonable analysis durations of around half an hour for larger systems (see Section 4.4), the scalability of the approach can be further enhanced. In particular, there are three directions for optimizing *SysWCEC* and mitigating the problem of exponential state growth: (1) constructing a smaller PSTG that also reduces the ILP size, (2) speeding up PSTG construction itself, and (3) improving ILP-solving times. First, it is possible to group several PSTG nodes together into a larger super structure [18]. Although this approach might sacrifice precision and lead to higher overestimations, it potentially enables smaller ILP formulations and thus shorter solving times. Second, `dOSEK`'s current state enumeration is implemented using a single thread. Consequently, the framework would greatly benefit from parallel symbolic state enumeration [11] and the usage of multi-core platforms. Third, although we applied heuristics to speed up ILP solving [28], the process can be further enhanced by providing upper bounds for variables by exploiting traditional WCRT analyses [5]. Thereby, search spaces can be narrowed and thus solving times reduced.

6.3 Trading Timeliness for Energy Consumption

Trading worst-case energy consumption for worst-case execution time and vice versa is an upcoming research area [26, 34]. In the evaluation, Benchmark #2 demonstrated that blocking the execution of a higher-priority task can be beneficial if it prevents the preemption of a lower-priority task that has previously activated a device. This blocking leads to a prolonged WCRT of the higher-priority task, but – depending on the deadlines, duration of the critical section, and power consumptions – the overall benefit can be optimized. With *SysWCEC*, we provide a framework that is capable of analyzing both WCRE and WCRT bounds along critical program paths of entire systems. This comprehensive framework is beneficial to find optimal solutions for peripheral-driven real-time systems in the time-vs-energy trade-off.

7 Conclusion

Power consumption in energy-constrained real-time systems depends not only on the processing unit but also on peripherals. Although these devices are often dynamically (de-)activated, current WCEC analyses do not take this fine-grained structure into account and therefore yield overestimations. Furthermore, with the (de-)activation of devices, the energy consumption of a single task can influence any other task (also tasks of higher priority).

The *SysWCEC* analyzer processes OSEK-compliant (i.e., ECC1) real-time systems and determines WCRE bounds. For this, we present the power-state–transition graph, a device and operating-system–aware data structure. Using this representation, we are able to enumerate all possible system states of fixed-priority real-time systems using multiple devices. This knowledge allows formulating an ILP, whose solution eventually yields the WCRE.

Source code of *SysWCEC*: `https://gitlab.cs.fau.de/syswcec`

References

1 AEEC. Avionics application software standard interface (ARINC specification 653-1), 2003.

2 aiT worst-case execution time analyzers. `http://www.absint.com/ait/`.

3 ARM Limited. Cortex-M4 technical reference manual, 2010.

4 ARM Limited. Cortex-M0+ technical reference manual, 2012.

5 N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

6 T. P. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. of the 11th Real-Time Systems Symp. (RTSS '90)*, pages 191–200, 1990.

7 T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

8 M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 15(1):7, 2016.

9 L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000.

10 A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. The auspicious couple: Symbolic execution and WCET analysis. In *Proc. of the 13th Int'l Work. on Worst-Case Execution Time Analysis (WCET '13)*, pages 53–63, 2013.

11 S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proc. of the ACM SIGOPS European Conf. on Computer Systems (EuroSys '11)*, pages 183–197, 2011.

12 A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. of the USENIX Annual Technical Conf. (ATC '10)*, pages 1–14, 2010.

13 N. Chang, K. Kim, and H. G. Lee. Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI. In *Proc. of the 2000 Int'l Symp. on Low Power Electronics and Design (ISLPED '00)*, pages 185–190, 2000.

14 J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Proc. of the 13th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pages 28–38, 2007.

15 R. I. Davis. On the evaluation of schedulability tests for real-time scheduling algorithms. In *Proc. of the Work. on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS '16)*, 2016.

16 H.-P. Deifel, C. Dietrich, M. Göttlinger, D. Lohmann, S. Milius, and L. Schröder. Automatic verification of application-tailored OSEK kernels. In *Proc. of the 17th Conf. on Formal Methods in Computer-Aided Design (FMCAD '17)*, pages 1–8, 2017.

17 C. Dietrich, M. Hoffmann, and D. Lohmann. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, pages 6:1–6:10, 2015.

18 C. Dietrich, M. Hoffmann, and D. Lohmann. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 16:35:1–35:25, 2017.

19 C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. SysWCET: Whole-system response-time analysis for fixed-priority real-time systems. In *Proc. of the 23nd Real-Time and Embedded Technology and Applications Symp. (RTAS '17)*, pages 37–48, 2017.

20 W. Dron, S. Duquennoy, T. Voigt, K. Hachicha, and P. Garda. An emulation-based method for lifetime estimation of wireless sensor networks. In *Proc. of the Int'l Conf. on Distributed Computing in Sensor Systems (DCOSS '14)*, pages 241–248, 2014.

21 EEMBC ULPMark FAQs. `http://www.eembc.org/ulpbench/faq.php`.

22 F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat. From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors. In *Proc. of the 19th Int'l Symp. on Real-Time Distributed Computing (ISORC '16)*, pages 134–141, 2016.

23 Freescale Semiconductor, Inc. *KL46 Sub-Family Reference Manual*, 2013.

24 Freescale Semiconductor, Inc. *Kinetis KL46 Sub-Family*, 2014.

25 Fujitsu Semiconductor. *FRAM MB85RC256V*, 2013.

26 R. Gran, J. Segarra, C. Rodríguez, L. C. Aparicio, and V. Viñals. Optimizing a combined WCET-WCEC problem in instruction fetching for real-time systems. *Journal of Systems Architecture*, 59(9):667–678, 2013.

27 N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static analysis of energy consumption for LLVM IR programs. In *Proc. of the 18th Int'l Work. on Software and Compilers for Embedded Systems (SCOPES 2015)*, pages 12–21. ACM, 2015.

28 Gurobi Optimization, Inc. Gurobi optimizer reference manual. `www.gurobi.com`, 2017.

29 S. Hepp, B. Huber, D. Prokesch, and P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proc. of the 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS '15)*, pages 277–292, 2015.

30 M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proc. of the 21st Real-Time and Embedded Technology and Applications Symp. (RTAS '15)*, pages 259–270, 2015.

31 N. Holsti, T. Langbacka, and S. Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. In *Proc. of the Data Systems in Aerospace Conf. (DASIA '00)*, pages 1–6, 2000.

32 O. Hruška. Esp8266 killing itself? `ondrovo.com/a/20170205-esp-self-destruct/`, 2017.

33 P. Huang, Pratyush Kumar, G. Giannopoulou, and L. Thiele. Energy efficient DVFS scheduling for mixed-criticality systems. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT '14)*, pages 1–10, 2014.

34 C. Imes, DHK Kim, M. Maggio, and H. Hoffmann. POET: A portable approach to minimizing energy under soft real-time constraints. In *Proc. of the 21th Real-Time and Embedded Technology and Applications Symp. (RTAS '15)*, pages 75–86, 2015.

35 R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *Proc. of the 12th Real-Time and Embedded Technology and Applications Symp. (RTAS '06)*, pages 81–90, 2006.

36 Keithley Instruments Inc. *Models 2611B, 2612B and 2614B, System SourceMeter*, 2013.

37 S. Kerrison and K. Eder. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 14(3):56, 2015.

38 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proc. of the 21st Int'l Conf. on Real-Time Networks and Systems (RTNS '13)*, pages 161–170, 2013.

39 A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. Haneveld, T. Parker, O. Visser, H. Lichte, and S. Valentin. Simulating wireless and mobile networks in OMNeT++ the MiXiM vision. In *Proc. of the 1st Int'l Conf. on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (SimuTools '08)*, pages 1–10, 2008.

40 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization (CGO '04)*, pages 75–86, 2004.

41 S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded RISC processors. *SIGPLAN Notices*, 36(8):1–10, 2001.

42 Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the 32nd Annual Design Automation Conf. (DAC '95)*, pages 456–461, 1995.

43 Y. Liu, W. Zhang, and K. Akkaya. Static worst-case energy and lifetime estimation of wireless sensor networks. In *Proc. of the 28th Int'l Performance Computing and Communications Conf. (IPCCC '09)*, pages 17–24, 2009.

44 A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking static analysis to the next level: Proving the absence of run-time errors and data races with Astrée. In *Proc. of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, pages 570–579, 2016.

45 S. Nikolaidis, N. Kavvadias, P. Neofotistos, K. Kosmatopoulos, T. Laopoulos, and L. Bisdounis. Instrumentation set-up for instruction level power modeling. In *Integrated Circuit Design*, pages 71–80, 2002.

46 NXP Semiconductors. *Manufacturer BOM Report*, 2013.

47 NXP Semiconductors. *Xtrinsic MAG3110 Three-Axis, Digital Magnetometer*, 2013.

48 NXP Semiconductors. *MMA8451Q, 3-axis, 14-bit/8-bit digital accelerometer*, 2017.

49 OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005.

50 J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modelling: A worst case perspective. *Computing Research Repository, arXiv*, 2015.

51 P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proc. of the 9th Work. on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS '13)*, pages 33–40, 2013.

52 P. Puschner and A. Schedl. Computing maximum task execution times: A graph-based approach. *Real-Time Systems*, 13:67–91, 1997.

53 RapiTime - worst-case execution time (WCET) analysis for critical systems. `http://www.rapitasystems.com/products/RapiTime`.

54 C. Rusu, R. Xu, R. Melhem, and D. Mossé. Energy-efficient policies for request-driven soft real-time systems. In *Proc. of the 16th Euromicro Conf. on Real-Time Systems (ECRTS '04)*, pages 175–183, 2004.

55 F. Scheler and W. Schröder-Preikschat. The real-time systems compiler: Migrating event-triggered systems to time-triggered systems. *Software: Practice and Experience*, 41(12):1491–1515, 2011.

56 J. Schneider. Why you can't analyze RTOSs without considering applications and vice versa. In *Proc. of the 2nd Int'l Work. on Worst-Case Execution Time Analysis (WCET '02)*, pages 79–84, 2002.

57 J. Schneider. *Combined schedulability and WCET analysis for real-time operating systems.* Shaker, 2003.

58 M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

59 M. Schoeberl et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61:449–471, 2015.

60 Y. S. Shao and D. Brooks. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *Proc. of the Int'l Symp. on Low Power Electronics and Design (ISLPED '13)*, pages 389–394, 2013.

61 V. Sieh, R. Burlacu, T. Hönig, H. Janker, P. Raffeck, P. Wägemann, and W. Schröder-Preikschat. An end-to-end toolchain: From automated cost modeling to static WCET and WCEC analysis. In *Proc. of the 20th Int'l Symp. on Real-Time Distributed Computing (ISORC '17)*, pages 1–10, 2017.

62 Silicon Laboratories, Inc. *Energy Debugging Tools for Embedded Applications.*

63 Silicon Laboratories, Inc. *User Manual Starter Kit EFM32GG-STK3700*, 2013.

64 Silicon Laboratories, Inc. *AN0822: Simplicity Studio™User's Guide*, 2016.

65 Silicon Laboratories, Inc. *EFM32GG Reference Manual*, 2016.

66 Symtavision tools: SymTA/S. `http://www.symtavision.com/symtas.html`.

67 Espressif Systems. *ESP8266 Technical Reference*, 2017.

68 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. of the Int'l Symp. on Circuits and Systems (ISCAS '00)*, volume 4, pages 101–104, 2000.

69 V. Tiwari and M. T.-C. Lee. Power analysis of a 32-bit embedded microcontroller. *VLSI Design*, 7(3):225–242, 1998.

70 M. Völp, M. Hähnel, and A. Lackorzynski. Has energy surpassed timeliness? – scheduling energy-constrained mixed-criticality systems. In *Proc. of the 20th Real-Time and Embedded Technology and Applications Symp. (RTAS '14)*, pages 275–284, 2014.

71 P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Proc. of the 27th Euromicro Conf. on Real-Time Systems (ECRTS '15)*, pages 105–114, 2015.

72 P. Wägemann, T. Distler, H. Janker, P. Raffeck, and V. Sieh. A kernel for energy-neutral real-time systems with mixed criticalities. In *Proc. of the 22nd Real-Time and Embedded Technology and Applications Symp. (RTAS '16)*, pages 25–36, 2016.

73 P. Wägemann, T. Distler, H. Janker, P. Raffeck, V. Sieh, and W. Schröder-Preikschat. Operating energy-neutral real-time systems. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, pages 11:1–11:25, 2017.

74 R. Wilhelm et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 7(3):1–53, 2008.

75 C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proc. of the USENIX Annual Technical Conf. (ATC '12)*, pages 1–14, 2012.

76 G. Zeng, T. Yokoyama, H. Tomiyama, and H. Takada. Practical energy-aware scheduling for real-time multiprocessor systems. In *Proc. of the 15th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA '09)*, pages 383–392, 2009.

77 D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of the Int'l Conf. on Computer Aided Design (ICCAD '04)*, pages 35–40, 2004.

78 Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *Proc. of the 10th Real-Time and Embedded Technology and Applications Symp. (RTAS '04)*, pages 84–93, 2004.

Using Lock Servers to Scale Real-Time Locking Protocols: Chasing Ever-Increasing Core Counts

Catherine E. Nemitz
The University of North Carolina at Chapel Hill, USA
nemitz@cs.unc.edu

Tanya Amert
The University of North Carolina at Chapel Hill, USA

James H. Anderson
The University of North Carolina at Chapel Hill, USA

Abstract

During the past decade, parallelism-related issues have been at the forefront of real-time systems research due to the advent of multicore technologies. In the coming years, such issues will loom ever larger due to increasing core counts. Having more cores means a greater potential exists for platform capacity loss when the available parallelism cannot be fully exploited. In this paper, such capacity loss is considered in the context of real-time locking protocols. In this context, lock nesting becomes a key concern as it can result in transitive blocking chains that force tasks to execute sequentially unnecessarily. Such chains can be quite long on a larger machine. Contention-sensitive real-time locking protocols have been proposed as a means of "breaking" transitive blocking chains, but such protocols tend to have high overhead due to more complicated lock/unlock logic. To ease such overhead, the usage of lock servers is considered herein. In particular, four specific lock-server paradigms are proposed and many nuances concerning their deployment are explored. Experiments are presented that show that, by executing cache hot, lock servers can enable reductions in lock/unlock overhead of up to 86%. Such reductions make contention-sensitive protocols a viable approach in practice.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Software and its engineering → Mutual exclusion, Software and its engineering → Real-time systems software, Software and its engineering → Synchronization, Software and its engineering → Process synchronization

Keywords and phrases multiprocess locking protocols, nested locks, priority-inversion blocking, reader/writer locks, real-time locking protocols

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.25

Supplement Material ECRTS Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.4.2.2`

Funding Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

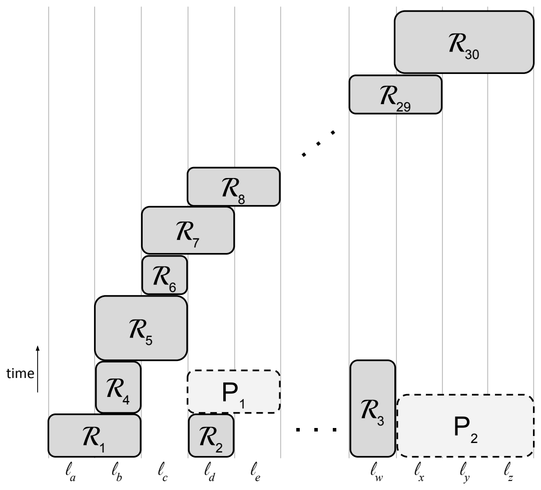


Figure 1 An illustration of transitive blocking.

1 Introduction

The evolution of multicore technologies over the past decade has shifted the focus of real-time systems research by making parallelism a paramount concern. During this time, the extent of parallelism available in commercially produced machines has steadily increased. Ten years ago, a quad-core machine was considered large. Today, machines with core counts of dozens or more are available. Looking forward, ever increasing core counts are likely to continue. The implication for real-time systems research is that resource-allocation methods shown to work well in the past may not scale as hardware platforms continue to evolve.

In this paper, we consider the issue of scale as it pertains to *real-time locking protocols*. For such a protocol to scale to large core counts, it must address intricate challenges posed by *nested lock requests*, which occur in a variety of applications [7, 13]. In particular, such requests can cause *transitive blocking chains* that cause tasks to unnecessarily execute sequentially. The potential for lost parallelism due to sequential execution increases with higher core counts. For example, assuming non-preemptive locks, if such a chain were to involve all cores of a quad-core machine, then 75% of the available processing capacity would be lost until the task at the head of the chain releases its acquired resources. On a much larger machine with, say, 32 cores, this percentage of loss would swell to nearly 97% if all cores were involved. Even if nested requests occur much less often than non-nested ones, they can still result in long blocking chains, particularly in the worst case, which would typically be considered in real-time schedulability analysis. We illustrate this point with an example chain of blocking that could occur, and thus must be accounted for in analysis.

▶ **Example 1.** Consider a set of 30 requests, some of which are shown in Fig. 1. Request $\mathcal{R}_1$ and requests $\mathcal{R}_4$ through $\mathcal{R}_{30}$ form a transitive blocking chain for the resources shown on the horizontal axis. The vertical axis shows time, with different box heights representing different critical-section lengths, and the placement along this axis representing when each request will be satisfied. Most of the blocking shown in Fig. 1 is avoidable. For example, $\mathcal{R}_8$ could move to position P_1, and $\mathcal{R}_{30}$ into P_2, greatly reducing their blocking. Note that moving $\mathcal{R}_8$ earlier reduces the blocking of later issued requests. Note also that even non-nested requests (*e.g.*, $\mathcal{R}_6$) can be transitively blocked.

Contention-sensitive real-time locking protocols guarantee the blocking time of each request is only proportional to the number of directly conflicting earlier requests by effectively "breaking" transitive blocking chains [43, 55]. Referring back to Ex. 1, enqueuing $\mathcal{R}_8$ as depicted is not contention sensitive as this queue ordering forces $\mathcal{R}_8$ to block on $\mathcal{R}_1$, $\mathcal{R}_4$, $\mathcal{R}_5$, and $\mathcal{R}_6$, none of which directly conflicts with $\mathcal{R}_8$ (they access different resources). In contrast, enqueuing $\mathcal{R}_8$ in position P_1 would ensure contention-sensitive blocking for it.

Unfortunately, the complex lock/unlock logic required to enable contention-sensitive enqueuings can result in higher overhead. To mitigate this issue, we explore herein the usage of lock servers to lessen such overhead. A *lock server* is a special process that sequentially performs all lock and unlock functions of a given protocol. The main advantage of using lock servers is that they can run cache hot (which is explained in the context of our platform in Sec. 3). The main disadvantage is the need to dedicate whole cores, or fractions of cores, to performing synchronization functions. However, on machines with high core counts, this may be a reasonable thing to do, as has been observed by others in other contexts [41, 50]. The main focus of this paper is to experimentally document the extent of overhead reduction lock servers enable when supporting contention-sensitive locking protocols. We show that such reductions can be *substantial*. We also examine various tradeoffs that arise with respect to how lock servers are deployed. We elaborate on these tradeoffs and our contributions below, after first presenting an overview of prior work to provide context.

Related work. The literature on real-time multiprocessor locking protocols is quite large [1, 2, 3, 4, 6, 8, 9, 12, 11, 10, 14, 15, 16, 18, 17, 19, 20, 21, 23, 24, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36, 37, 42, 43, 48, 49, 51, 53, 54, 55, 57, 58, 59, 60, 61, 62, 64, 65, 63, 67, 68, 71, 73]. Of the just-cited papers, we comment on several that are particularly relevant to our work.

A number of server-based locking protocols have been proposed previously that employ notions similar to a lock server but for a different purpose, namely, to ease the calculation of bounds on priority-inversion blocking (pi-blocking).[1] The first such protocol was the *distributed priority ceiling protocol (DPCP)* [57, 58, 59], which statically binds resources to cores and requires tasks to perform lock and unlock calls for a resource on the core assigned to that resource. Subsequently, a number of server-based protocols were proposed that follow a similar approach [21, 32, 33, 41, 42, 49, 73]. In contrast to these various server-based protocols, our focus in this paper is to preserve the blocking bounds of a given protocol while reducing its overhead. Also, our main concern is dealing with nested lock requests, which are actually precluded in most prior server-based protocols.

Only a few real-time multiprocessor locking protocols have been proposed that support nested lock requests. Among such protocols, only those in the *real-time nested locking protocol (RNLP)* family provide asymptotically optimal pi-blocking bounds [43, 55, 62, 64, 65, 63]. The RNLP family also includes the only proposed real-time locking protocols shown to be contention sensitive. We review these protocols in more detail later. Outside of the RNLP family, two other protocols have been proposed that directly support lock nesting, the *multiprocessor bandwidth inheritance protocol* [32, 33] and *MrsP* [21, 36, 73]. Neither is optimal, but both use creative techniques, like migration, to lessen blocking times.

Our work was partially inspired by work on a concept called *remote core locking* (*RCL*), which was directed at improving the performance of legacy *non-real-time* code when moving it from a uniprocessor system to a multiprocessor one [50]. In particular, RCL seeks to avoid

[1] Pi-blocking, which is more carefully considered in Sec. 2, is the primary basis on which different locking protocols are compared.

cache-line bouncing when a resource is accessed on different cores by requiring all resource accesses to occur on a designated core. In these sense, RCL is similar to the DPCP and related protocols, but the emphasis in work on RCL is to enable critical sections to run cache hot. In contrast, we want lock and unlock routines to run cache hot.

Contributions. We present an in-depth study of lock servers as a means for providing efficient implementations of contention-sensitive real-time locking protocols on large multicore machines. We restrict attention to protocols that use spinning (*i.e.*, busy waiting) to realize task blocking. We take our particular test platform as an exemplar of a "large" machine; this platform provides 36 cores split evenly across two sockets. We define lock servers in a way that does not fundamentally alter the blocking analysis of the locking protocol being supported. Thus, such analysis is not our major focus: *overhead* is.

We introduce lock servers by initially assuming a particular contention-sensitive locking protocol is to be supported that was designed assuming that all critical sections are uniformly of the same length. Using this protocol, we present four lock-server paradigms that are defined by specifying servers as either static or floating and either global or local. A *static* lock server is bound to a single core, while a *floating* one may migrate. A *global* lock server handles requests from all cores, while a *local* one handles requests from only its socket. Our test platform has two sockets, so in that context, the local case requires consideration of two lock servers, which require further arbitration. We do this by letting these lock servers alternate in phases, where the phase switching is controlled by a novel synchronization mechanism introduced here for the first time called a *phase-fair reader/reader lock*. After examining these various alternatives, we consider the ramifications of relaxing the uniformity requirement and allowing critical sections to be of different lengths.

To assess the efficacy of using lock servers, we conducted an extensive experimental evaluation on our test platform of all of the lock-server configurations mentioned above. In these experiments, the use of lock servers often reduced overhead dramatically. When supporting non-uniform critical sections, one of our lock-server paradigms reduced overhead by up to 72%. When supporting uniform critical sections, this decrease was as high as 86%.

Organization. In the rest of this paper, we provide needed background (Sec. 2), introduce static (Sec. 3) and floating (Sec. 4) lock servers assuming critical-section lengths are uniform, eliminate this uniformity assumption (Sec. 5), present our experimental evaluation (Sec. 6), and conclude (Sec. 7).

2 Background

In this section, we present relevant background material on task and resource models and provide further details concerning the locking protocols most relevant to our work.

Task Model. We consider a sporadic task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$. (We assume familiarity with the sporadic model.) These n tasks are scheduled on m processors by a job-level fixed-priority scheduler, such as one using earliest-deadline-first (EDF) priorities.

Resource Model. We focus on spin-based locking protocols invoked non-preemptively. We assume a set of n_r shared resources denoted $\mathcal{L} = \{\ell_1, \ldots, \ell_{n_r}\}$. When a job J requires access to one or more of these resources, it *issues* a request. We index requests in the order they are issued. An arbitrary request of J is denoted $\mathcal{R}_i$, and the set of resources it requires is

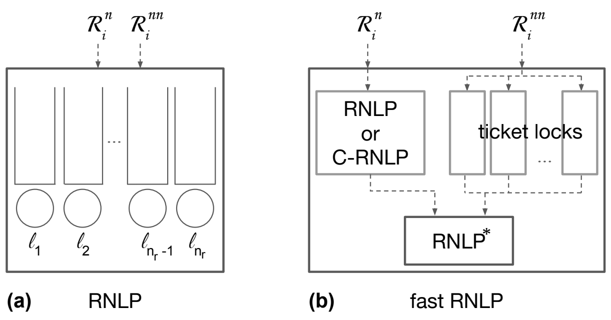


Figure 2 Important RNLP variants.

denoted D_i. $\mathcal{R}_i$ is said to be *satisfied* when J *holds* all resources in D_i. J then executes its *critical section* for L_i time units. When J *releases* all of the resources it held, $\mathcal{R}_i$ *completes*. $\mathcal{R}_i$ is considered to be *active* from the time it is issued until the time it completes.

Real-time locking protocols must have proven bounds on *priority-inversion blocking (pi-blocking)*. Pi-blocking occurs when a job cannot execute because of lower-priority work. In the context of non-preemptive spin-based protocols, a job is pi-blocked if it is spinning or if it cannot execute because some lower-priority job is executing non-preemptively.

Allowing requests to be issued for multiple resources at once as specified above provides a mechanism called a *dynamic group lock (DGL)* [63]. With DGLs, lock nesting is supported by requiring a job to issue one request for all of its needed resources, instead of issuing a separate request for each resource. The dynamic nature of DGLs allows groups of requested resources to be determined as required at runtime. This is in contrast to static group locks [9], which require resource groups to be determined offline and remain fixed.

We use DGLs to prevent deadlock. Another common approach is to define a resource ordering and require that resources be requested in that order [29, 38]. If conditional code exists, DGLs require the acquisition of resources that may not actually be needed. However, the use of DGLs and the use of a resource ordering result in the same pi-blocking bounds [62].

In stating such bounds, we assume that the maximum critical-section length, L_{max}, is constant. Additionally, we refer to the *contention* c_i experienced by a request $\mathcal{R}_i$: c_i is defined to be the number of requests that are active while $\mathcal{R}_i$ is active and that require one or more of the same resources as $\mathcal{R}_i$. A non-preemptive spin-based locking protocol is *contention sensitive* if it ensures a pi-blocking bound of $O(\min(m, c_i))$ per request.

In comparing different locking protocols, we care about overhead in addition to pi-blocking bounds. If a request $\mathcal{R}_i$ is issued at time t, and t' is the earliest time it either starts spinning or is satisfied, then the *lock overhead* $\mathcal{R}_i$ experiences is $t' - t$. Similarly, the *unlock overhead* $\mathcal{R}_i$ experiences is the total time needed to release all of its acquired resources. When we use the term *overhead* without qualification, we mean total lock plus unlock overhead.

The RNLP. The *RNLP (real-time nested locking protocol)* was the first real-time locking protocol to support nested lock requests with asymptotically optimal worst-case pi-blocking bounds [63]. The RNLP is actually a suite of protocols: both spin- and suspension-based variants exist and deadlock avoidance can be achieved by using either resource orderings or DGLs. We focus here on the spin-based DGL variant. At a high level, this variant is quite simple. As shown in Fig. 2(a), per-resource FIFO spin queues are used, and when a request for a set of resources is issued by some task, that resource is atomically enqueued onto the queues of all requested resources. Note that this atomic enqueueing requires the usage of an

Protocol	Pi-blocking	Overhead
RNLP	O(m)	moderate
C-RNLP	O(min(m,c))	high
fast RNLP (with RNLP)	O(m) nested	moderate (nested)
	O(min(m,c)) non-nested	low (non-nested)
fast RNLP (with C-RNLP)	O(min(m,c)) nested	high (nested)
	O(min(m,c)) non-nested	low (non-nested)

Figure 3 Important RNLP variants.

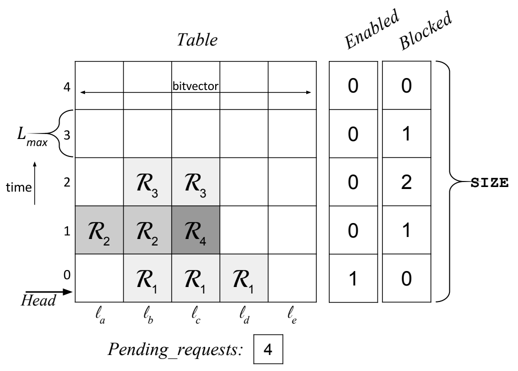


Figure 4 Variables used in the U-C-RNLP (a similar depiction appears in [43]).

underlying mutex lock, which results in moderate lock overhead. Also, the RNLP provides no mechanism for reducing transitive blocking. For example, all of the transitive blocking shown in Fig. 1 can occur if requests are atomically enqueued as in the RNLP.

Contention-sensitive variants. The *C-RNLP (contention-sensitive RNLP)* was proposed to eliminate the long transitive blocking chains that can occur under the RNLP [43]. It does this by using a *cutting ahead* mechanism that enables contention-sensitive pi-blocking at the expense of higher overhead. A fairly detailed overview of the C-RNLP is provided later in Sec. 3 in discussing lock servers, so we refrain from providing further details now.

The *fast RNLP*[2] was proposed to achieve contention sensitivity and low overhead for non-nested requests, which are likely the common case in practice [55]. Nested requests can be made either contention sensitive at the expense of relatively high overhead for them, or non-contention sensitive, which entails much lower overhead. This functionality is achieved by employing a modular structure, as shown in Fig. 2(b). Each non-nested request acquires a simple ticket lock associated with its resource, while each nested request competes within either the RNLP (if contention sensitivity is not provided for such requests) or the C-RNLP (if it is). The RNLP* is a low-overhead version of the RNLP that must merely arbitrate between at most one non-nested request and at most one nested request per resource.

The RNLP variants just overviewed are summarized in Table 3.

3 Static Lock Servers

In this section, we consider the use of static lock servers to implement the C-RNLP. The C-RNLP is described in [43] in an abstract rule-based way. These rules can be realized in different ways in an actual implementation. For ease of exposition, we limit attention for now to the *uniform* implementation of the C-RNLP given in [43], which was designed assuming that all critical sections are the same length. Later, in Sec. 5, we will relax this assumption. In order to understand how to implement the uniform C-RNLP using lock servers, a basic understanding of it is required.

[2] Actually, the fast RNLP was proposed as the fast RW-RNLP because it provides reader/writer sharing. For simplicity, we ignore read requests in this paper and focus only on mutex sharing.

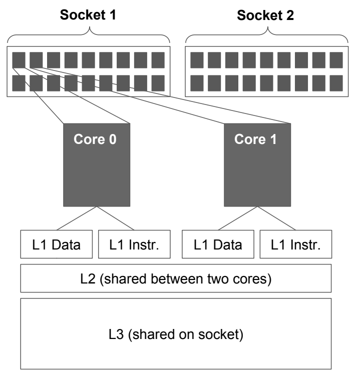


Figure 5 Test platform architecture.

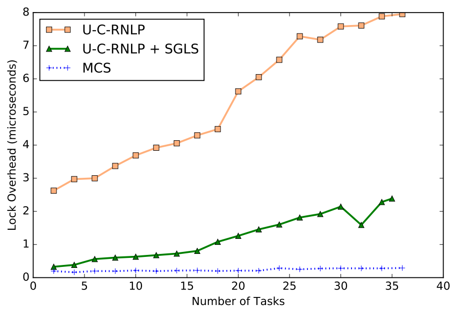


Figure 6 Lock overhead under the U-C-RNLP with and without a lock server.

Uniform C-RNLP. Under the uniform C-RNLP, denoted U-C-RNLP, each request $\mathcal{R}_i$ is satisfied within $\min(m, (c_i + 1))L_{max}$ time units, which meets the definition of contention sensitivity. This bound is realized by using a *Table* of possible satisfaction times. Each row of *Table* stores one or more bit vectors and represents a single start time, with each bit in that row representing one resource, as depicted in Fig. 4 with four requests. In the simplest implementation on a 64-bit machine, one bit vector is used, allowing 64 resources to be managed. The corresponding arrays *Enabled* and *Blocked* track which set of requests is satisfied and how many requests are immediately blocking a row in *Table*, respectively. For example, in Fig. 4, all requests in Row 0 of *Table*—here just request $\mathcal{R}_1$ for $D_1 = \{\ell_b, \ell_c, \ell_d\}$—are currently satisfied, as indicated by $Enabled[0] = 1$. The requests in the other rows are not currently satisfied. Requests in Row 1 are immediately waiting for one request, namely $\mathcal{R}_1$, to complete, as recorded by $Blocked[1] = 1$. Requests in Row 2 are waiting for two requests immediately preceding them to complete, as indicated by $Blocked[2] = 2$.

Platform description. In order to describe the lock-server paradigms considered in this paper more concretely, we specifically focus on our particular test platform, which is a dual-socket, 18-cores-per-socket Intel Xeon E5-2699. This platform provides significant per-socket parallelism while allowing issues on a multi-socket machine to be explored. As depicted in Fig. 5, each core has a 32KB L1 data cache and a 32KB L1 instruction cache. Pairs of cores share a unified 256KB L2 cache, and all cores on a socket share a unified 45MB L3 cache. We refer to lock state as *cache hot* if it maintains cache affinity in the lowest-level cache shared among all cores on which the server may execute.

The problem. Before delving into some of the nuances of using lock servers, let us examine the problem that they are intended to solve. Fig. 6 plots lock overhead as a function of core count (and thus number of requests) for three possibilities: the U-C-RNLP as originally presented in [43]; the same protocol but implemented using a single global lock server (denoted U-C-RNLP + SGLS); and an implementation in which all resources are coalesced under one lock using Mellor-Crummey and Scott's queue lock (denoted MCS) [52]. We take the latter as the gold standard for low overhead. We will carefully examine many such graphs in Sec. 6, so we will not bother to describe this particular one in any more detail now. However, notice the wide gap between the lock overhead for the U-C-RNLP compared to that for MCS. Our objective in this paper is to narrow this gap, hopefully considerably.

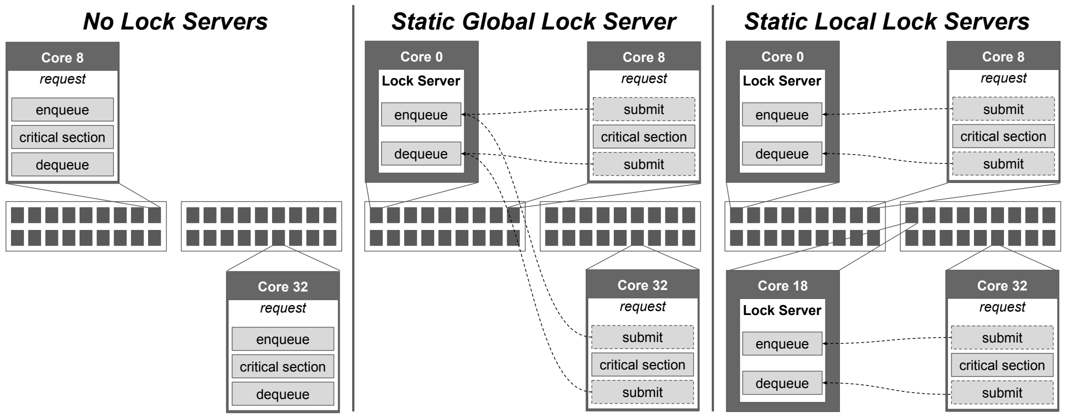


Figure 7 Three options: no lock servers (left), a single static global lock server (middle), and two per-socket static local lock servers (right).

Algorithm 1 Static Global Lock Server.

```
1:  procedure SGLS($core$: array of ptr to $core\_data$)
2:      var $k$: unsigned int
3:      while (TRUE):
4:          if $core[k]\rightarrow service$ = LOCK_SERVICE:
5:              $\hat{\ } core[k]\rightarrow spin\_location$ := LS-LOCK($core[k]\rightarrow requested$)
                                          ▷ Non-blocking LS-LOCK returns spin location
6:              $core[k]\rightarrow service$ := NULL
7:          else if $core[k]\rightarrow service$ = UNLOCK_SERVICE:
8:              LS-UNLOCK($core[k]\rightarrow requested$)
9:              $core[k]\rightarrow service$ := NULL
10:         $k := k + 1 \mod$ NR_CPUS
```

Lock servers. Recall that our focus in this section is static lock servers that are pinned to dedicated cores. We consider two variations of this idea: using a *global* lock server that services requests from all cores, and using (on our platform) two *local* lock servers, each servicing requests coming from one socket. Fig. 7 depicts these two possibilities in comparison to a conventional locking protocol implementation that does not use lock servers. The potential value of lock servers can be seen by comparing the curve for U-C-RNLP + SGLS to the U-C-RNLP curve in Fig. 6. (Again, we consider graphs like this in detail later.)

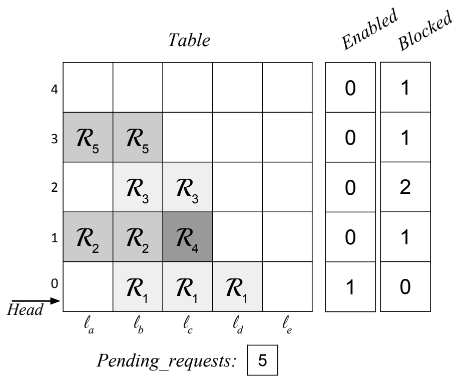
3.1 A Static Global Lock Server

The simplest way to employ a lock server is to dedicate a single core to servicing all lock requests. The server uses a special version of a given protocol's LOCK call, denoted LS-LOCK, that updates the lock state to add a given request and then, instead of waiting by spinning to be satisfied, returns the location of a variable on which to spin. Similarly, a special version of UNLOCK, denoted LS-UNLOCK, is used. Note that these routines require no underlying mutex, as no task other than the lock server will ever access the lock state.

The behavior of the lock server is as specified in Alg. 1. It is continually active (Line 3), looping through each core (Line 10). Because our focus is non-preemptive, spin-based protocols, we know each core will have at most one active request at a given time. For a specific core k, the server checks if there is an active request that needs lock service (Line 4). If so, it uses LS-LOCK to add the request to the lock state and determine the spin location for it (Line 5). In the case of the U-C-RNLP, this is the entry in *Enabled* that corresponds

Algorithm 2 New "Lock" and "Unlock" Submit Routines.

```
1: procedure SUBMIT-LOCK($c$: ptr to $core\_data$, $D$: set of resources)
2:     $c \rightarrow requested := D$
3:     $c \rightarrow service :=$ LOCK_SERVICE
4:     await $c \rightarrow service =$ NULL
5:     await $c \rightarrow spin\_location =$ TRUE
6: procedure SUBMIT-UNLOCK($c$: ptr to $core\_data$, $D$: set of resources)
7:     $c \rightarrow requested := D$
8:     $c \rightarrow service :=$ UNLOCK_SERVICE
9:     await $c \rightarrow service =$ NULL
```



	ℓ_a	ℓ_b	ℓ_c	ℓ_d	ℓ_e	Enabled	Blocked
4						0	1
3	$\mathcal{R}_5$	$\mathcal{R}_5$				0	1
2		$\mathcal{R}_3$	$\mathcal{R}_3$			0	2
1	$\mathcal{R}_2$	$\mathcal{R}_2$	$\mathcal{R}_4$			0	1
0		$\mathcal{R}_1$	$\mathcal{R}_1$	$\mathcal{R}_1$		1	0

Figure 8 $\mathcal{R}_5$ is added to Row 3 of *Table*.

to the row in *Table* to which the request was added. The server then indicates that this core no longer requires service (Line 6). If instead, a request on core k requires unlock service (Line 7), the server removes it from the lock state by calling LS-UNLOCK (Line 8). It then updates the service variable indicating that core k no longer requires service (Line 9).

In the next example, we now turn our focus to the behavior of a requesting task.

▶ **Example 2.** Fig. 8 shows the result of processing a request $\mathcal{R}_5$ for $D_5 = \{\ell_a, \ell_b\}$ that is issued after requests $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$ shown in Fig. 4. With a single global lock server, $\mathcal{R}_5$ executes SUBMIT-LOCK as shown in Alg. 2. It first sets *Requested* (Line 2) for its core and then indicates that it is awaiting lock service by the server (Line 3). After it has been serviced (Line 4), it spins on the location the server determined based on the other active requests (Line 5). As implied by Fig. 8, $\mathcal{R}_5$ spins on *Enabled*[3].

Using a global lock server in this manner has no impact on blocking; it simply changes the enqueuing and dequeuing portions of request processing in order to reduce overhead.

3.2 Static Local Lock Servers

In contrast to a global lock server, a local one is allowed to handle resource requests from only one socket. Our test platform has two sockets, so two lock servers are required to handle all requests; we denote them as $\mathcal{LS}_1$ and $\mathcal{LS}_2$. In this section, we assume that these lock servers are static, which means that each lock server is pinned to a specific core on its socket. The advantage of having two lock servers is that each must handle requests from only half the cores, and thus should execute with lower overhead. The disadvantage is that some arbitration mechanism is needed to mediate conflicting requests managed by the two servers. We illustrate the nature of the needed mediation with an example.

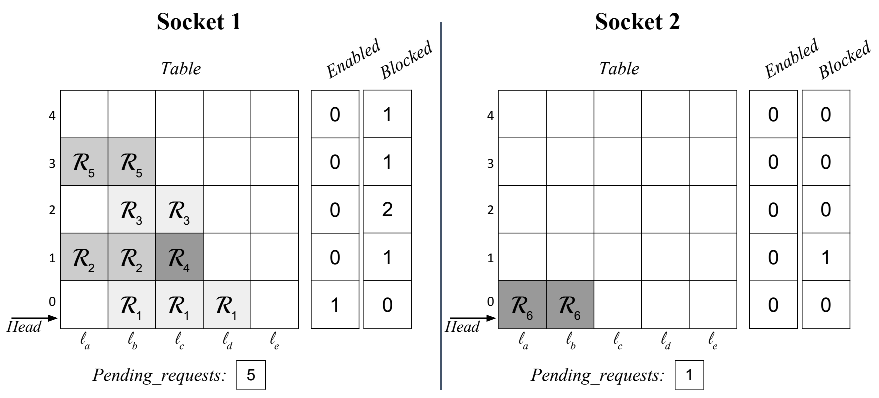


Figure 9 $\mathcal{R}_6$ is added to *Table* of Socket 2.

Algorithm 3 Static Local Lock Server.

```
1: procedure SLLS(core: array of ptr to core_data, s: socket identifier)
2:     Service lock and unlock requests like in Alg. 1, but with the following changes:
3:         Only requests from the local socket $s$ are handled
4:         Coordinate Phase with other lock server
5:         Set spin_location := TRUE for requests that are eligible to be satisfied while $Phase = s$
```

▶ **Example 2** (continued). Suppose that the requests in Fig. 8 were actually issued on Socket 1. Suppose now a request $\mathcal{R}_6$ for $D_6 = \{\ell_a, \ell_b\}$ is issued on Socket 2. This results in the two lock states shown in Fig. 9. Though $\mathcal{R}_6$ is the only request in $\mathcal{LS}_2$'s lock state, it should not be satisfied, as it conflicts with request $\mathcal{R}_1$ for resource ℓ_b. Thus, it must wait.

To mediate requests from the two lock servers, we propose to let them alternate execution in phases. In App. A, we present a phase-management protocol to coordinate these phases. In the U-C-RNLP, a natural way to define which requests belong to a certain phase is to let each row of *Table* indicate a phase. As shown in App. A, when defining and managing phases in this way, the blocking experienced by request $\mathcal{R}_i$ is at most $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units, where $c_{i,s}$ is the contention $\mathcal{R}_i$ experiences on Socket s and $L_{max,s}$ is the maximum critical-section length on Socket s. In Alg. 3, this boundary and change between phases is coordinated in Line 4 and the current phase is stored in the variable *Phase*. The coordination must ensure bounded time before a change of *Phase* when requests are waiting on the other socket. Thus, in Line 5, a request must be *able* to be satisfied (*e.g.*, it is in the active row of *Table* in the U-C-RNLP) and the phase must be set to the local socket before the request can be *marked* as satisfied by updating its spin location.

4 Floating Lock Servers

In the prior section, we implicitly assumed that static lock server(s) are to be supported by devoting full core(s) to them. While this may be reasonable on a large platform, we could instead allow other work to execute on the core(s) assigned to static lock servers(s) as long as that work executes at a lower priority. The impact lock servers have on such work could be assessed similarly to how interrupt accounting is done.

In this section, we explore a simpler alternative: floating lock servers. When using static lock servers, every request executes a spin loop for each server interaction in order to wait for a response. When using floating lock servers, the processor time wasted during these spin

Algorithm 4 Floating Global/Local Lock Server

```
 1: global var Server_exists: boolean initially FALSE

 2: procedure FLOATING-LOCK(c: ptr to core_data, D: set of resources)
 3:     var i_am_server: boolean initially FALSE
 4:     c→requested := D
 5:     c→service := LOCK_SERVICE
 6:     i_am_server := WAIT-UNTIL(^(c→service), NULL)
 7:     if (i_am_server = FALSE):
 8:         i_am_server := WAIT-UNTIL(^(c→spin_location), TRUE)
 9:     if (i_am_server = TRUE):
10:         while (c→service ≠ NULL) or (c→spin_location ≠ TRUE):        ▷ Until satisfied, be server
11:             Perform lock server functionality
12:         Server_exists := FALSE

13: procedure FLOATING-UNLOCK(c: ptr to core_data, D: set of resources)
14:     var i_am_server: boolean initially FALSE
15:     c→requested := D
16:     c→service := UNLOCK_SERVICE
17:     i_am_server := WAIT-UNTIL(^(c→service), NULL)
18:     if (i_am_server = TRUE):
19:         if c→service ≠ NULL:                                   ▷ This request has not been serviced
20:             Perform unlock for this request
21:         Server_exists := FALSE

22: procedure WAIT-UNTIL(location: ptr, value)
23:     var t: unsigned int
24:     t := TestAndSet(&Server_exists)
25:     while (t = TRUE) and (*location ≠ value):
26:         if (Server_exists = FALSE):
27:             t := TestAndSet(&Server_exists)        ▷ TestAndSet return value of FALSE means . . .
28:     return (t = FALSE)                          ▷ . . . Server_exists was FALSE so I am now server
```

loops is reclaimed to execute lock-server code. This approach is tantamount to employing a *helping mechanism* [39], but unlike the traditional sense of helping, where one request may help another to complete a *critical section*, a request here performs only *lock logic* on behalf of another request. We describe the floating lock-server paradigm more fully below by first considering global servers and then local ones.

4.1 A Floating Global Lock Server

In this section, we more carefully describe the notion of a floating global lock server. Unlike static lock servers, in floating ones, request code and lock-server code are inextricably linked. Thus, we specify how a floating global lock server works via one code listing in Alg. 4.

In Alg. 4, a request in its lock call performs the same logic as it would using a static server (marking itself as requiring service, waiting for a location on which to spin, and then spinning), with intermediate checks to ensure that some request is acting as the lock server. The existence of a lock server is maintained in the global variable *Server_exists*. The helper method WAIT-UNTIL waits until a designated location holds a desired value, with the waiting terminated if the caller becomes the server (as determined in a test-and-test-and-set manner). The return value of this method indicates whether the caller is now the server.

Examining the FLOATING-LOCK routine in a bit more detail, a request first marks that it is ready to be serviced (Line 5). Then it waits to be serviced (Line 6). If it is not the lock server, then it spins on *spin_location* (Line 8). If it becomes the lock server, then it performs the lock server functionality until it is satisfied (Lines 10-11). Notice that whenever a request functions as the lock server here it would have been spinning in the global static lock server paradigm waiting for a server response.

The Floating-Unlock routine is similar, except that a request that becomes the lock server only services itself (Line 20). This is because an unlock does not involve blocking, so servicing other requests would not replace useless spinning, but would just slow the unlock.

4.2 Floating Local Lock Servers

While a floating global lock server has the benefit over static lock server(s) of not requiring dedicated core(s), it also would be expected to suffer higher overhead due to eroded cache affinity when lock state moves between sockets. Fortunately, there is a quick fix to keep lock state in cache: implement a floating *local* lock server. In this paradigm, a request can only perform the functions of the lock server for the socket from which it was issued. By restricting to a single socket, L3 cache affinity can be maintained. A floating local lock server uses the structure found in Alg. 4, but with the server logic in Lines 11 and 20 being that of a local lock server (with phase arbitration).

5 Handling Non-Uniform Requests

Recall from Sec. 3 that the C-RNLP is defined in an abstract rule-based way and that the U-C-RNLP is just one implementation of it [43]. The U-C-RNLP can be used to handle non-uniform requests by pessimistically viewing all critical sections as L_{max}. However, this changes the worst-case blocking bound of the general version from $\min(mL_{max}, c_i(L_{max} + L_i))$ to $\min(m, (c_i{+}1))L_{max}$ [43]. In this section, we discuss an alternate non-uniform implementation, denoted as the G-C-RNLP, that maintains the original bound.

The G-C-RNLP uses $|D_i|$ nodes to represent a request $\mathcal{R}_i$, one corresponding to each resource in D_i. A separate queue is maintained for each resource in the system. When $\mathcal{R}_i$ is processed, a satisfaction time is recorded for it by considering the satisfaction times for other requests and the critical-section length of each. Then, the queue for each resource in D_i is updated by inserting a node for $\mathcal{R}_i$ at a position that ensures that $\mathcal{R}_i$ will be at the head of its respective queues by its recorded satisfaction time. This protocol would likely give rise to prohibitively high overhead if the tasks themselves were to execute the queuing logic concurrently. In particular, when enqueuing a request $\mathcal{R}_i$, $|D_i|$ queues must be checked for the satisfaction times of existing requests and $|D_i|$ nodes must be inserted (sometimes in the middle of queues). However, if this protocol is implemented using lock servers,[3] then the overhead becomes quite reasonable, as we show in Sec. 6.

Using global lock servers (Secs. 3.1 and 4.1) to implement the G-C-RNLP is straightforward: we merely use the G-C-RNLP instead of the uniform C-RNLP in the LS-Lock and LS-Unlock routines. On the other hand, using local lock servers (Secs. 3.2 and 4.2) is more problematic due to the phase management such servers require. We show why by considering two examples. For the time being, we assume that a basic phase-management protocol called *Greedy Satisfaction* is used that allows only requests that can be satisfied at the start of a phase to be satisfied during that phase.

[3] Although not reflected in the pseudocode given in this paper, our lock-server implementations have been carefully honed using bit-vector operations and other techniques to improve efficiency. All of our code is publicly available online [56].

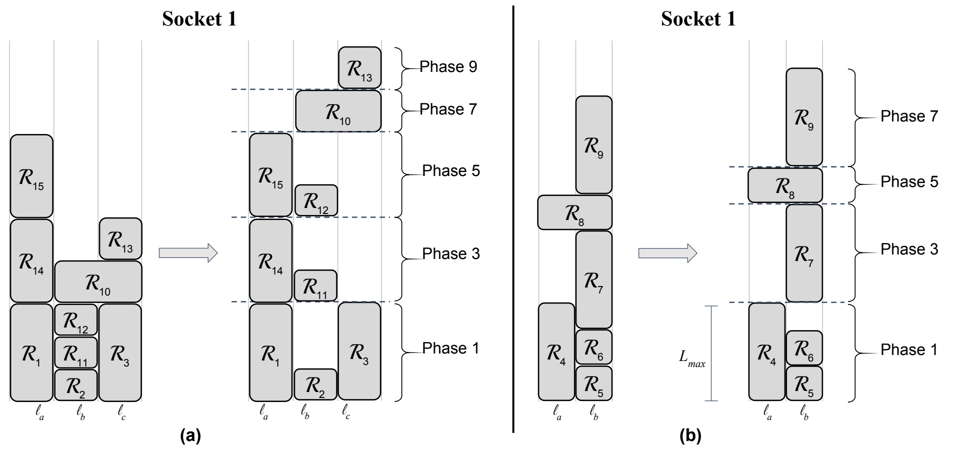


Figure 10 Scenarios with complicated phase management.

▶ **Example 3.** Consider the requests shown in Fig. 10(a), all issued on Socket 1. $\mathcal{R}_2$, $\mathcal{R}_{11}$, and $\mathcal{R}_{12}$ are "short" requests for resource ℓ_b and most of the other requests (for various resources) are longer. Under Greedy Satisfaction, requests would be satisfied in phases as shown in the right half of Fig. 10(a), with dashed lines indicating phase boundaries. Observe that, under this policy, only $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are satisfied in the first phase. $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ are satisfied later. Notice that all of the phases have odd indicies. This is because Socket 2 executes during even-indexed phases.

Ex. 3 shows that Greedy Satisfaction can unnecessarily delay requests: $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ both could have completed by the time $\mathcal{R}_3$ completed. Instead, they are moved to two later phases. We call this the *Long-Short Problem*: when requests vary in length, shorter requests can be delayed, further delaying other requests. In this example, $\mathcal{R}_{13}$ in particular is delayed substantially by requests with which it does not conflict.

Ex. 3 highlights the fact that, for some protocols, Greedy Satisfaction is inadequate. A better solution is a policy we call *Timed Satisfaction*, which allows requests that can finish within L_{max} time units to be satisfied in the same phase.

▶ **Example 4.** In Fig. 10(b), we apply Timed Satisfaction to a different set of requests on Socket 1. On the left, the requests are shown as they are ordered by the G-C-RNLP. On the right, the requests are shifted to occupy the phases the lock server would enforce. $\mathcal{R}_4$ and $\mathcal{R}_5$ are satisfied at the start of Phase 1. After $\mathcal{R}_5$ completes, $\mathcal{R}_6$ is also satisfied in this phase. However, after $\mathcal{R}_6$ completes, $\mathcal{R}_7$ cannot be satisfied, as it cannot be guaranteed to complete within L_{max} time units from the start of the phase. Therefore, $\mathcal{R}_7$ must wait until Socket 2 is allowed another phase, namely, Phase 3.

Ex. 4 illustrates another source of added blocking: $\mathcal{R}_7$ is forced to delay until the start of the next phase to be satisfied. Even if we were to increase the time window, the problem could arise again: another request could be issued that cannot complete within the window. We call this difficulty the *Overlap Problem*. A phase must end at some point to prevent the starvation of requests on the other socket. Whatever value we choose, we may have requests that would overlap a phase boundary and need to be delayed. The Overlap Problem can force a request that could otherwise be satisfied to be delayed until the current phase of its lock server completes followed by a full phase of the other lock server before being satisfied.

When considering the effect of local lock servers on blocking with the G-C-RNLP, we assume Timed Satisfaction is the phase-management policy used. (Again, the issues just discussed are unique to local servers.) As seen in Ex. 4, Timed Satisfaction is susceptible to the Overlap Problem. This is the reason why the worst-case blocking bounds presented in App. A for the G-C-RNLP are worse than those for the U-C-RNLP.

6 Evaluation

Our primary reason for exploring lock servers is to minimize overhead by keeping all lock state cache hot. For static global and static local servers, cache hot means the lock state should maintain L1 cache affinity on our platform, whereas a floating local server should tend to execute out of its socket's L3 cache. On the other hand, a floating global server will likely not be able to maintain much cache affinity if tasks execute on more than one socket.

Given these expectations, a number of questions arise. How do the different lock-server paradigms presented previously differ with respect to overhead, and do these differences match the above expectations? To what extent do lock servers lower overhead compared to not using lock servers? Are the overhead improvements enough to make contention-sensitive locking practical? How do lock servers scale with increasing core counts?

To answer these questions, we conducted an experimental study. Before covering the results revealed by our study, we first describe our experimental setup.

Experimental setup. Recall from Sec. 3 that our test platform is a dual-socket, 18-cores-per-socket platform. We used this platform to evaluate the lock-server paradigms discussed previously by conducting experiments involving tasks that repeatedly issue lock and unlock calls for random resources. We varied the number of tasks, n, number of resources, n_r, nesting depth (which defines the number of resources required for request R_i), $\mathbb{D} = |D_i|$, and critical-section length, L_i, to evaluate each parameter's effect on overhead and blocking. We define a *scenario* as an assignment of values to three of these parameters while varying the fourth. We considered the following parameter ranges: $n \in \{2, 4, ..., 36\}$, $n_r \in \{16, 32, 64\}$, $\mathbb{D} \in \{1, 2, 4, ..., 10\}$, and $L_i \in \{1\mu s, 20\mu s, 40\mu s, ..., 100\mu s\}$. In our experiments, all requests in a scenario have the same nesting depth. Unless stated otherwise, they also all have the same critical-section length L_i.

We recorded overhead and blocking times at user level, with one task pinned to each core. This setup ensures that requests execute non-preemptively. For a given scenario, we configured each task to perform 10,000 lock and unlock calls, with critical sections simulated by spinning for a duration of L_i. For task systems running on at most 18 cores, we used only the cores on one socket. When using more than 18 cores, all cores on Socket 1 were used with the remainder on Socket 2. Our workload is comprised solely of tasks making lock and unlock calls as described above. Thus, our evaluation focuses on cache affinity losses inherent to running a protocol and ignores potential evictions from other tasks; there exist techniques to keep cache affinity in some systems [5, 22, 25, 40, 46, 47, 66, 69, 70, 72].

In the graphs that follow, we plot 99th percentile measurements as worst-case values to filter out any spurious measurements caused by performing measurements at user level.[4] Across over 150 scenarios, we generated approximately 1,000 graphs. The graphs shown in this section were chosen as examples of trends seen across the entire collection of graphs. The full set can be found in an online appendix [56].

[4] This filtering does not guarantee smoothness of all curves.

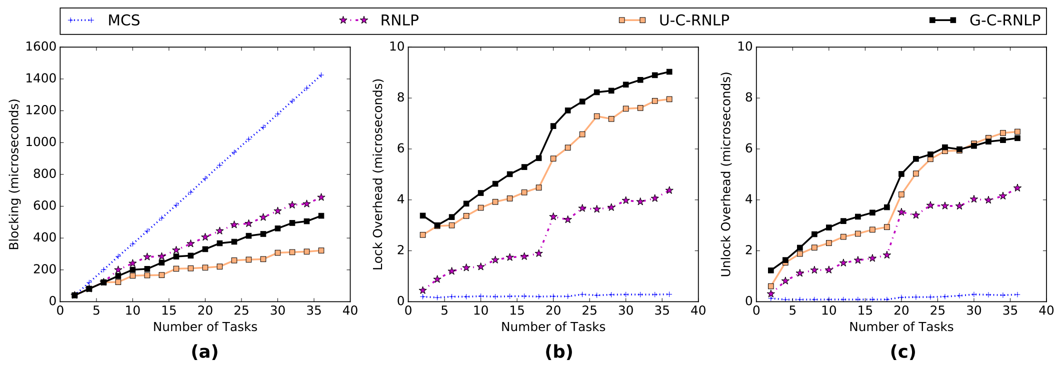


Figure 11 Blocking and lock/unlock overhead when no lock servers are used. For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

Overhead and blocking without lock servers. Before delving into results pertaining to lock-server paradigms, we examine a range of server-less implementation options. To gauge the tradeoffs involved in supporting lock nesting, we experimentally evaluated two contention-sensitive options, the U-C-RNLP and the G-C-RNLP, both implemented without lock servers, and the RNLP, which supports nesting but is not contention sensitive. As a baseline, we evaluated coalescing all resources under one MCS queue lock [52]. We conducted experiments in which these options were compared on the basis of blocking and overhead.

We now state several observations that follow from the full range of scenarios considered in these experiments. We illustrate these observations using the graphs in Fig. 11.[5] In this figure, we present lock and unlock overhead separately to demonstrate their relative scale: enqueuing takes slightly longer than dequeuing, but both operations require manipulating lock state, and thus both contribute to overhead. In later figures, we will combine lock and unlock overhead to yield one overhead graph.

▶ **Obs. 1.** *Without using lock servers, both C-RNLP variants have dramatically higher overhead than MCS.*

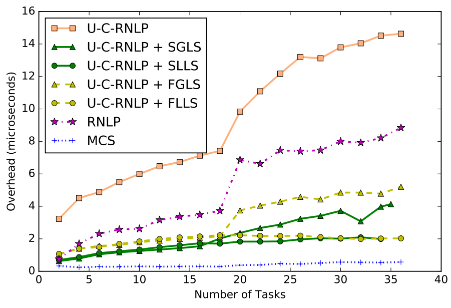
This is expected behavior, as MCS implements just a single spin queue. As shown in insets (b) and (c) of Fig. 11, the U-C-RNLP has lock and unlock overhead up to 27.4 and 23.9 times that of MCS, respectively. For the G-C-RNLP, these values are similarly high: up to 31.1 and 22.9 times, respectively.

▶ **Obs. 2.** *Compared to MCS, contention-sensitive protocols demonstrate significantly better blocking bounds as the number of requests increases.*

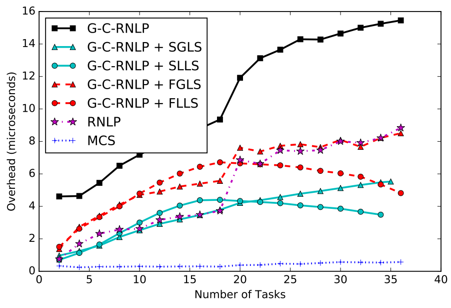
The low overhead of MCS (Obs. 1) comes at the expense of unscalable blocking. As shown in Fig. 11(a), worst-case blocking under MCS grows up to 5.3 and 2.9 times faster than that under the U-C-RNLP and G-C-RNLP, respectively.

Considering the RNLP is instructive because it provides some insights into the extra cost of providing contention sensitivity in addition to handling lock nesting. As shown in insets (b) and (c) of Fig. 11, lock and unlock overhead under the U-C-RNLP (resp., G-C-RNLP) are up to 1.8 and 2.1 (resp., 1.5 and 1.4) times that under the RNLP, respectively.

[5] In every such figure that we consider, the applicable scenario is stated in the figure's caption. Note that not all curves extend to $n = 36$. This is because up to two cores are reserved for lock servers and this number is scheme-dependent.



(a) U-C-RNLP overhead.



(b) G-C-RNLP overhead.

Figure 12 For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

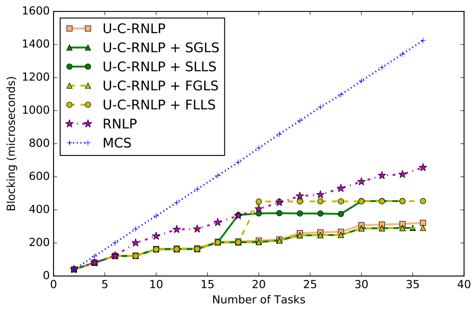


Figure 13 Worst-case blocking for the scenario in Fig. 12(a).

Applying lock servers. In Secs. 3 and 4, we presented four lock-server paradigms, each of which can be applied to any locking protocol. We conducted experiments to explore how these paradigms differ when used to implement the U-C-RNLP and the G-C-RNLP. We now state several observations that follow from the full range of scenarios considered in these experiments. We illustrate these observations using the graphs in Figs. 12 and 13. In Fig. 12(a), we compare the four possible lock-server variants of the U-C-RNLP against the baselines of MCS, the RNLP, and the U-C-RNLP without lock servers. Fig. 12(b) is similar, but is directed at the G-C-RNLP instead of the U-C-RNLP. (We abbreviate lock-server paradigms in figure captions, *e.g.*, static global lock server is SGLS.)

▶ **Obs. 3.** *Using lock server(s) results in significantly lower overhead.*

This can be seen both in Fig. 12(a) for the U-C-RNLP and in Fig. 12(b) for the G-C-RNLP. Observe that using lock server(s) usually resulted in overhead even lower than that of the RNLP. In fact, using local lock servers in this scenario reduced the overhead of the U-C-RNLP and the G-C-RNLP by up to 86% and 77%, respectively.

▶ **Obs. 4.** *When there are requests on only one socket, static lock servers result in the largest overhead reduction.*

This trend appears consistently in our results, and matches our intuition, as a static lock server can maintain L1 cache affinity. In Fig. 12, only one socket is used when $n < 18$ (it is strictly less because the lock server uses one core).

▶ **Obs. 5.** *When considering requests on two sockets, as the number of tasks increases, the overhead of local lock servers scales better than that of a global lock server.*

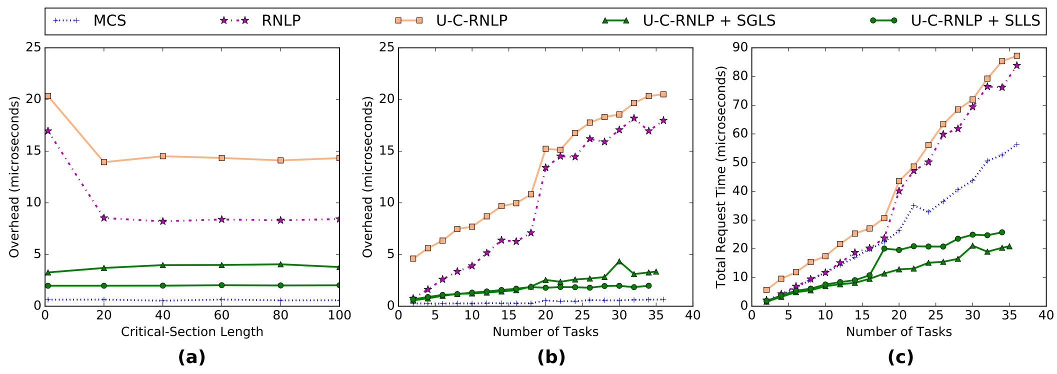


Figure 14 **(a)** Overhead as a function of critical-section length, for $n = 34, n_r = 64$, and $\mathbb{D} = 4$. **(b)** Overhead and **(c)** blocking as a function of n, for $n_r = 64, \mathbb{D} = 4$, and $L_i = 1\mu s$ for all i.

For example, in Fig. 12, the overhead of the U-C-RNLP (resp., G-C-RNLP) with floating local lock servers is up to 61% (resp., 43%) lower than with a floating global lock server.

▶ **Obs. 6.** *Floating global lock servers scale the poorest of the four lock-server paradigms.*

This observation is entirely expected and clearly evident in Fig. 12. Note that a floating global lock server still reduces overhead to be comparable to or better than the RNLP.

In Fig. 13, worst-case blocking under the U-C-RNLP is plotted for each lock-server paradigm for the same scenario presented in Fig. 12.

▶ **Obs. 7.** *Moving from one socket to two can negatively impact blocking of local lock servers.*

This observation is evident in Fig. 13. Two local lock servers are required if $n \geq 18$. The extra blocking is due to phase management and request imbalances between the two sockets. For example, for $n = 18$ there are 17 requests on Socket 1 and one request on Socket 2. The request on Socket 2 will have very low blocking, but requests on Socket 1 can experience twice as much blocking as when only one socket is in use. Without the mitigation in App. A, blocking scales poorly with increasing socket counts (*e.g.*, a four-socket platform [56]).

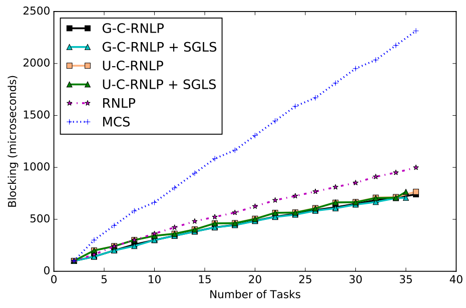
Requests with short critical sections. Inset (a) of Fig. 14 plots overhead as a function of critical-section length, while insets (b) and (c) provide data for a scenario with a short critical section of $1\mu s$. (The G-C-RNLP variants are omitted from this figure for clarity; overhead for them is higher than their U-C-RNLP counterparts but follows similar trends.) Such short critical sections result in overhead being a higher proportion of total request time (overhead plus blocking). Note that the blocking time of a request includes the overhead of any request upon which it must wait, so reducing overhead additionally reduces blocking.

▶ **Obs. 8.** *Overhead is (mostly) constant for all U-C-RNLP variants with respect to L_i.*

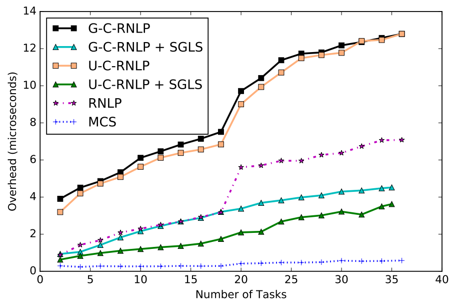
This is demonstrated in Fig. 14(a). Note that, when static lock servers are used, overhead remains low even for small L_i.

▶ **Obs. 9.** *When critical sections are short, lock servers greatly reduce the impact of overhead on total request time.*

The data in insets (b) and (c) of Fig. 14 indicates that, under the U-C-RNLP, requests with $1\mu s$ critical sections can experience worst-case overhead that is up to 23.4% of the total request time. When using a static local lock server, this is reduced to 9.6%.



(a) Blocking.



(b) Overhead.

Figure 15 For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for 75% of requests and $L_i = 100\mu s$ for the remaining 25% of requests.

	U-C-RNLP	U-C-RNLP + SGLS	U-C-RNLP + SLLS	U-C-RNLP + FGLS	G-C-RNLP + SGLS
Total Firsts	0	92	0	23	12
Total Seconds	1	26	18	70	4
Total Thirds	68	2	17	20	8
Total	69	120	35	113	24

Figure 16 Results of total request time comparison.

A case where the G-C-RNLP wins. From the results presented thus far, it is tempting to discount the G-C-RNLP entirely. In cases where all critical sections are of the same duration, the G-C-RNLP suffers worse overhead and blocking than the U-C-RNLP. We now explore scenarios in which the G-C-RNLP has very competitive worst-case blocking; this occurs when a large fraction of requests have critical-section lengths much less than L_{max}. Such a scenario is depicted in Fig. 15.

▶ **Obs. 10.** *When most requests have critical sections much shorter than L_{max}, the G-C-RNLP and U-C-RNLP have similar performance when both use a static global lock server.*

In Fig. 15, the G-C-RNLP with a static global lock server has lower blocking and only slightly higher overhead than the U-C-RNLP with the same lock-server setup.

Overall winner. Judging the lock-server paradigms should be done with a specific workload, but to make a general summary, we determined the "best" paradigm to the extent possible in our experimental framework as follows. For each considered scenario,[6] we calculated a single "total request time" score (blocking plus overhead) for each protocol variant by approximating the area under its curve using a midpoint Riemann sum. We then ranked the protocol variants for that scenario. Fig. 16 gives the total number of first-, second-, and third-place finishes for each protocol variant. The U-C-RNLP with a static global lock server was the overall winner. However, our experimental setup mostly generates scenarios in which critical sections are uniform, which tends to make the G-C-RNLP variants less competitive. Still, these results show there is value in using lock servers.

[6] We filtered out scenarios with $\mathbb{D} \in \{8, 10\}$, as they require nearly coalescing all resources under a single lock, which has non-contention-sensitive blocking.

7 Conclusion

In this paper, we have considered for the first time the use of lock servers on large multicore platforms to lessen overhead associated with contention-sensitive real-time locking protocols, without modifying the associated pi-blocking bounds. We proposed four specific lock-server paradigms and presented an experimental study in which the overhead reductions enabled by these paradigms was assessed. This study showed that such reductions can be dramatic. For example, the paradigm that generally performed best, static global lock servers, typically exhibited overhead reductions in the range 25%–75% compared to not using lock servers.

This paper is certainly not the last word on lock servers. Indeed, we hope that our work sparks further interest by others in this topic and more broadly raises an appreciation for investigating scalability issues affecting real-time resource-allocation methods as core counts continue to climb. With respect to lock servers themselves, a number of avenues for further research come to mind. First, while we have limited attention to spin-based locking protocols, the very notion of a lock server lends itself to an operating-system-based implementation. In that setting, suspension-based protocols warrant detailed consideration. Second, we have focused on one particular large multicore platform as an exemplar. Other platforms, including manycore platforms with different interconnects, warrant further study. Third, it would be interesting to apply the ideas in this paper to support transactions in a real-time database. In fact, a contention-sensitive real-time locking protocol together with lock server(s) can be thought of as a lock-based variant of software transactional memory that targets real-time applications. Fourth, we have focused herein on the extent to which lock servers can lower overhead. In the future, we will assess the *schedulability*-related impacts of different lock-server deployments, which will require investigating lock server behavior in the context of more complex workloads and exploring task balancing among lock servers. Finally, in a hard real-time system, it might be necessary to *provably* ensure that lock servers always execute in cache. Such assurances could be provided by integrating lock servers with cache-isolation techniques explored elsewhere [5, 22, 25, 40, 46, 47, 66, 69, 70, 72].

References

1 S. Afshar, M. Behnam, R. Bril, and T. Nolte. Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In *SIES '14*, pages 41–51. IEEE, 2014. `doi:10.1109/SIES.2014.6871185`.

2 S. Afshar, M. Behnam, R. Bril, and T. Nolte. An optimal spin-lock priority assignment algorithm for real-time multi-core systems. In *RTCSA '17*, pages 1–11. IEEE Computer Society, 2017. `doi:10.1109/RTCSA.2017.8046310`.

3 S. Afshar, M. Behnam, R. Bril, and T. Nolte. Per processor spin-based protocols for multiprocessor real-time systems. *Leibniz Transactions on Embedded Systems*, 4(2), 2017. `doi:10.4230/LITES-v004-i002-a003`.

4 S. Afshar, F. Nemati, and T. Nolte. Towards resource sharing under multiprocessor semi-partitioned scheduling. In *SIES '12*, pages 315–318. IEEE, 2012. `doi:10.1109/SIES.2012.6356605`.

5 S. Altmeyer, R. Douma, W. Lunniss, and R. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS '14*, pages 15–26. IEEE Computer Society, 2014. `doi:10.1109/ECRTS.2014.11`.

6 B. Andersson and A. Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010.

7 D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI '98*, pages 258–268, 1998. `doi:10.1145/277650.277734`.

8 A. Biondi and B. Brandenburg. Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors. In *ECRTS '16*, pages 39–49. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.30.

9 A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, pages 47–56. IEEE Computer Society, 2007. doi:10.1109/RTCSA.2007.8.

10 B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

11 B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS '13*, pages 141–152. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531087.

12 B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS '14*, pages 61–71. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.26.

13 B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT '07*, 2007.

14 B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *OPODIS '08*, pages 105–124, 2008. doi:10.1007/978-3-540-92221-6_9.

15 B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *RTCSA '08*, pages 185–194, 2008. doi:10.1109/RTCSA.2008.13.

16 B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*, pages 49–60. IEEE Computer Society, 2010. doi:10.1109/RTSS.2010.17.

17 B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1), 2010.

18 B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*, pages 69–78. ACM, 2011. doi:10.1145/2038642.2038655.

19 B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2013.

20 B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS '08*, pages 342–353. IEEE Computer Society, 2008. doi:10.1109/RTAS.2008.27.

21 A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS '13*, pages 282–291. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.37.

22 M. Campoy, A.P. Ivars, and J.V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop '01*, 2001.

23 Y. Chang, R. Davis, and A. Wellings. Reducing queue lock pessimism in multiprocessor schedulability analysis. In *RTNS '10*, 2010.

24 C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. Dept. of Computer Science, Univ. of Maryland. Technical report, CS-TR-3252, April, 1994.

25 M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS '15*, pages 305–316. IEEE Computer Society, 2015. doi:10.1109/RTSS.2015.36.

26 T. Craig. Queuing spin lock algorithms to support timing predictability. In *RTSS '93*, pages 148–157. IEEE Computer Society, 1993. doi:10.1109/REAL.1993.393505.

27 R. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *RTSS '06*, pages 257–270. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.42.

28 U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *ECRTS '06*, pages 75–84. IEEE Computer Society, 2006. `doi:10.1109/ECRTS.2006.10`.

29 E. Dijkstra. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands.

30 A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS '09*, pages 377–386, 2009. `doi:10.1109/RTSS.2009.37`.

31 G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170, 2013.

32 D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *ECRTS '10*, pages 90–99, 2010. `doi:10.1109/ECRTS.2010.19`.

33 D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6), 2012.

34 P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *RTAS '03*, page 189, 2003. `doi:10.1109/RTTAS.2003.1203051`.

35 P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*, pages 73–83. IEEE Computer Society, 2001. `doi:10.1109/REAL.2001.990598`.

36 J. Garrido, S. Zhao, A. Burns, and A. Wellings. Supporting nested resources in MrsP. In *Ada-Europe International Conference on Reliable Software Technologies '17*, volume 10300 of *Lecture Notes in Computer Science*, pages 73–86. Springer, 2017. `doi:10.1007/978-3-319-60588-3_5`.

37 J. Han, D. Zhu, X. Wu, L. Yang, and H. Jin. Multiprocessor real-time systems with shared resources: Utilization bound and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

38 J. Havender. Avoiding deadlock in multitasking systems. *IBM systems journal*, 7(2):74–84, 1968.

39 M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

40 J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *ECRTS '11*, pages 23–32. IEEE Computer Society, 2011. `doi:10.1109/ECRTS.2011.11`.

41 P. Hsiu, D. Lee, and T. Kuo. Task synchronization and allocation for many-core real-time systems. In *EMSOFT '11*, pages 79–88. ACM, 2011. `doi:10.1145/2038642.2038656`.

42 W. Huang, M. Yang, and J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *RTSS '16*, pages 111–122. IEEE Computer Society, 2016. `doi:10.1109/RTSS.2016.020`.

43 C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS '15*, pages 3–12. ACM, 2015. `doi:10.1145/2834848.2834874`.

44 Y. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

45 P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *PODC '99*, pages 23–32, 1999. `doi:10.1145/301308.301319`.

46 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS '13*, pages 80–89. IEEE Computer Society, 2013. `doi:10.1109/ECRTS.2013.19`.

47 D. Kirk and J. Strosnider. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *RTSS '90*, pages 322–330. IEEE Computer Society, 1990. doi:10.1109/REAL.1990.128764.

48 L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems (TOCS)*, 15(1):3–40, 1997.

49 K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS '09*, pages 469–478. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.51.

50 J. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC'12*, pages 65–76. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

51 G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *ECRTS '11*, pages 262–271. IEEE Computer Society, 2011. doi:10.1109/ECRTS.2011.32.

52 J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization of shared-memory multiprocessors. *Transactions on Computer Systems*, 9(1), 1991.

53 F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS '11*, pages 251–261. IEEE Computer Society, 2011. doi:10.1109/ECRTS.2011.31.

54 F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *OPODIS '10*, volume 6490 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2010. doi:10.1007/978-3-642-17653-1_20.

55 C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: Optimizing the common case. In *RTNS '17*, pages 38–47. ACM, 2017. doi:10.1145/3139258.3139262.

56 C. Nemitz, T. Amert, and J. Anderson. Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts (extended version), 2018. URL: http://www.cs.unc.edu/~anderson/papers.html.

57 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS '90*, pages 116–123. IEEE Computer Society, 1990. URL: https://doi.org/10.1109/ICDCS.1990.89257, doi:10.1109/ICDCS.1990.89257.

58 R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

59 R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS '88*, pages 259–269, 1988. doi:10.1109/REAL.1988.51121.

60 H. Takada and K. Sakamura. Real-time scalability of nested spin locks. In *RTCSA '95*, pages 160–167, 1995. URL: https://doi.org/10.1109/RTCSA.1995.528766, doi:10.1109/RTCSA.1995.528766.

61 C. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *ISPAN '96*, pages 70–76. IEEE Computer Society, 1996. doi:10.1109/ISPAN.1996.508963.

62 B. Ward. *Sharing Non-Processor Resources in Multiprocessor Real-Time Systems.* PhD thesis, University of North Carolina, Chapel Hill, NC, 2016.

63 B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*, pages 223–232. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.17.

64 B. Ward and J. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *RTNS '13*, pages 67–76. ACM, 2013. doi:10.1145/2516821.2516843.

65 B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *IPDPS '14*, pages 177–186. IEEE Computer Society, 2014. `doi:10.1109/IPDPS.2014.29`.

66 B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*, pages 157–167. IEEE Computer Society, 2013. `doi:10.1109/ECRTS.2013.26`.

67 A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS '13*, pages 45–56. IEEE Computer Society, 2013. `doi:10.1109/RTSS.2013.13`.

68 A. Wieder and B. Brandenburg. On the complexity of worst-case blocking analysis of nested critical sections. In *RTSS '14*, pages 106–117. IEEE Computer Society, 2014. `doi:10.1109/RTSS.2014.34`.

69 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS '16*, 2016.

70 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *RTAS '17*, pages 211–222, 2017. `doi:10.1109/RTAS.2017.15`.

71 M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *RTSS '15*, pages 1–12. IEEE Computer Society, 2015. `doi:10.1109/RTSS.2015.8`.

72 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS '14*, pages 155–166. IEEE Computer Society, 2014. `doi:10.1109/RTAS.2014.6925999`.

73 S. Zhao, J. Garrido, A. Burns, and A. Wellings. New schedulability analysis for MrsP. In *RTCSA '17*, pages 1–10. IEEE Computer Society, 2017. `doi:10.1109/RTCSA.2017.8046311`.

A Local Lock Server Phase Management and Blocking Bounds

In this appendix, we provide additional details concerning the phase-management protocol needed for the local lock servers described in Secs. 3.2 and 4.2. Such a server must determine which requests will execute in each of its phases in addition to managing phase changes.

Request selection. We restrict phases on Socket s to execute for at most the maximum critical-section length on that socket, denoted $L_{max,s}$. For the U-C-RNLP, the requests in a phase are determined by selecting the row in *Table* pointed to by *Head*. For the G-C-RNLP, Timed Satisfaction (recall Sec. 4.2) is used instead.

Phase coordination. Because all requests that can be satisfied simultaneously under C-RNLP rules can run concurrently relative to each other, they may be processed like read requests. With this in mind, the synchronization mechanism we need can be obtained by building on the idea of a phase-fair reader/writer lock [17]. Such a lock supports two kinds of requests, *reads* and *writes*, which execute in phases that alternate if both kinds of requests are present, where any number of reads can occur during a read phase but only one write during a write phase. The synchronization mechanism we desire similarly needs to support two kinds of requests that execute in alternating phases, but in our case, any number of requests can execute in a given phase. That is, we need a *reader/reader* lock. To our knowledge, such locks have not been studied in the context of real-time systems, so we present a new phase-fair reader/reader locking algorithm with corresponding blocking bounds in an online appendix [56]. (The phase-fair reader/reader problem is similar to the group mutual exclusion problem [44, 45] except that we require $O(1)$ pi-blocking bounds.)

Using this reader/reader lock, it is straightforward to support phase management in a way that satisfies the following general properties.

- Each lock server is either *active* or *passive* and at most one lock server is active at any time. A maximal interval of time when a lock server is active is called a *phase*.
- A request can be satisfied only if its lock server is active and if it can be satisfied under the variant of the C-RNLP being used by that server.
- A passive lock server with unsatisfied requests becomes active within L_{max} time units.
- All requests satisfied in a phase finish by the end of that phase.

Based on these properties, we prove the worst-case acquisition-delay bounds stated below in an online appendix [56]. In stating these bounds, recall that $\mathcal{LS}_s$ denotes the local lock server on Socket s. Also, we denote the contention a request $\mathcal{R}_i$ experiences on Socket s as $c_{i,s}$. We call such a request *entitled* if it could be satisfied under the C-RNLP.

▶ **Theorem 5.** *A request $\mathcal{R}_i$ on socket s that is serviced by a local lock server running the U-C-RNLP will be satisfied within $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units.*

▶ **Theorem 6.** *A request $\mathcal{R}_i$ on Socket 1 (resp., Socket 2) that is serviced by a local lock server running the G-C-RNLP will be satisfied within $c_{i,1}(3L_{max,1}+2L_{max,2}+L_i)$ (resp., $c_{i,2}(2L_{max,1}+3L_{max,2}+L_i)$) time units.*

These bounds have implications regarding how to partition a workload under schedulers that assign tasks to execute on specific cores or clusters of cores. We illustrate this point in the context of the U-C-RNLP.

To begin, suppose that the requests for each resource can be evenly split between sockets such that $L_{max,1} = L_{max,2} = L_{max}$. Then, $c_{i,1} = c_{i,2} = \frac{1}{2}c_i$, and the blocking bound in Theorem 5 reduces to $(\frac{1}{2}c_i+1)(2L_{max}) = (c_i+2)L_{max}$, which is only one critical-section length longer than that for the original protocol.

While splitting contention evenly like this may be desirable, a system designer could instead choose to assign tasks so as to decrease $c_{i,1}$ at the expense of $c_{i,2}$, which may be a more effective strategy if critical sections of different lengths exist. To see this, suppose that a fraction α of all requests have critical sections of at most $\beta \cdot L_{max}$ time units, where $0 < \beta \leq 1$. If tasks can be assigned so that these shorter requests are all issued from Socket 1 and all others from Socket 2, then the bound from Theorem 5 becomes $(\alpha c_i+1)(\beta L_{max}+L_{max}) = (\alpha c_i+1)(\beta+1)L_{max}$ when applied to Socket 1, and $((1-\alpha)c_i+1)(\beta L_{max}+L_{max}) = ((1-\alpha)c_i+1)(\beta+1)L_{max}$ for Socket 2. Depending on the system, such a task assignment could lower the bounds applicable to all requests, as seen in the following example.

▶ **Example 7.** Suppose $c_i = 10$, $L_{max} = 100\mu$s, $\alpha = \frac{1}{5}$, and $\beta = \frac{1}{10}$. With the partitioning of requests described above, the bound on Socket 1 is $(\frac{1}{5}\cdot 10+1)(\frac{1}{10}\cdot 100+100)\mu s = 330\mu s$, and the bound on Socket 2 is $990\mu s$, both of which are lower than the bound of $(c_i+1)L_{max} = (10+1)100 = 1100\mu s$ for a server-less system (recall the U-C-RNLP discussion in Sec. 3).

Note that the improvement in the above example holds for both sockets, not just the one with lower critical-section lengths.

On Strong and Weak Sustainability, with an Application to Self-Suspending Real-Time Tasks

Felipe Cerqueira
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
felipec@mpi-sws.org

Geoffrey Nelissen
CISTER Research Centre, ISEP, Polytechnic Institute of Porto (IPP), Porto, Portugal
grrpn@isep.ipp.pt

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Motivated by an apparent contradiction regarding whether certain scheduling policies are sustainable, we revisit the topic of sustainability in real-time scheduling and argue that the existing definitions of sustainability should be further clarified and generalized. After proposing a formal, generic sustainability theory, we relax the existing notion of (strongly) sustainable scheduling policy to provide a new classification called weak sustainability. Proving weak sustainability properties allows reducing the number of variables that must be considered in the search of a worst-case schedule, and hence enables more efficient schedulability analyses and testing regimes even for policies that are not (strongly) sustainable. As a proof of concept, and to better understand a model for which many mistakes were found in the literature, we study weak sustainability in the context of dynamic self-suspending tasks, where we formalize a generic suspension model using the Coq proof assistant and provide a machine-checked proof that any JLFP scheduling policy is weakly sustainable with respect to job costs and variable suspension times.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability

Keywords and phrases real-time scheduling, sustainability, self-suspending tasks, machine-checked proofs

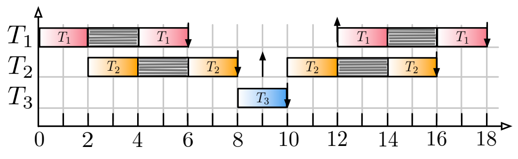
Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.26

Funding This work was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 391919384, and partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

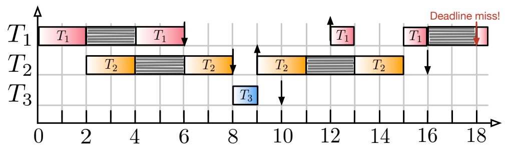
1 What Really is Sustainability?

Since the seminal paper by Liu and Layland [13], the analysis and certification of real-time systems has often relied on the fundamental notion of sustainability [5], which at a high level expresses the idea that "if a system is proven to be safe under extreme conditions, then it will remain safe if the conditions improve at runtime." By allowing system designers to focus on such extreme scenarios (rather than the entire state space of the system), sustainability plays a fundamental role in the design, prototyping, analysis, and validation of real-time systems.

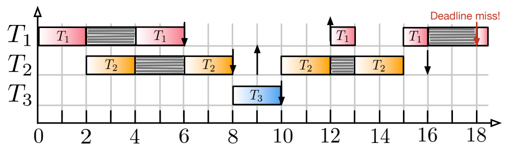
One common application of this principle is to determine the schedulability of the system by identifying worst-case scheduling scenarios. For example, any schedulability analysis for uniprocessor fixed-priority (FP) scheduling of sporadic tasks [14] that assumes that jobs



(a) EDF schedule of the original job set $\mathcal{J}$. No jobs are released after time 18.



(b) Scheduling anomaly represented by job set $\mathcal{J}_{better}$, generated by reducing the cost of task T_3.



(c) Alternative job set $\mathcal{J}_{susp}$ with original costs and shorter suspensions that is as hard to schedule as $\mathcal{J}_{better}$.

Figure 1 (adapted from [1]**)** Three schedules under the segmented suspension model showing that the impact of the lack of sustainability tightly depends on the considered task model. Despite the anomaly shown in schedule (b), there exists a harder schedule (c) with no anomaly that still incurs a deadline miss. If we assume that the task model allows variable suspension times, then any schedulability analysis that covers all possible scenarios would not claim the job set in (c) (and thus the task set) to be schedulable, regardless of the anomaly present in schedule (b).

execute for their worst-case execution time (WCET) or arrive at maximum rate exploits the fact that the FP scheduling policy for sporadic tasks is sustainable, *i.e.*, the occurrence of "better" job parameters (namely, larger inter-arrival times or lower execution times) at runtime does not cause any deadline miss.

While precursors to this concept were already identified and proven in earlier papers [10, 11], the general concept of sustainability was first formalized by Baruah and Burns [5, 6] and later refined by Baker and Baruah [2]. Although the definition by Baker and Baruah is more rigorous than the original definition, we argue in this paper that there is still a need for improvement in terms of clarity and precision.

To support our claim, in §1.1 and §1.2 we present an example in the context of uniprocessor scheduling with self-suspending tasks [16], where we show a scheduling policy that can be interpreted as both *sustainable* and *not sustainable* with respect to job execution times (also called job costs hereafter). Both claims are correct according to the existing definitions of sustainability and only depend on varying interpretations by the reader. This example shows that, despite being a well-established concept, the theory of sustainability needs further clarification and formalization.

1.1 Uniprocessor EDF Scheduling with Self-Suspensions is not Sustainable w.r.t. Job Costs

Consider uniprocessor earliest-deadline-first (EDF) scheduling of self-suspending tasks under the segmented suspension model. Self-suspending tasks are used to model workloads that may have their execution suspended at given times, for example, to perform remote operations on co-processors, acquire locks, wait for data, or synchronize with other tasks. The segmented self-suspending task model can be formalized as follows.

▶ **Definition 1** (Sporadic Task Model with Segmented Self-Suspensions)**.** Let τ be a task set and let $\mathcal{J}$ be a job set generated by τ. Each task $T_i \in \tau$ is defined by a period p_i, deadline d_i and a sequence of execution and suspension segments $S_i = [e_i^1, s_i^1, e_i^2, s_i^2, \ldots, e_i^n]$. These task parameters encode the constraints that any two jobs generated by T_i must be separated by a minimum inter-arrival time p_i. Each job released by T_i must finish its execution by a relative deadline d_i, and alternates between execution and suspension segments as defined by the sequence S_i. The execution time of the k-th execution segment of job j is upper-bounded by e_i^k, and the suspension time of its k-th suspension segment is upper-bounded by s_i^k.

Next, let us recall the definition of sustainable policy as proposed by Burns and Baruah [6].

▶ **Definition 2** (Sustainable Policy – original definition from [6]). A scheduling policy and/or a schedulability test for a scheduling policy is sustainable if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual tasks are changed in any, some, or all of the following ways: (i) decreased execution requirements; (ii) larger periods; (iii) smaller jitter; and (iv) larger relative deadlines.

As explained by Burns and Baruah [6], the interpretation of Definition 2 for scheduling policies concerns the values of job parameters at runtime: *"[...] a scheduling policy that guarantees to retain schedulability if actual execution requirements during run-time are smaller than specified WCET's, and if actual jitter is smaller than the specified maximum jitters, would be said to be sustainable with respect to WCET's and jitter".*

Thus, in order to show that a scheduling policy is not sustainable with respect to execution requirements (*i.e.*, job costs), we must find a counterexample that shows a job set $\mathcal{J}$ that is schedulable under that policy, along with a job set $\mathcal{J}_{better}$ with lower or equal job execution times that is not schedulable under the same policy.

Fig. 1 depicts such a counterexample for uniprocessor EDF scheduling on the segmented self-suspending task model, adapted from prior work by Abdeddaïm and Masson [1]. Fig. 1-(a) shows the original EDF schedule of three tasks T_1, T_2 and T_3, which contains no deadline misses. Next, by reducing the cost of T_3's job by 1 time unit as shown in Fig. 1-(b), the different interleaving of suspension times during the time interval [13, 16) increases the interference incurred by task T_1, causing a deadline miss at time 18.

This counterexample, which is simple enough to make the claim non-disputable, proves that, according to Definition 2, EDF scheduling under the segmented suspension model is *not sustainable* with respect to job costs.

1.2 Uniprocessor EDF Scheduling with Self-Suspensions is Sustainable w.r.t. Job Costs

Consider the same platform, task model and scheduling policy as in §1.1, and recall the definition of sustainable policy proposed by Baker and Baruah [2].

▶ **Definition 3** (Sustainable Policy – original definition from [2]). Let A denote a scheduling policy. Let τ denote any sporadic task system that is A-schedulable. Let $\mathcal{J}$ denote a collection of jobs generated by τ. Scheduling policy A is said to be sustainable if and only if A meets all deadlines when scheduling any collection of jobs obtained from $\mathcal{J}$ by changing the parameters of one or more individual jobs in any, some, or all of the following ways: (i) decreased execution requirements; (ii) larger relative deadlines; and (iii) later arrival times with the restriction that successive jobs of any task $T_i \in \tau$ arrive at least p_i time units apart.

Definition 3 is similar to Definition 2, except that it explicitly makes the difference between the notion of jobs and tasks. It requires the job set $\mathcal{J}$ with original parameters to be generated by a task set τ that is A-schedulable, *i.e.*, all job sets generated by τ exhibit no deadline misses when scheduled by A. However, note that the modified job set obtained from $\mathcal{J}$ (which we call $\mathcal{J}_{better}$) does not have to be generated by τ.

Now, we must check whether the counterexample in Fig. 1 is still valid. At a first glance, the job sets $\mathcal{J}$ and $\mathcal{J}_{better}$ depicted in Fig. 1-(a) and Fig. 1-(b) seem to prove that uniprocessor EDF scheduling with segmented self-suspending tasks is *not sustainable* with respect to job costs, according to Definition 3. After all, we can assume that job set $\mathcal{J}$ is generated for instance by some task set $\tau = \{(p_1 = 12, d_1 = 6, S_1 = [2,2,2]), (p_2 = 9, d_3 = 7, S_2 = [2,2,2]), (p_3 = 10, d_3 = 10, S_3 = [2])\}$.

However, let us consider the alternative job set $\mathcal{J}_{susp}$ in Fig. 1-(c), in which the job of task T_3 has the original cost of 2 time units, and the suspension time of the second job of task T_2 is reduced by 1 time unit. Clearly, $\mathcal{J}_{susp}$ can be generated by task set τ, since the job costs are the same as in $\mathcal{J}$ and the suspension segments are no larger than those in $\mathcal{J}$, which is allowed by the segmented suspension model. Moreover, we can observe that in the schedule of $\mathcal{J}_{susp}$, task T_1 again misses a deadline at time 18.

Since job set $\mathcal{J}_{susp}$ generated by τ is not schedulable, it is clear that τ does not satisfy the assumption of being A-schedulable (*i.e.*, EDF-schedulable) required by Definition 3. Therefore, job sets $\mathcal{J}$ and $\mathcal{J}_{better}$ in Fig. 1 are not a valid counterexample for establishing that the policy is not sustainable. Since the counterexample is not valid, what can we really say about the sustainability of this policy? Why do the two definitions disagree?

One aspect that is implicit but unclear in both definitions is whether all job parameters other than the sustainable parameter (*i.e.*, job costs) must remain constant. In fact, as shown in $\mathcal{J}_{susp}$ from Fig. 1-(c), in some cases we can vary the other parameters (*i.e.*, job suspension times) to compensate the increase in interference that would otherwise cause the scheduling anomaly. Since this parameter variation is allowed by the task constraints, this suggests that a task set that is schedulable for any possible job suspension times may in effect be resilient to scheduling anomalies on job costs, even though individual schedulable job sets are not.

In fact, by constructing job sets similar to $\mathcal{J}_{susp}$ in the example above, we provide a *mechanized* proof (*i.e.*, a proof that is verified by the Coq proof assistant) in §4 that establishes that uniprocessor job-level fixed priority (JLFP) scheduling of sporadic tasks under the dynamic suspension model is, what we later define as, *weakly sustainable* with respect to job costs and *variable suspension times.*

Note that this result does not make the counterexample of Abdeddaïm and Masson incorrect. Their result is simply based on a different interpretation of sustainability where nothing but the job parameter under consideration for the sustainability property can vary between the compared schedules; thus, the results stated in §1.1 and §4 are both correct. In §3, we will complement the existing sustainability theory with the notions of *strong* and *weak* sustainability to distinguish those contradictory but correct interpretations of sustainability.

1.3 This Paper

The seemingly contradictory observations in §1.1 and §1.2 suggest the need for clarification in the definitions of sustainability, which are currently restricted to the standard sporadic task model and are not precise with respect to how parameters can vary across the original and modified job sets $\mathcal{J}$ and $\mathcal{J}_{better}$.

We believe that the solution to this problem lies in formalizing the abstract concepts of *real-time scheduling meta-theory* such as "job and task parameters" in a rigorous way, so that the different notions of sustainability can be stated precisely. Additionally, this approach allows transcribing those concepts into a proof assistant such as Coq to formalize and mechanically prove key results [7]. With that in mind, we propose a formal sustainability theory for real-time scheduling, which we present in §2.

Our goal in this paper is not only to clarify what sustainability means, but also to provide a foundation for more efficient schedulability analyses for policies that are sustainable *with varying parameters* (such as the suspension times in the example from §1.2), a new concept that we call *weakly sustainable policy*. The exact definition and implications of weak sustainability will be discussed in §3.

Finally, we apply this newly defined notion of weak sustainability in §4, where we formalize self-suspending tasks in Coq and mechanically prove that uniprocessor, job-level fixed priority (JLFP) scheduling of self-suspending tasks under the dynamic suspension model is weakly sustainable with respect to job costs and varying suspension times.

To summarize, this paper makes the following contributions:

1. a formal theory of sustainability in real-time scheduling, with definitions of sustainable policy [2, 6], sustainable analysis [2, 5, 6] and self-sustainable analysis [2] generalized to *any scheduling policy and any task and platform models* (§2);
2. the definition of the new notions of strongly and weakly sustainable policies (§3), and the corresponding composition rules (§3.2);
3. the first formalization of sustainability theory and real-time scheduling with self-suspensions in a proof assistant (§4.1 and online appendix [15]); and
4. a mechanized proof of weak sustainability of uniprocessor JLFP scheduling of dynamic self-suspending tasks with respect to job costs and varying suspension times (§4.2–§4.4 and online appendix [15]).

2 Formalization of Sustainability Theory

In this section, we formalize the theory of sustainability in real-time scheduling and characterize the basic notions of sustainability proposed in the literature, namely *sustainable policy* [2, 6], *sustainable analysis* [5, 6] and *self-sustainable analysis* [2].

Our motivation for developing this theory is twofold: we aim to **(a)** clarify and generalize the existing notions of sustainability so that they become compatible with *any scheduling policy and any task and platform models*, and **(b)** provide the theoretical support for defining the new concept of *weak sustainability*, which will be covered in §3 and mechanically proven in §4 for uniprocessor JLFP scheduling of dynamic self-suspending tasks.

Note that this section does not introduce fundamentally new concepts; rather, it spells out precisely common implicit assumptions about the task and platform models and gives a more formal presentation of the underlying *real-time scheduling meta-theory*, which will be used to mechanically prove the results (see §4).

In order to distinguish the different nuances of sustainability, one must be able to correlate the variation of job and task parameters with schedulability. Hence, we must formalize the system model and present the basic definitions related to jobs and tasks.

2.1 Platform Model

We begin by stating assumptions about the platform model, in particular the notions of time and platform parameters, which specify part of the scheduling problem to be solved. Note that all definitions in this paper are compatible with both *discrete* and *dense* time.

▶ **Definition 4** (Processor Platform)**.** Let platform Π be the system on which jobs are scheduled.

▶ **Definition 5** (Platform Parameter). Each platform Π has a finite set of parameters $\mathcal{P}_{plat}$.

▶ **Example 6** (Common Platforms). Examples of platforms include uniprocessor systems, identical multiprocessors [9], and uniform multiprocessors [3]. Multiprocessor platforms usually have an associated parameter $m \in \mathcal{P}_{plat}$ that indicates the number of processors.

Note that Definition 5 does not limit the set of parameters defining a platform to its number of processors; in fact, the set of parameters $\mathcal{P}_{plat}$ could also express the heterogeneity of the platform [4], its power consumption, or execution speed profiles [17]. We keep the set of parameters unspecified in order to retain maximal generality and not limit our definitions to a fixed subset of system models.

This approach is uncommon. Most works tend to limit their results to a specific system model (e.g., task-level fixed priority scheduling of sequential tasks on single or multi-core processors). Instead, we prefer generality to specificity, so that the concepts and properties presented hereafter can be instantiated for any scheduling problem.

2.1.1 Jobs

After discussing the general aspects of the system model, we now define a job set.

▶ **Definition 7** (Job Set). A job set $\mathcal{J}$ is a (potentially infinite) collection of jobs.

Next, in order to define sustainability without being restricted to a particular task model, we generalize the notion of a job parameter.

▶ **Definition 8** (Job Parameter). We denote as job parameters any finite set $\mathcal{P}_{job}$, where each parameter $p \in \mathcal{P}_{job}$ is a function over jobs.

▶ **Example 9.** Common job parameters include $cost(j)$, the actual job execution time, $arrival(j)$, the absolute job arrival time, and $deadline(j)$, the relative job deadline. They may for instance also include the job suspension time in the case of self-suspending jobs, its level of parallelism and/or its energy consumption if such properties are of interest.

Next, we define the notion of scheduling policy, which specifies the strategy for selecting jobs to be scheduled, *i.e.*, allocated to a processor at a given time.

▶ **Definition 10** (Scheduling Policy). Given a platform Π and a job set $\mathcal{J}$ with job parameters $\mathcal{P}_{job}$, we define a scheduling policy σ as any algorithm that determines whether a job $j \in \mathcal{J}$ is scheduled at a given time t on a processor $\pi \in \Pi$.

For job sets that have associated deadlines, we can also define whether they are schedulable.

▶ **Definition 11** (Schedulable Job Set). Assume that jobs have a deadline as one of their parameters. Then we say that a job set $\mathcal{J}$ is schedulable on platform Π under policy σ iff none of its jobs misses a deadline when scheduled on Π under policy σ.

To compare different job sets, we must also be able to express how job parameters can vary across job sets (*e.g.*, job costs may increase while their arrival times remain constant). For that, we define whether two job sets differ only by a given set of parameters.

▶ **Definition 12** (Varying Job Parameters in V). Consider any subset of job parameters $V \subseteq \mathcal{P}_{job}$, which we call variable parameters, and consider two enumerated job sets $\mathcal{J} = \{j_1, j_2, \ldots\}$ and $\mathcal{J}' = \{j'_1, j'_2, \ldots\}$. We say that $\mathcal{J}$ and $\mathcal{J}'$ differ only by V iff $|\mathcal{J}| = |\mathcal{J}'|$ and $\forall i, \forall p \in (\mathcal{P}_{job} \setminus V),\ p(j_i) = p(j'_i)$, where $|\mathcal{J}|$ denotes the cardinality of job set $\mathcal{J}$.

▶ **Example 13.** By stating that $\{j_1, j_2\}$ and $\{j'_1, j'_2\}$ differ only by $V = \{cost\}$, we claim that jobs j_1 and j'_1 (respectively, j_2 and j'_2), are identical in all parameters other than *cost*. This is useful to formalize, for example, the idea that "schedulability is maintained when reducing *only* the cost of a job."

2.1.2 Tasks

While some notions of sustainability apply exclusively to job sets, one can also describe how the variation of task parameters affects schedulability analysis results. To be able to reason at the task level, we begin by defining task set and task parameters.

▶ **Definition 14** (Task Set)**.** A task set τ is as a finite collection of tasks $\{T_1, \ldots, T_n\}$.

From a mathematical point of view, tasks are opaque objects, elements of an enumerated set. Their utility comes from defining task parameters and using them to constrain the sets of jobs that can possibly be generated at runtime (see Definition 18 below).

▶ **Definition 15** (Task Parameters)**.** We call task parameters any finite set $\mathcal{P}_{task}$, where each parameter $p \in \mathcal{P}_{task}$ is a function over tasks.

▶ **Example 16.** Similar to the job parameters in Example 9, common task parameters include, but are not limited to, $WCET(T_i)$, the worst-case execution time of task T_i, and $period(T_i)$, the period or minimum inter-arrival time of task T_i.

Next, we define a task model, which determines how job sets are related to task sets.

▶ **Definition 17** (Task Model)**.** A task model $\mathcal{M}$ is the collection of all task sets that can be defined with given task parameters $\mathcal{P}_{task}$, along with a set of constraints relating job parameters with task parameters.

▶ **Definition 18** (Generated Job Sets)**.** Every task set $\tau \in \mathcal{M}$ generates a (potentially infinite) collection of job sets, denoted $jobsets(\tau) = \{\mathcal{J}_1, \mathcal{J}_2, \ldots\}$, with the condition that, for every job set $\mathcal{J} \in jobsets(\tau)$ and every job $j \in \mathcal{J}$, **(a)** j belongs to an associated task in τ, denoted $task(j)$, and **(b)** the job parameters of j are constrained by the task parameters of $task(j)$, as determined by $\mathcal{M}$.

One example of such a task model constraint is the upper bound on job execution times.

▶ **Example 19** (Constraint on Job Execution Time)**.** Let $\mathcal{M}$ be the sporadic task model. Let the job parameter $cost(j)$ denote the actual execution time of job j and let the task parameter $WCET(T_i)$ denote the WCET of task T_i. For every job set $\mathcal{J}$ generated by $\mathcal{M}$, the cost of each job $j \in \mathcal{J}$ is upper-bounded by the cost of its task, *i.e.*,

$$\forall \tau \in \mathcal{M}, \forall \mathcal{J} \in jobsets(\tau), \forall j \in \mathcal{J}, cost(j) \leq WCET(task(j)).$$

Using the notion of generated job sets, we can now define whether a task set is schedulable.

▶ **Definition 20** (Schedulable Task Set)**.** A task set $\tau \in \mathcal{M}$ is schedulable on platform Π under scheduling policy σ iff every generated job set $\mathcal{J} \in jobsets(\tau)$ is schedulable on Π under σ.

Similarly to Definition 12, in order to relate parameters across task sets, we define whether two task sets differ only by a given set of parameters.

▶ **Definition 21** (Varying Task Parameters in V)**.** Consider any subset of task parameters $V \subseteq \mathcal{P}_{task}$, which we call variable parameters, and consider two task sets $\tau = \{T_1, T_2, \ldots\}$ and $\tau' = \{T'_1, T'_2, \ldots\}$. We say that τ and τ' differ only by V iff $|\tau| = |\tau'|$ and $\forall\, i, \forall\, p \in (\mathcal{P}_{task} \setminus V),\ p(T_i) = p(T'_i)$, where $|\tau|$ denotes the cardinality of task set τ.

2.2 Generalized Sustainability Definitions

In this section, we use the basic concepts of jobs and tasks to formalize the notions of sustainability found in the literature, namely *sustainable policy* (§2.2.1), *sustainable analysis* (§2.2.2) and *self-sustainable analysis* (§2.2.3). Note that, differently from prior work [2, 5, 6], our definitions are generic and compatible with different task and platform models.

2.2.1 Sustainable Scheduling Policy

We begin by generalizing the concept of a *sustainable scheduling policy* [2, 6], which was briefly discussed in §1. The definition captures the idea that, if a policy is sustainable with respect to a set of job parameters, having "better" values for those parameters (*e.g.*, lower job execution costs, larger periods, less jitter, *etc.*) at runtime does not cause any deadline miss. We call this notion "strong sustainability," for reasons that will be made clear in §3.

▶ **Definition 22** (Strongly Sustainable Policy). Assume any scheduling policy σ and platform Π, and consider any subset of job parameters $S \subseteq \mathcal{P}_{job}$, which we call sustainable parameters. For each parameter $p \in S$, let $\preceq_p$ be any partial order over job sets, such that $\mathcal{J} \preceq_p \mathcal{J}'$ holds iff every job in $\mathcal{J}$ has no worse parameter p than its corresponding job in $\mathcal{J}'$. Then we say that the scheduling policy σ is strongly sustainable with respect to the job parameters in S iff

$$\begin{aligned}
&\forall\ \mathcal{J}\ \textbf{s.t.}\ \mathcal{J} \text{ is schedulable on platform } \Pi \text{ under policy } \sigma,\\
&\quad \forall\ \mathcal{J}_{better}\ \textbf{s.t.}\ \mathcal{J} \text{ and } \mathcal{J}_{better} \text{ differ only by } S \textbf{ and } \forall p \in S,\ \mathcal{J}_{better} \preceq_p \mathcal{J},\\
&\qquad \mathcal{J}_{better} \text{ is schedulable on platform } \Pi \text{ under policy } \sigma.
\end{aligned}$$

Definition 22 states that, under a strongly sustainable scheduling policy σ, whenever we compare two job sets and show that the job set with "worse parameters" does not miss any deadline, then the job set with "better parameters" must also not miss any deadline.

Note that the relation $\preceq_p$ is a crucial part of the specification and should be clearly indicated in the sustainability claim, as shown in the next examples.

▶ **Example 23** (Sustainability with Decreasing Job Costs). Let σ denote any uniprocessor work-conserving, fixed-priority scheduling policy and let $cost(j)$ denote the actual execution time of job j. Given any job sets $\mathcal{J} = \{j_1, j_2, \ldots\}$ and $\mathcal{J}' = \{j'_1, j'_2, \ldots\}$, we define the relation $\mathcal{J} \preceq_{cost} \mathcal{J}'$ as $\forall i, cost(j_i) \leq cost(j'_i)$.

Using the relation $\preceq_{cost}$, we can instantiate Definition 22. This property expresses the notion that, under policy σ, decreasing job execution times does not render the system unschedulable. This property was proven by Ha and Liu [11] for various job models.

Similarly, one can also define sustainability with respect to job inter-arrival times. It just requires a more nuanced partial order definition, as shown in the following example.

▶ **Example 24** (Sustainability with Increasing Job Inter-Arrival Times). Let σ denote any work-conserving, fixed-priority scheduling policy and let $arrival(j)$ denote the absolute arrival time of job j. Next, given any job sets $\mathcal{J} = \{j_1, j_2, \ldots\}$ and $\mathcal{J}' = \{j'_1, j'_2, \ldots\}$, we define the relation $\preceq_{interarrival}$ as

$$\begin{aligned}
&\forall i, \forall j_{prev}, \forall j'_{prev}\ \textbf{s.t.}\\
&\quad task(j_i) = task(j_{prev}) = task(j'_i) = task(j'_{prev}) \textbf{ and}\\
&\quad arrival(j_{prev}) < arrival(j_i) \textbf{ and } arrival(j'_{prev}) < arrival(j'_i),\\
&\qquad arrival(j_i) - arrival(j_{prev}) \geq arrival(j'_i) - arrival(j'_{prev}).
\end{aligned}$$

This relation expresses that, if $\mathcal{J} \preceq_{interarrival} \mathcal{J}'$, then the distance between two jobs of the same task in $\mathcal{J}$ is no worse (i.e., no smaller) than in $\mathcal{J}'$.

Finally, note that Definition 22 differs from Definition 3 (in §1.2) due to Baker and Baruah [2], as it does not require the original job set $\mathcal{J}$ to belong to some schedulable task set τ. Thus, according to our definition, Figs. 1-(a) and 1-(b) are a valid counterexample for establishing the non-sustainability (in the strong sense) of JLFP schedulers w.r.t. job costs in the presence of self-suspensions, which agrees with Definition 2 in §1.1.

2.2.2 Sustainable Schedulability Analysis

Having discussed how sustainability applies to scheduling policies, we now present the corresponding definitions for schedulability analyses, starting with the notion of *sustainable schedulability analysis* [2, 5, 6]. Before we proceed, we must define schedulability analysis.

▶ **Definition 25** (Schedulability Analysis). Let a schedulability analysis $\mathcal{A}$ for task model $\mathcal{M}$, platform Π, and scheduling policy σ denote any algorithm that assesses whether a task set $\tau \in \mathcal{M}$ is schedulable on Π under policy σ.

Now we state whether a given schedulability analysis $\mathcal{A}$ is sustainable. The intuition is that, if analysis $\mathcal{A}$ is sustainable with respect to certain job parameters, then, if a task set τ is deemed schedulable by $\mathcal{A}$, any job set with "better" parameters than those of a job set generated by τ does not miss any deadlines.

▶ **Definition 26** (Sustainable Analysis). Consider any schedulability analysis $\mathcal{A}$ for task model $\mathcal{M}$, platform Π, and scheduling policy σ, and consider any subset of job parameters $S \subseteq \mathcal{P}_{job}$, which we call sustainable parameters. For each parameter $p \in S$, let $\preceq_p$ be any partial order over job sets, such that $\mathcal{J} \preceq_p \mathcal{J}'$ holds iff every job in $\mathcal{J}$ has no worse parameter p than its corresponding job in $\mathcal{J}'$. Then we say that analysis $\mathcal{A}$ is sustainable with respect to S iff

$$\begin{aligned}
&\forall\ \tau\ \in \mathcal{M}\ \textbf{s.t.}\ \tau \text{ is deemed schedulable by } \mathcal{A},\\
&\quad \forall\ \mathcal{J} \in \mathit{jobsets}(\tau), \forall\ \mathcal{J}_{better}\ \textbf{s.t.}\\
&\qquad \mathcal{J} \text{ and } \mathcal{J}_{better} \text{ differ only by } S\ \textbf{and}\ \forall p \in S, \mathcal{J}_{better} \preceq_p \mathcal{J},\\
&\qquad\quad \mathcal{J}_{better} \text{ is schedulable on } \Pi \text{ under policy } \sigma.
\end{aligned}$$

Although the definitions of strongly sustainable policy (Definition 22) and sustainable analysis (Definition 26) both refer to the runtime behavior of the policy, the two notions are different. If the analyzed policy σ is strongly sustainable w.r.t. some parameters S, then any sufficient or exact schedulability analysis for σ is also sustainable w.r.t. S. However, even if σ is not strongly sustainable, it is possible to find sufficient schedulability analyses that are sustainable. In fact, we argue this is exactly the case that an intuitive notion of a "safe analysis" is trying to address: the underlying policy σ may exhibit various kinds of scheduling anomalies, but if a specific task set is deemed schedulable by a sustainable analysis, then no deadlines will be missed in the actual system even if some parameters turn out to be "better in the real system than assumed during analysis."

2.2.3 Self-Sustainable Analysis

Another type of sustainability that can be found in the literature, also related to schedulability analysis, is the notion of *self-sustainable analysis* [2]. The intuition is that, if analysis $\mathcal{A}$ is self-sustainable with respect to a set of task parameters, then, if a task set τ is deemed schedulable by analysis $\mathcal{A}$, every task set with "better" parameters than τ will also be deemed schedulable by $\mathcal{A}$.

▶ **Definition 27** (Self-Sustainable Analysis)**.** Let $\mathcal{A}$ be any schedulability analysis for task model $\mathcal{M}$, platform Π, and scheduling policy σ, and consider any subset of task parameters $S \subseteq \mathcal{P}_{task}$. For each parameter $p \in S$, let $\preceq_p$ be any partial order over task sets, such that $\tau \preceq_p \tau'$ holds iff every task in τ has no worse parameter p than its corresponding task in τ'. Then we say that schedulability analysis $\mathcal{A}$ is self-sustainable with respect to S iff

$$\begin{aligned}
&\forall\, \tau \in \mathcal{M} \textbf{ s.t. } \tau \text{ is deemed schedulable by } \mathcal{A},\\
&\qquad \forall\, \tau_{better} \textbf{ s.t. } \tau \text{ and } \tau_{better} \text{ differ only by } S \textbf{ and } \forall p \in S, \tau_{better} \preceq_p \tau,\\
&\qquad\quad \tau_{better} \text{ is deemed schedulable by } \mathcal{A}. \qquad (1)
\end{aligned}$$

To clarify the definition, we provide an example.

▶ **Example 28** (RTA is Self-Sustainable with respect to Decreasing Task Costs)**.** Let $\mathcal{A}$ be some response-time analysis (RTA) for the sporadic task model and let $WCET(T_i)$ denote the worst-case execution time of task T_i. Given any task sets $\tau = \{T_1, T_2, \ldots\}$ and $\tau' = \{T_1', T_2', \ldots\}$ with the same number of tasks, we define the relation $\tau \preceq_{WCET} \tau'$ as $\forall i,\ WCET(T_i) \leq WCET(T_i')$.

Based on the task parameter $WCET$ and the relation $\preceq_{WCET}$, we can instantiate the self-sustainability property as in Definition 27, which then expresses that, if the RTA claims τ to be schedulable, then it must also claim task sets with lower WCETs to be schedulable.

Note that, despite their similarity, the notions of sustainable and self-sustainable analysis are fundamentally different. While sustainability refers to job parameters, self-sustainability concerns task parameters. Moreover, to prove that an analysis $\mathcal{A}$ is sustainable, one must show that the job sets generated by a task set τ deemed schedulable by $\mathcal{A}$ do not have any anomalies. On the other hand, proving that analysis $\mathcal{A}$ is self-sustainable is a *purely algorithmic* property, akin to a notion of monotonicity, of the analysis procedure itself and has nothing to do with the safety at runtime of a system under analysis. For example, to prove the property in Example 28, one must show that, if the RTA computes a fixed point R for given task costs, then it will compute a fixed point $R' \leq R$ if lower task costs are provided.

3 Weakly Sustainable Scheduling Policies

Recall from §1.1 that uniprocessor EDF scheduling of self-suspending tasks was proven to be not sustainable with respect to job costs [1], and as mentioned at the end of §2.2.1, this result agrees with our notion of a strongly sustainable policy (Definition 22).

However, in §1.2, we also hinted (but did not prove) that this scheduling policy is still sustainable to some extent with respect to job costs. As shown in Fig. 1-(c), by reducing suspension times (*i.e.*, a transformation that is compliant with the task model and its constraints), we were able to construct a job set $\mathcal{J}_{susp} \in jobsets(\tau)$ that is as hard to schedule as job set $\mathcal{J}_{better}$. This suggests that any schedulability analysis $\mathcal{A}$ applied to task set τ would deem it "not schedulable" anyway because of job set $\mathcal{J}_{susp}$.

Thus, the fact that $\mathcal{J}_{better}$ itself is not schedulable does not straightforwardly prove that the uniprocessor EDF scheduling policy applied to self-suspending tasks is not sustainable in some sense w.r.t. job costs, at least if self-suspension times may vary at runtime. In fact, whether or not any parameters *other than the sustainable parameters* should be allowed to vary at runtime is the cause of most confusion in the various interpretations of sustainability found in the state of the art [2, 5, 6], and our motivation for formalizing the notion of varying job and task parameters in Definitions 12 and 21.

While the notion of strongly sustainable policy (Definition 22) expresses that the system remains schedulable if we decrease job costs *while maintaining all other parameters constant*, we believe that this is too strong an assumption in many settings, since most useful schedulability analyses will consider that job parameters can vary freely and concurrently across a range of possible values. The sustainability property that we are going to define thus allows *other parameters to vary*, subject to the constraints defined by the given task set. The rationale for this is that it allows for more efficient schedulability analyses if certain job parameters can be assumed to have maximal values while others are considered variable. The current sustainability theory does not allow such fine-grained categorization.

To develop a supporting theory for schedulability analyses based on this idea, in this section we propose a new classification of sustainable scheduling policies that differentiates between strong sustainability and weak sustainability.

3.1 Definition of Weakly Sustainable Policy

As suggested in the previous section, in order to define weak sustainability, we must be able to infer that a collection of job sets remains schedulable when certain parameters are allowed to vary. This idea is captured by the following definition.

▶ **Definition 29** (Schedulable with Varying Job Parameters V)**.** Given a task set τ and a subset of job parameters $V \subseteq \mathcal{P}_{job}$, we say that a job set $\mathcal{J}$ is schedulable with varying parameters V subject to task set τ on platform Π under policy σ iff any job set $\mathcal{J}_{other} \in jobsets(\tau)$ that differs from $\mathcal{J}$ only by V is also schedulable on Π under policy σ.

To illustrate the definition, we provide an example.

▶ **Example 30** (Schedulable with Varying Costs)**.** Assume any scheduling policy σ and consider the set of variable parameters $V = \{cost\}$. Given a job set $\mathcal{J} = \{j_1, j_2\}$ generated by task set τ, we say that $\mathcal{J}$ is schedulable with varying costs subject to task set τ iff every job set $\mathcal{J}_{other}$ generated by τ that has two jobs and the same parameters as $\mathcal{J}$ except for their costs is schedulable. That is, any job set constructed by changing only the job costs of $\mathcal{J}$ (to higher or lower values), without violating the constraints set forth by the parameters of task set τ, must be schedulable.

In other words, one may understand this notion to mean that job set $\mathcal{J}$ is not only schedulable itself, but also a "schedulability witness" for a whole family of related job sets that are identical in all parameters except for those in V. Based on this concept, we can now define precisely under which conditions a policy is weakly sustainable.

▶ **Definition 31** (Weakly Sustainable Policy)**.** Assume any platform Π, task model $\mathcal{M}$, and scheduling policy σ, and consider any disjoint subsets of job parameters $S \subseteq \mathcal{P}_{job}$ and $V \subseteq \mathcal{P}_{job}$, which we call sustainable and variable parameters, respectively. For each sustainable parameter $p \in S$, let $\preceq_p$ be any partial order over job sets, such that $\mathcal{J} \preceq_p \mathcal{J}'$ holds iff every job in $\mathcal{J}$ has no worse parameter p than its corresponding job in $\mathcal{J}'$. Then we say that scheduling policy σ is weakly sustainable with sustainable parameters S and variable parameters V iff

$$\forall\, \tau \in \mathcal{M}, \forall\, \mathcal{J} \in jobsets(\tau) \textbf{ s.t.}$$
$$\mathcal{J} \text{ is schedulable with varying } V \text{ subject to } \tau \text{ on platform } \Pi \text{ under policy } \sigma,$$
$$\forall\, \mathcal{J}_{better} \textbf{ s.t. } \mathcal{J} \text{ and } \mathcal{J}_{better} \text{ differ only by } S \textbf{ and } \forall\, p \in S, \mathcal{J}_{better} \preceq_p \mathcal{J},$$
$$\mathcal{J}_{better} \text{ is schedulable on platform } \Pi \text{ under policy } \sigma.$$

The idea of weak sustainability is that, if we can determine that a job set is schedulable for all variations of parameters in V (subject to the constraints imposed by its associated task set), then all job sets with better parameters S must be schedulable. For clarity, we provide the following example.

▶ **Example 32** (Weak Sustainability w.r.t. Job Costs and Varying Suspension Times). Consider a uniprocessor JLFP scheduling policy σ and the dynamic suspension model, *i.e.*, jobs can suspend at any time but the total suspension duration of each job is bounded by its task's maximum suspension time. Let $susp(j)$ denote the total suspension time of job j and $cost(j)$ the execution time of job j.

By defining the sets of job parameters $S = \{cost\}$ and $V = \{susp\}$, and the relation $\preceq_{cost}$ as in Example 23, one can instantiate Definition 31 and prove (as shown in §4) that, for any task set $\tau \in \mathcal{M}$, if job set $\mathcal{J}$ generated by τ is schedulable for all possible suspension times (subject to the upper limit imposed by τ), then all job sets with lower or equal job costs are also schedulable.

In the specific case where the set of varying parameters V is empty, we call the scheduling policy *strongly sustainable.*

▶ **Definition 33** (Strongly Sustainable Policy). We say that a policy is strongly sustainable with respect to the job parameters in S iff it is weakly sustainable with respect to the sustainable parameters in S and an empty set of variable parameters $V = \emptyset$.

Note that if $V = \emptyset$, proving that job set $\mathcal{J}$ is schedulable with varying parameters V is the same as establishing that $\mathcal{J}$ itself is schedulable. That implies the following equivalence, which connects the definitions of sustainable policy in §2 and §3.

▶ **Corollary 34** (Equivalence of Strong Sustainability). *The notion of strongly sustainable policy as defined in Definition 33 is equivalent to Definition 22.*

The weak sustainability property is useful for constraining the search space when developing schedulability analyses. As is already known, if some policy σ is strongly sustainable with respect to the parameters in S, maximizing/minimizing such parameters enables constructing worst-case scenarios (*e.g.*, the critical instant for uniprocessor FP scheduling of sporadic tasks [13]), so that only a single worst-case scenario must be analyzed (rather than the entire space of all possible parameter combinations).

However, recall that policy σ might not be strongly sustainable with respect to S. But if we are still able to prove that σ is weakly sustainable with respect to S and variable parameters V, we can still maximize/minimize the parameters in S, as long as the schedulability analysis covers all values of the parameters in V. In other words, establishing a weak sustainability property can be thought of as a dimensionality reduction of the search space that must be considered by a safe schedulability analysis.

For instance, having proven in Theorem 54 in §4.4 that uniprocessor JLFP scheduling of self-suspending tasks is weakly sustainable with respect to job costs and variable suspension times, we know that any schedulability analysis for that model may assume that all jobs generated by the tasks execute for their maximum execution time, and must search only for the worst-case assignments of job suspension times.

3.2 Composing Weak and Strong Sustainability Results

Although the definition of strong sustainability refers to a set S of multiple parameters, one can still establish the sustainability of each parameter in isolation. In fact, the critical instant for the sporadic task model is obtained by composing worst-case assumptions about individual job parameters: maximizing job costs, minimizing inter-arrival time, etc.

As will be shown in Theorem 38, this composition rule applies not only for strong sustainability (as discussed in prior work [2]), but can also be extended to weak sustainability. Before presenting the theorem, we first provide an alternative (but equivalent) definition of weak sustainability based on the contrapositive of Definition 31, which simplifies the proof of Theorem 38 below.

▶ **Definition 35** (Weakly Sustainable Policy – alternative definition). Assume any platform Π, task model $\mathcal{M}$ and scheduling policy σ, and consider any disjoint subsets of job parameters $S \subseteq \mathcal{P}_{job}$ and $V \subseteq \mathcal{P}_{job}$, which we call sustainable and variable parameters, respectively. For each sustainable parameter $p \in S$, let $\preceq_p$ be any partial order over job sets, such that $\mathcal{J} \preceq_p \mathcal{J}'$ holds iff every job in $\mathcal{J}$ has no worse parameter p than its corresponding job in $\mathcal{J}'$. Then we say that the scheduling policy is weakly sustainable with sustainable parameters S and variable parameters V iff

$$\begin{aligned}
&\forall\ \mathcal{J}\ \textbf{s.t.}\ \mathcal{J} \text{ is } \textit{not} \text{ schedulable on platform } \Pi \text{ under policy } \sigma,\\
&\quad \forall\ \tau \in \mathcal{M},\ \forall \mathcal{J}_{worse} \in \textit{jobsets}(\tau)\ \textbf{s.t.}\\
&\qquad \mathcal{J} \text{ and } \mathcal{J}_{worse} \text{ differ only by } S \textbf{ and } \forall p \in S, \mathcal{J} \preceq_p \mathcal{J}_{worse},\\
&\qquad\quad \exists\ \mathcal{J}'_{worse} \in \textit{jobsets}(\tau)\ \textbf{s.t.}\\
&\qquad\qquad \mathcal{J}_{worse} \text{ and } \mathcal{J}'_{worse} \text{ differ only by } V \textbf{ and}\\
&\qquad\qquad \mathcal{J}'_{worse} \text{ is not schedulable on platform } \Pi \text{ under policy } \sigma.
\end{aligned}$$

Put differently, for any job set $\mathcal{J}$ that is not schedulable, if we can find another job set $\mathcal{J}_{worse}$ that is generated by some task set τ and $\mathcal{J}$ is "better" than $\mathcal{J}_{worse}$, then there exists a member in $\mathcal{J}_{worse}$'s "family" of related job sets that is also not schedulable.

For instance, recall that this is the same reasoning underlying the counterexample in §1.2: given a job set $\mathcal{J}$ that is not schedulable (Fig. 1-(b)) and a job set $\mathcal{J}_{worse}$ with higher job costs (Fig. 1-(a)), we were able to show that there exists a job set $\mathcal{J}'_{worse}$ that only differs from $\mathcal{J}_{worse}$ in its suspension times and that also misses a deadline (Fig. 1-(c)).

In addition, we must introduce the notion of independent sets of job parameters.

▶ **Definition 36** (Independent Sets of Job Parameters). We say that subsets of job parameters $A \subset \mathcal{P}_{job}$ and $B \subset \mathcal{P}_{job}$ are independent with respect to task model $\mathcal{M}$ iff for each task parameter p_{task} defined by $\mathcal{M}$, and for every $p_A \in A$ and $p_B \in B$, if p_A is constrained by p_{task} according to model $\mathcal{M}$, then p_B is not constrained by p_{task} according to model $\mathcal{M}$.

In most task models commonly considered in the real-time literature, job parameters are usually independent of each other.

▶ **Example 37** (Parameters Are Usually Independent). In the sporadic task model with self-suspending tasks, the sets of job parameters $A = \{\textit{cost}, \textit{arrival}\}$ and $B = \{\textit{susp}\}$ have independent task constraints, since these job parameters are each constrained by a different task parameter, namely, the task WCET, minimum inter-arrival time and maximum suspension time. In contrast, in a hypothetical task model where every job j is split into two execution sections of length $\textit{cost}_1(j)$ and $\textit{cost}_2(j)$ such that $\textit{cost}_1(j) + \textit{cost}_2(j) \leq \textit{WCET}(\textit{task}(j))$, the parameters $\{\textit{cost}_1\}$ and $\{\textit{cost}_2\}$ are clearly not independent.

Using the definition of weak sustainability above (Definition 35) and the notion of independent sets of job parameters (Definition 36), we establish the composition rule for weakly sustainable policies.

▶ **Theorem 38** (Composition Rule: Weak – Weak). *Consider any task model $\mathcal{M}$, scheduling policy σ and processor platform Π. Let S_a, V_a, S_b, V_b denote subsets of the job parameters $\mathcal{P}_{job}$ such that $S_a \cap V_b = \emptyset$ and $S_b \cap V_a = \emptyset$, and such that either S_b is independent of $\mathcal{P}_{job} \setminus S_b$, or S_a is independent of $\mathcal{P}_{job} \setminus S_a$, with respect to task model $\mathcal{M}$. Assume that* ***(a)*** *σ is weakly sustainable with respect to S_a and variable parameters V_a, and that* ***(b)*** *σ is weakly sustainable with respect to S_b and variable parameters V_b. Then* ***(c)*** *σ is weakly sustainable with respect to $S_a \cup S_b$ and variable parameters $V_a \cup V_b$.*

Proof. Consider a job set $\mathcal{J}$ that is not schedulable on platform Π under policy σ. Let τ be any task set, and let $\mathcal{J}_{worse} \in jobsets(\tau)$ be a job set that only differs from $\mathcal{J}$ by the parameters in $S_a \cup S_b$ and that has no better parameters than $\mathcal{J}$ w.r.t. $S_a \cup S_b$. Then, according to Definition 35, we must prove that there exists a job set $\mathcal{J}'_{worse} \in jobsets(\tau)$ that only differs from $\mathcal{J}_{worse}$ with respect to $V_a \cup V_b$ and that is also not schedulable.

Using the independent parameters assumption, assume without loss of generality that it is S_b that is independent of all other job parameters $\mathcal{P}_{job} \setminus S_b$, with respect to model $\mathcal{M}$. If this is not the case, then by assumption we have that S_a is independent of other parameters $\mathcal{P}_{job} \setminus S_a$ and we can exchange the indices a and b in the remainder of the proof.

1. **Step 1 – Construction of $\mathcal{J}'_a$ from $\mathcal{J}$.** Let $\mathcal{J}_a$ be the same job set as $\mathcal{J}$, but with the same job parameters in S_a as $\mathcal{J}_{worse}$. That is, let $\mathcal{J} = \{j_1, j_2, \ldots\}$ and $\mathcal{J}_{worse} = \{j_1^w, j_2^w, \ldots\}$ and recall that they have the same number of jobs. Then we define $\mathcal{J}_a = \{j_1^a, j_2^a, \ldots\}$ with the same cardinality such that, for any index i, we have $\forall p \in S_a, p(j_i^a) = p(j_i^w)$ and $\forall p \notin S_a, p(j_i^a) = p(j_i)$.
   Next, we construct a task set $\tau_a \in \mathcal{M}$ such that, for every task parameter p_{task} that constrains job parameters in $\mathcal{P}_{job} \setminus S_b$, the value of p_{task} in τ_a is the same as in τ, and for every task parameter p_{task} that constrains job parameters in S_b, the value of p_{task} in τ_a is the same to the task set that generated job set $\mathcal{J}$. Since $\mathcal{J}_a$ only differs from $\mathcal{J}_{worse} \in jobsets(\tau)$ with respect to S_b, and S_b is independent of the other job parameters, it follows that $\mathcal{J}_a \in jobsets(\tau_a)$.
   Since $\mathcal{J}$ is not schedulable, and $\mathcal{J}$ and $\mathcal{J}_a$ differ only by S_a, we can exploit the fact that σ is weakly sustainable with S_a and varying V_a. Thus, it follows that there exists a job set $\mathcal{J}'_a \in jobsets(\tau_a)$ that differs from $\mathcal{J}_a$ only by the parameters in V_a and that is not schedulable on platform Π under policy σ.
2. **Step 2 – Construction of $\mathcal{J}'_{ab}$ from $\mathcal{J}'_a$.** Let $\mathcal{J}_{ab}$ be the same job set as $\mathcal{J}'_a$ except that the job parameters in S_b are the same as in $\mathcal{J}_{worse}$. That is, let $\mathcal{J}'_a = \{j_1^{a'}, j_2^{a'}, \ldots\}$ and $\mathcal{J}_{worse} = \{j_1^w, j_2^w, \ldots\}$ and recall that they have the same number of jobs. Then we define $\mathcal{J}_{ab} = \{j_1^{ab}, j_2^{ab}, \ldots\}$ with same cardinality such that, for any index i, we have $\forall p \in S_b, p(j_i^{ab}) = p(j_i^w)$ and $\forall p \notin S_b, p(j_i^{ab}) = p(j_i^{a'})$.
   Note that by construction, $\mathcal{J}_{ab}$ has the same job parameters as $\mathcal{J}_{worse} \in jobsets(\tau)$, except for those in V_a, which were obtained when generating $\mathcal{J}'_a$ via weak sustainability. However, note that $\mathcal{J}'_a$ is generated by task set τ_a, which has the same constraints for V_a as τ, since $V_a \cap S_b = \emptyset$. Thus, every job parameter of $\mathcal{J}_{ab}$ is compatible with τ, *i.e.*, $\mathcal{J}_{ab} \in jobsets(\tau)$.
   Since $\mathcal{J}'_a$ is not schedulable, and $\mathcal{J}'_a$ and $\mathcal{J}_{ab}$ differ only by S_b, we can exploit the fact that σ is weakly sustainable with S_b and varying V_b. Thus, there must exist a job set $\mathcal{J}'_{ab} \in jobsets(\tau)$ that differs from $\mathcal{J}_{ab}$ only by the parameters in V_b and that is not schedulable on platform Π under policy σ.

Since $\mathcal{J}$ has the same parameters as $\mathcal{J}_{worse}$ except for those in $S_a \cup S_b$, and because $\mathcal{J}_a$ and $\mathcal{J}_{ab}$ were constructed from $\mathcal{J}$ by copying the parameters S_a and S_b from $\mathcal{J}_{worse}$ and varying the parameters in $V_a \cup V_b$, it follows that $\mathcal{J}'_{ab}$ has the same parameters as $\mathcal{J}_{worse}$, except for the variable parameters V_a and V_b. Moreover, since $S_a \cap V_b = \emptyset$ and $S_b \cap V_a = \emptyset$, this guarantees that S_a and S_b do not vary during the construction of $\mathcal{J}'_a$ and $\mathcal{J}'_{ab}$, so for every $p \in S_a \cup S_b$, the order $\preceq_p$ is preserved across the successive job set transformations.

Thus, there exists a job set $\mathcal{J}'_{worse} = \mathcal{J}'_{ab}$ that belongs to *jobsets*(τ), that only differs from $\mathcal{J}_{worse}$ with respect to $V_a \cup V_b$ and is also not schedulable on platform Π under policy σ. ◀

Assuming $V_b = \emptyset$ yields a rule for combining strong and weak sustainability results.

▶ **Corollary 39** (Composition Rule: Weak – Strong). *Consider any scheduling policy σ and processor platform Π. Let S_a, V_a and S_b denote subsets of the job parameters $\mathcal{P}_{job}$ such that $S_b \cap V_a = \emptyset$, and such that either S_b is independent of $\mathcal{P}_{job} \setminus S_b$, or S_a is independent of $\mathcal{P}_{job} \setminus S_a$, with respect to task model $\mathcal{M}$. Assume that σ is weakly sustainable with respect to S_a and variable V_a and also strongly sustainable with respect to S_b. Then σ is weakly sustainable with respect to $S_a \cup S_b$ and variable V_a.*

Finally, assuming $V_a = V_b = \emptyset$ yields the composition rule for strong sustainability, which was already proven by Baker and Baruah [2].

▶ **Corollary 40** (Composition Rule: Strong – Strong). *Consider any scheduling policy σ and processor platform Π. Let S_a and S_b denote subsets of the job parameters $\mathcal{P}_{job}$, and such that either S_b is independent of $\mathcal{P}_{job} \setminus S_b$, or S_a is independent of $\mathcal{P}_{job} \setminus S_a$, with respect to task model $\mathcal{M}$. Assume that σ is strongly sustainable with respect to S_a and also strongly sustainable with respect to S_b. Then σ is strongly sustainable with respect to $S_a \cup S_b$.*

Note that, although necessary in the general case, the parameter independence constraint in Corollary 40 was not explicitly stated in the original definition [2], since Baker and Baruah only considered the sporadic task model, in which all parameters are independent.

4 Uniprocessor Scheduling of Dynamic Self-Suspending Tasks is Weakly Sustainable w.r.t. Job Costs and Variable Suspensions

In this section, we prove that uniprocessor JLFP scheduling with dynamic self-suspending tasks is weakly sustainable with respect to job costs and variable suspension times. Although we could have focused on other real-time task models, we chose to study the sustainability of self-suspending tasks for the following reasons.

1. **Recent errors.** This topic has faced many misconceptions in the past, with a considerable number of unsound results being published [8]. We hope that our work on sustainability introduces helpful formalism and a better understanding of the task model.
2. **Future work on schedulability analysis.** Proving weak sustainability of uniprocessor JLFP scheduling of dynamic self-suspending tasks can provide directions for future work. It enables more efficient schedulability analyses to be developed, by reducing the search space to only the parameters that must be kept variable (i.e., suspension times), while the others (i.e., execution times) can remain constant.

To address the issue of recent errors and increase the degree of confidence in the results, our proof has been mechanized in Prosa [7], a library for the Coq proof assistant for formal specification and machine-checked proofs of real-time scheduling theory. The specification and proofs are available online [15] and can be checked independently with the CoqChk tool. Simple step-by-step instructions are provided on the website.

Note that, despite being phrased in terms of sporadic tasks for the sake of simplicity, this proof is conceptually also compatible with other job arrival models (periodic, bursty, *etc.*).

The rest of this section is structured as follows. First, we present our formalization of the dynamic suspension model, which is required for stating the theorems in Prosa. Next, we provide an overview of our proof strategy based on schedule reductions, which can be reused in other sustainability proofs. In the remaining subsections, we discuss the high-level steps of the proof, which despite being specific for scheduling with self-suspensions, highlight key steps necessary in a rigorous proof of sustainability.

4.1 A Generic Suspension Model

In order to instantiate the sustainability claim for real-time scheduling of self-suspending tasks, we must formally define the concept of self-suspension.

▶ **Definition 41** (Job Suspension Time). We define job suspension time as a function $susp(j, s)$ such that, for any job j and any value $s \in \mathbb{N}$, $susp(j, s)$ expresses the duration for which j must suspend immediately after receiving s units of service.

The job suspension parameter is explained more clearly in the following example.

▶ **Example 42** (Table of Suspension Durations). Job suspension times $susp(j, s)$ can be understood as a table containing the duration of the suspension intervals associated with job j. For example, for a job j such that $cost(j) = 5$, we can define $susp(j, s)$ to equal 0 except for $susp(j, 3) = 2$ and $susp(j, 4) = 3$.

This suspension table indicates that job j executes for 3 time units, then suspends for 2 time units, then executes for 1 more time unit, then suspends for 3 more time units and finally completes its last time unit of execution. Note that this assignment is both an instance of the dynamic suspension model (with total suspension time equal to 5) and of the segmented suspension model (with execution segments $[e^1 = 3, s^1 = 2, e^2 = 1, s^2 = 3, e^3 = 1]$).

By allowing arbitrary suspension durations between each unit of service, this model is generic enough to represent any suspension pattern under discrete time. Thus, it supports both segmented [16] and dynamic [12] suspension models, as shown in Example 42.

Next, by accumulating suspension durations, we define the total suspension time of a job.

▶ **Definition 43** (Total Suspension Time). We define the total suspension time $susp_\Sigma(j)$ of job j as the cumulative suspension time up to completion, *i.e.*, $susp_\Sigma(j) = \sum_{s<cost(j)} susp(j, s)$.

After clarifying job suspension times, we now define task suspension times and show how both are related under the dynamic suspension model.

▶ **Definition 44** (Task Suspension Time). For any task T_i, we define the task suspension time $susp(T_i)$ as an upper-bound on the total suspension time of any job of T_i.

▶ **Definition 45** (Suspension Time Constraints). The dynamic suspension model requires that the total suspension time of any job is upper-bounded by the suspension time of its task, *i.e.*,

$$\forall\ \tau \in \mathcal{M}, \forall\ \mathcal{J} \in jobsets(\tau), \forall\ j \in \mathcal{J},\ \ susp_\Sigma(j) \leq susp(task(j)).$$

Beside its suspension time, every task T_i is defined by a WCET, a minimum inter-arrival time or period, and a deadline, as stated in Definition 1.

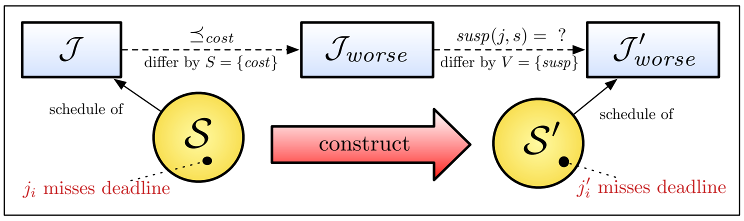


Figure 2 Proof strategy for establishing weak sustainability with respect to job costs and variable suspension times. Given a job j_i that misses a deadline in schedule $\mathcal{S}$ of the original job set $\mathcal{J}$, we construct a new job set $\mathcal{J}'_{worse}$ and a new schedule $\mathcal{S}'$ where the corresponding job j'_i misses a deadline. Note that in schedule $\mathcal{S}'$, job costs are no smaller than in $\mathcal{S}$, suspension times can be defined arbitrarily (within the bounds of the task set), and all other job parameters (i.e., arrival time, deadline) remain unchanged.

4.2 Overview of the Proof Strategy

Having presented the main characteristics of the dynamic self-suspending task model, we now explain our proof strategy for establishing weak sustainability of uniprocessor JLFP scheduling of dynamic self-suspending tasks w.r.t. job costs and variable suspension times. For simplicity, the proof is based on the alternative definition of weakly sustainable policy (Definition 35). According to Definition 35, we must prove that

$$\begin{aligned}
&\forall\ \mathcal{J}\ \textbf{s.t.}\ \mathcal{J}\text{ is \textit{not} schedulable under a uniprocessor JLFP scheduling policy } \sigma,\\
&\quad \forall\ \tau \in \mathcal{M},\ \forall \mathcal{J}_{worse} \in \textit{jobsets}(\tau)\ \textbf{s.t.}\\
&\qquad \mathcal{J}\text{ and }\mathcal{J}_{worse}\text{ differ only by } S = \{cost\}\ \textbf{and}\ \mathcal{J} \preceq_{cost} \mathcal{J}_{worse},\\
&\qquad\quad \exists\ \mathcal{J}'_{worse} \in \textit{jobsets}(\tau)\ \textbf{s.t.}\\
&\qquad\qquad \mathcal{J}_{worse}\text{ and }\mathcal{J}'_{worse}\text{ differ only by } V = \{susp\}\ \textbf{and}\\
&\qquad\qquad \mathcal{J}'_{worse}\text{ is not schedulable under policy } \sigma.
\end{aligned}$$

That is, first we consider any job set $\mathcal{J}$ that is not schedulable and any job set $\mathcal{J}_{worse}$ that has "no better *job costs*" than $\mathcal{J}$ (and that is otherwise identical). Then we must show that there exists a job set $\mathcal{J}'_{worse}$ generated by the same task set as $\mathcal{J}_{worse}$ that differs from $\mathcal{J}_{worse}$ only by its job *suspension times* and that it is not schedulable. In particular $\mathcal{J}$ and $\mathcal{J}_{worse}$ have equal suspension times (but not necessarily equal execution costs), whereas $\mathcal{J}_{worse}$ and $\mathcal{J}'_{worse}$ have equal execution costs (but not necessarily equal suspension times).

Our proof begins by considering any job set $\mathcal{J}$ and its associated schedule $\mathcal{S}$ where some job misses a deadline. Then we construct a job set $\mathcal{J}'_{worse}$ together with its schedule $\mathcal{S}'$ where some job also misses a deadline. This strategy is illustrated in Fig. 2.

In the next section, we present an algorithm for iteratively constructing schedule $\mathcal{S}'$ (and hence the associated job set $\mathcal{J}'_{worse}$) based on $\mathcal{S}$. It is followed by the two main proof obligations: **(a)** proving that some job misses a deadline in $\mathcal{S}'$ (§4.3.1) and **(b)** proving that $\mathcal{S}'$ is a valid schedule of $\mathcal{J}'_{worse}$ (§4.3.2). This proves that $\mathcal{J}'_{worse}$ is not schedulable.

4.3 Constructing $\mathcal{J}'_{worse}$ and Schedule $\mathcal{S}'$

Based on the strategy proposed in §4.2, we now present the algorithm to construct schedule $\mathcal{S}'$ and the associated job set $\mathcal{J}'_{worse}$, based on the original schedule $\mathcal{S}$. In the remainder of this paper, whenever we want to refer to the same job before and after the parameter transformation (*i.e.*, from $\mathcal{J}$ to $\mathcal{J}'_{worse}$), we refer to them as corresponding jobs j_i and j'_i.

Before proceeding, we must first introduce the concept of job service.

▶ **Definition 46** (Job Service). Given a schedule $\mathcal{S}$, we define $service(j, t)$ as the cumulative amount of time during which job j executes in the interval $[0, t)$.

Next, recall that jobs in $\mathcal{J}'_{worse}$ have no better job costs than in $\mathcal{J}$, *i.e.*, for any corresponding jobs j_i and j'_i, $cost(j_i) \leq cost(j'_i)$. Since they might have to execute for different durations, we begin by defining the notion of added cost.

▶ **Definition 47** (Added Cost). We define the added cost $\Delta_{cost}(j'_i)$ of job j'_i in $\mathcal{J}'_{worse}$ as the difference between its original and inflated costs, *i.e.*, $\Delta_{cost}(j'_i) = cost(j'_i) - cost(j_i) \geq 0$.

In order to guarantee that schedule $\mathcal{S}'$ becomes as hard as schedule $\mathcal{S}$ (*i.e.*, so that jobs still miss their deadlines) and at the same time easy to compare in terms of received job service (*i.e.*, the time for which a job executed since its release), we construct $\mathcal{S}'$ based on the idea of "picking jobs that are late with respect to $\mathcal{S}$," where late is defined as follows.

▶ **Definition 48** (Late job). We say that job j'_i is late in schedule S' at time t iff the service received by j'_i in S' up to time t is less than the service received by the corresponding job j_i in schedule $\mathcal{S}$ (compensated by the added cost), *i.e.*, $service(j'_i, t) < service(j_i, t) + \Delta_{cost}(j'_i)$.

We now present the algorithm used to iteratively build schedule $\mathcal{S}'$ and job set J'_{worse}. Algorithm 49 ensures that (i) every job $j'_i \in \mathcal{J}'_{worse}$ executes for its total execution cost $cost(j'_i)$ ($\geq cost(j_i)$), (ii) every job $j'_i \in \mathcal{J}'_{worse}$ has a total suspension time $susp(j'_i)$ upper-bounded by the suspension time $susp(j_i)$ of its corresponding job in schedule $\mathcal{S}$, and (iii) at least one job of $\mathcal{J}'_{worse}$ misses its deadline in $\mathcal{S}'$ (as proven in Sec 4.3.1).

▶ **Algorithm 49** (Construction of Job Set $\mathcal{J}'_{worse}$ and Schedule $\mathcal{S}'$). *Consider any time t and let $J(t)$ denote the set that contains every job j'_i that is ready (i.e., released, not completed, and not suspended) in schedule $\mathcal{S}'$ at time t and such that either **(a)** j'_i is late at time t or **(b)** the corresponding job j_i is scheduled in $\mathcal{S}$ at time t.*

1. ***Schedule:** We schedule in S' at time t the highest-priority job in $J(t)$, or idle the processor if $J(t)$ is empty.*
2. ***Suspensions:** Any job $j'_i \in \mathcal{J}'_{worse}$ suspends in $\mathcal{S}'$ at time t iff the corresponding job j_i is suspended in $\mathcal{S}$ and j'_i is not late.*

Note that Algorithm 49 not only picks late jobs, but also favors higher-priority jobs and tries to copy schedule $\mathcal{S}$ if possible. While rule (a) ensures that the schedule respects the JLFP policy, rule (b) provides a tie-breaking rule if there are multiple jobs that can be picked, in which case we choose the same job as the job scheduled in $\mathcal{S}$.

It only remains to be shown that schedule $\mathcal{S}'$ results in a deadline miss (Theorem 52) and schedule $\mathcal{S}'$ does not violate any property of the scheduling policy, platform, and task model, such as work conservation, priority enforcement, *etc.* (Theorem 53).

4.3.1 Proving that $\mathcal{S}'$ Misses a Deadline

In order to prove that some job $j'_i \in \mathcal{J}'_{worse}$ misses a deadline in $\mathcal{S}'$, we establish the following key invariant that relates the service in the two schedules $\mathcal{S}$ and $\mathcal{S}'$.

▶ **Lemma 50** (Service Invariant). *For any corresponding jobs $j_i \in \mathcal{J}$ and $j'_i \in \mathcal{J}'_{worse}$, at any time t, $service(j'_i, t) \leq service(j_i, t) + \Delta_{cost}(j'_i)$.*

Proof. Proven in Prosa [15]. Consider any pair of corresponding jobs j_i and j'_i. The proof follows by induction on time t.

1. **Base Case:** At time $t = 0$, jobs have received no service, thus $\mathit{service}(j'_i, 0) = 0 = \mathit{service}(j_i, t) \leq \mathit{service}(j_i, t) + \Delta_{cost}(j'_i)$.
2. **Inductive Step:** Assume as the induction hypothesis that, for some t, $\mathit{service}(j'_i, t) \leq \mathit{service}(j_i, t) + \Delta_{cost}(j'_i)$. Then we must prove $\mathit{service}(j'_i, t+1) \leq \mathit{service}(j_i, t+1) + \Delta_{cost}(j'_i)$. First, consider the simple case where job j'_i is not scheduled in $\mathcal{S}'$ at time t. Then,

$$\begin{aligned}
\mathit{service}(j'_i, t+1) &= \mathit{service}(j'_i, t) && (j'_i \text{ is not scheduled in } \mathcal{S}' \text{ at } t)\\
&\leq \mathit{service}(j_i, t) + \Delta_{cost}(j'_i) && \text{(by induction hypothesis)}\\
&\leq \mathit{service}(j_i, t+1) + \Delta_{cost}(j'_i). && \text{(by monotonicity of service)}
\end{aligned}$$

Otherwise, assume that j'_i is scheduled in $\mathcal{S}'$ at time t. From the schedule construction (Algorithm 49), it follows that either (a) $\mathcal{S}$ and $\mathcal{S}'$ schedule *corresponding jobs* at time t, or (b) $\mathcal{S}'$ schedules a late job at time t. We analyze both cases.

a. **Corresponding Jobs are Scheduled:** The corresponding jobs scheduled in $\mathcal{S}$ and $\mathcal{S}'$ at time t must be j_i and j'_i, so

$$\begin{aligned}
\mathit{service}(j'_i, t+1) &= \mathit{service}(j'_i, t) + 1 && (j'_i \text{ is scheduled in } \mathcal{S}' \text{ at time } t)\\
&\leq \mathit{service}(j_i, t) + \Delta_{cost}(j'_i) + 1 && \text{(by induction hypothesis)}\\
&= \mathit{service}(j_i, t+1) + \Delta_{cost}(j'_i) && (j_i \text{ is scheduled in } \mathcal{S} \text{ at time } t).
\end{aligned}$$

b. **Late Job:** Job j'_i must be the highest-priority late job in $\mathcal{S}'$ at time t. By the definition of late job (Definition 48), it follows that $\mathit{service}(j'_i, t) < \mathit{service}(j_i, t) + \Delta_{cost}(j'_i)$, so

$$\begin{aligned}
\mathit{service}(j'_i, t+1) &= \mathit{service}(j'_i, t) + 1 && (j'_i \text{ is scheduled in } \mathcal{S}' \text{ at time } t)\\
&< \mathit{service}(j_i, t) + \Delta_{cost}(j'_i) + 1 && \text{(by assumption)}\\
&\leq \mathit{service}(j_i, t) + \Delta_{cost}(j'_i). && \text{(by converting } < \text{ to } \leq)
\end{aligned}$$

The claim holds in all cases, which concludes the proof by induction. ◀

Since we must prove that schedule $\mathcal{S}'$ results in a deadline miss, we use the service invariant above to conclude that jobs complete earlier in $\mathcal{S}$ than in $\mathcal{S}'$.

▶ **Corollary 51** (Jobs Complete Earlier in $\mathcal{S}$). *For any corresponding jobs $j_i \in \mathcal{J}$ and $j'_i \in \mathcal{J}'_{worse}$, if j'_i has completed in schedule $\mathcal{S}'$ by time t, then j_i has completed in $\mathcal{S}$ by time t.*

Proof. Proven in Prosa [15]. Follows from Lemma 50, since j_i receives enough service in $\mathcal{S}$ to complete before the corresponding j'_i in $\mathcal{S}'$. ◀

Recall that we initially assumed that some job misses a deadline in $\mathcal{S}$. We can thus conclude that the corresponding job also misses a deadline in $\mathcal{S}'$.

▶ **Theorem 52** (Deadline Miss). *There exists a job $j'_i \in \mathcal{J}'_{worse}$ that misses a deadline in $\mathcal{S}'$.*

Proof. Proven in Prosa [15]. Recall from Corollary 51 that corresponding jobs complete earlier in $\mathcal{S}$ than in $\mathcal{S}'$. Since by assumption there exists a job j_i that misses a deadline in $\mathcal{S}$, the corresponding job j'_i must also miss a deadline in $\mathcal{S}'$. ◀

4.3.2 Proving that $\mathcal{S}'$ is a Valid Schedule

Although we have already established the non-schedulability of the generated schedule $\mathcal{S}'$, it remains to be shown that schedule $\mathcal{S}'$ is valid and compatible with the task model.

▶ **Theorem 53** (Valid Schedule). *Schedule $\mathcal{S}'$ is a valid uniprocessor schedule of job set $\mathcal{J}'_{worse}$ assuming JLFP scheduling of sporadic, dynamic self-suspending tasks.*

Proof. Proven in Prosa [15]. Follows from Algorithm 49, since suspension intervals in schedule $\mathcal{S}'$ are no longer than those in $\mathcal{S}$ and the fact that the dynamic self-suspension model imposes only an upper bound on total job suspension time, and since by construction the derived schedule $\mathcal{S}'$ is work-conserving, respects self-suspensions, and respects job priorities. ◀

4.4 Main Claim

Based on the strategy explained in §4.2, by combining Theorems 52 and 53, we prove that the scheduling policy is weakly sustainable.

▶ **Theorem 54** (Weak Sustainability). *Uniprocessor JLFP scheduling of sporadic self-suspending tasks under the dynamic suspension model is weakly sustainable with respect to job costs and variable suspension times.*

Proof. Proven in Prosa [15]. Instantiate Definition 35 with uniprocessor JLFP scheduling of sporadic self-suspending tasks under the dynamic suspension model for $S = \{cost\}$ and $V = \{susp\}$. Theorems 52 and 53 imply that, for any schedule $\mathcal{S}$ of job set $\mathcal{J}$ that contains a deadline miss, there exists a schedule $\mathcal{S}'$ of the transformed set $\mathcal{J}'_{worse}$ that also contains a deadline miss. ◀

We emphasize that Algorithm 49 builds a schedule $\mathcal{S}'$ and hence a job set $\mathcal{J}'_{worse}$ that has different suspension times than the original job set $\mathcal{J}$. Therefore, the presented argument indeed proves the *weak* (but not strong) sustainability of uniprocessor JLFP scheduling under the dynamic self-suspending task model w.r.t. job costs and *variable suspension times.*

5 Conclusion and Future Work

Sustainability is a central aspect of real-time theory with many applications in the development of real-time systems. By allowing system designers to target only extreme scheduling scenarios, it simplifies the design, prototyping, and analysis of real-time systems. In addition, the use of sustainable scheduling policies and analyses greatly aids the validation and certification process, by ensuring that only a subset of execution scenarios must be checked, and that any variation within the system's specified bounds does not compromise safety.

In this paper, we have identified that the existing notions of sustainability in real-time scheduling allow for multiple interpretations with regard to whether real-time scheduling of self-suspending tasks is sustainable with respect to job costs. To resolve the issue, we developed a precise sustainability theory for real-time scheduling that is compatible with any task and platform model (§2), and also proposed the new notions of strongly and weakly sustainable policies (§3), which can be used to derive more efficient schedulability analyses for policies that were shown to not be strongly sustainable.

To better understand a model for which many mistakes were found in the literature [8], we chose to study weak sustainability in the context of self-suspending tasks. For that, we developed a generic model for self-suspensions (§4.1) that was formalized in the Coq proof assistant and integrated into Prosa [15, 7]. Finally, we mechanically proved in Prosa that uniprocessor JLFP scheduling of self-suspending tasks is weakly sustainable with respect to job costs and variable suspension times (§4.2–§4.4).

In ongoing work, we are working towards leveraging the obtained weak sustainability result to derive new, machine-checked schedulability tests for the dynamic suspension model. In future work, it will be interesting to identify instances of weakly sustainable parameters in other task models, platforms, and scheduling algorithms to improve the complexity of their associated timing analyses and to lessen test coverage requirements.

References

1 Yasmina Abdeddaïm and Damien Masson. The scheduling problem of self-suspending periodic real-time tasks. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS'12, pages 211–220, 2012.

2 Theodore P. Baker and Sanjoy K. Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, ECRTS '09, pages 141–150, 2009.

3 Sanjoy Baruah. Scheduling periodic tasks on uniform multiprocessors. *Information Processing Letters*, 80(2):97–104, 2001.

4 Sanjoy Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th Real-Time Systems Symposium*, RTSS'04, pages 37–46. IEEE, 2004.

5 Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Proceedings of the 27th Real-Time Systems Symposium*, RTSS'06, pages 159–168. IEEE, 2006.

6 Alan Burns and Sanjoy Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

7 Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, ECRTS'16, pages 273–284. IEEE, 2016.

8 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, and Dionisio de Niz. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Department of Computer Science, TU Dortmund, 2016.

9 Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

10 Rhan Ha. *Validating Timing Constraints in Multiprocessor and Distributed Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

11 Rhan Ha and Jane WS Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, ICDCS'94, pages 162–171. IEEE, 1994.

12 In-Guk Kim, Kyung-Hee Choi, Seung-Kyu Park, Dong-Yoon Kim, and Man-Pyo Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, RTCSA'95, pages 54–59. IEEE, 1995.

13 Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

14 Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

15 Prosa – Weak Sustainability. Supplemental material and formal proofs, `http://prosa.mpi-sws.org/releases/sustainability`.

16 Ragunathan Rajkumar. Dealing with suspending periodic tasks. *IBM Thomas J. Watson Research Center*, 1991.

17 Dongkun Shin and Jihong Kim. A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 271–274. ACM/IEEE, 2001.