# 33rd Euromicro Conference on Real-Time Systems

ECRTS 2021, July 5–9, 2021, Virtual Conference

Edited by

Björn B. Brandenburg

LIPIcs

LIPIcs – Vol. 196 – ECRTS 2021 www.dagstuhl.de/lipics

*Editor*

**Björn B. Brandenburg**
Max Planck Institute for Software Systems, Kaiserslautern, Germany
bbb@mpi-sws.org

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

## ISBN 978-3-95977-192-4

*Published online and open access by*
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-192-4.

*Publication date*
July, 2021

*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 https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.ECRTS.2021.0

**ISBN 978-3-95977-192-4** **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

## 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**

**https://www.dagstuhl.de/lipics**

# Contents

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# Preface

## Message from the Chairs

It is our pleasure to welcome you to ECRTS 2021, the second — and hopefully last — fully virtual instance of the conference. ECRTS is the premier European conference series in the area of real-time systems and, alongside RTSS and RTAS, ranks as one of the top three international conferences on this topic.

As we all wait for travel and life in general to normalize again, we are delighted to have you join us for an exciting online program consisting of both scientific talks and opportunities for socializing and networking. The centerpiece of the program will be a series of live presentations introducing new results spanning the entire domain of real-time systems, from algorithmic foundations to applied systems.

ECRTS 2021 received a total of 84 submissions from Asia, Europe, and North America. Each submission was reviewed by at least three expert members of the program committee (PC) and discussed at a virtual PC meeting that took place on April 19 and 20, 2021. Ultimately, the PC decided to accept 16 papers for publication and presentation, which translates to an acceptance rate of 19%.

ECRTS has been at the forefront of recent innovations in the real-time systems community such as artifact evaluation and open-access proceedings. Continuing its tradition of innovation, ECRTS trialed a flexible page limit this year. We believe that scientists should focus on the content of their papers, and not worry too much about formatting tricks and layout micro-optimizations to squeeze the last few paragraphs under a given hard page limit. Authors should invest their time into making their manuscripts more compelling and more appealing to readers, not into fighting LaTeX to comply with ultimately somewhat arbitrary page limits. In this spirit, rather than policing formatting violations "with an iron fist," the flexible page limit introduced this year aimed at reducing the incentive for space hacks in the first place by giving authors the option to submit manuscripts exceeding the typical length of 15-18 pages of content.

As the flexible page limit is a "new feature" without precedent in the community, cautious rules were put in place. These rules required authors to provide a short justification of their need for extra pages, and to obtain *a priori* permission from the PC Chair to submit a manuscript exceeding the 18-page soft limit. Similarly, the policy allowed authors to request additional pages for the camera-ready versions of their papers. This allowed reviewers and shepherds to ask for expanded discussions, and gave authors the liberty to address reviewer feedback fully even if it required additional space. The resulting variation in paper lengths is reflected in these proceedings.

Ultimately, 9 out of 84 submissions made use of the flexible page policy to submit manuscripts exceeding 18 pages of content (10.7%). Among the 16 papers accepted for publication, 2 comprised more than 18 pages of content at the time of submission (12.5%). It should be noted that the flexible page limit did not result in excessive amounts of content that would have exceeded the limits or nature of a conference paper, which is perhaps not surprising as concision is of course a hallmark of good academic writing.

Overall, we believe that the flexible page policy is a success in two ways: the PC was freed from concerning itself with formatting minutiae and the policy made a positive difference for some of the authors who opted to make use of it. However, uptake by the community was

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

more subdued than expected. As of now, it is still undecided whether the page limit will remain flexible (in a revised manner) in future years, or wether the conference series will revert to a more traditional hard limit.

Double-blind peer reviewing is another innovation successfully adopted in 2021, meaning that authors submitted blinded manuscripts that left reviewers unaware of the names and affiliations of the authors. As a result, all major conferences of the real-time systems community now follow a largely similar double-blind peer-reviewing process, which we welcome as significant community-wide change for the better that has been accomplished in just a few short years, thanks to the efforts by many in the community. We are thankful to have had the opportunity to play a small role in this transition and are hopeful that it will promote the fairness and the meritocratic nature of the evaluation process.

A major conference such as ECRTS rests on many shoulders. First of all, we thank the PC members for their hard work and outstanding service, and in particular for delivering high-quality reviews on time despite a very tight timeline and all the burdens of a strange and difficult year. Similarly, we are grateful to all external and secondary reviewers, who provided many valuable perspectives and important feedback. We are especially grateful to those PC members and additional reviewers who went "above and beyond" serving as anonymous shepherds — you know who you are. We would also like to extend our thanks to the Artifact Evaluation Chairs Alessandro Biondi and Angeliki Kritikakou and their board of Artifact Evaluators for running the AE process. Finally, we thank the new Euromicro Real-Time Technical Committee (TC) for its trust in us and their valuable guidance along the way.

Our very special thanks go to the former, long-serving Euromicro Real-Time TC Chair Gerhard Fohler for making ECRTS what it is today. Thank you, Gerhard! You have built and nurtured something very special here. The new TC will have to work hard to live up to the example you set.

Last but not least, we thank all authors for submitting their work to ECRTS 2021. Whether or not it was ultimately accepted for publication, we deeply appreciate your fine work and the tremendous effort and care that has gone into it; this conference would not be possible without you.

Thanks to the authors, we are looking forward to an inspiring, high-quality program. Please join us in enjoying both the science and everything around it — not just despite, but *especially* in these trying times.

| | |
|---|---|
| Marcus Völp | Björn Brandenburg |
| General Chair, ECRTS 2021 | Program Chair, ECRTS 2021 |

# Organizers

## Euromicro Real-Time Technical Committee

Sebastian Altmeyer, University of Augsburg, Germany
Sophie Quinton, INRIA Grenoble Rhône-Alpes, France
Marcus Völp, SnT, University of Luxembourg

## General Chair

Marcus Völp, SnT, University of Luxembourg

## Program Chair

Björn B. Brandenburg, Max Planck Institute for Software Systems (MPI-SWS), Germany

## Artifact Evaluation Chairs

Alessandro Biondi, Scuola Superiore Sant'Anna – Pisa, Italy
Angeliki Kritikakou, IRISA, Rennes, France

## Program Committee

Benny Akesson, University of Amsterdam / TNO, The Netherlands
Sebastian Altmeyer, University of Augsburg, Germany
Jim Anderson, University of North Carolina at Chapel Hill, USA
Sanjoy Baruah, Washington University in St. Louis, USA
Enrico Bini, Università degli Studi di Torino, Italy
Konstantinos Bletsas, CISTER, ISEP, Polytechnic Institute of Porto, Portugal
Florian Brandner, Télécom Paris, France
Giorgio Buttazzo, Scuola Superiore Sant'Anna – Pisa, Italy
Marco Caccamo, TU Munich, Germany
Daniel Casini, Scuola Superiore Sant'Anna – Pisa, Italy
Francisco Cazorla, Barcelona Supercomputing Center, Spain
Thidapat Chantem, Virginia Tech, USA
Jian-Jia Chen, TU Dortmund, Germany
Dakshina Dasari Robert Bosch GmbH, Germany
Robert Davis University of York, UK
Pontus Ekberg, Uppsala University, Sweden
Rolf Ernst, TU Braunschweig, Germany
Nathan Fisher, Wayne State University, USA
Gerhard Fohler, TU Kaiserslautern, Germany
Joël Goossens, Université libre de Bruxelles ULB, Belgium
Giovani Gracioli, Federal University of Santa Catarina, Brazil
Mohamed Hassan, McMaster University, Canada
Angeliki Kritikakou, Univ Rennes, Inria, IRISA, France
Martina Maggio, Saarland University, Germany
Renato Mancuso, Boston University, USA
Ahlem Mifdaoui, University of Toulouse, France

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Mitra Nasri, Eindhoven University of Technology, The Netherlands
Claire Pagetti, ONERA, France
Alessandro Papadopoulos, Mälardalen University, Sweden
Gabriel Parmer, George Washington University, USA
Risat Mahmud Pathan, Zenseact AB, Sweden
Rodolfo Pellizzoni, University of Waterloo, Canada
Isabelle Puaut, Université de Rennes 1/ IRISA, France
Christine Rochange, University of Toulouse, France
Selma Saidi, TU Dortmund, Germany
Simon Schliecker, Volkswagen AG, Germany
Corey Tessler, Towson University, USA
Marcus Völp, University of Luxembourg
Georg von der Brüggen, Max Planck Institute for Software Systems (MPI-SWS), Germany
Peter Wägemann, Friedrich-Alexander University Erlangen-Nürnberg, Germany
Heechul Yun, University of Kansas, USA

## Artifact Evaluators

Tanya Amert, University of North Carolina at Chapel Hill, USA
Matthias Becker, KTH, Sweden
Bryan Donyanavard, San Diego State University, USA
Romain Jacob, ETH Zurich, Switzerland
Leonidas Kosmidis, Barcelona Supercomputing Center, Spain
Paolo Pazzaglia, Saarland University, Germany
Benjamin Rouxel, University of Amsterdam, The Netherlands
Fernando Fernandes dos Santos, Universidade Federal do Rio Grande do Sul, Brazil
Lea Schönberger, TU Dortmund, Germany
Stefanos Skalistis, Collins Aerospace, Ireland

## Additional Reviewers

Jaume Abella
Shareef Ahmed
Ibrahim Alkoudsi
Abderaouf Nassim Amalou
Mihail Asavoae
Muhammad Ali Awan
Zhenyu Bai
Joshua Bakita
Nicolas Bellec
Antoine Bertout
Benjamin Binder
Alessandro Biondi
Tobias Blass
Frédéric Boniol
Étienne Borde
Marc Boyer
Sergey Bozhko
Thomas Carle
Hugues Cassé
Pierre-Julien Chaine
Kuan-Hsun Chen
Mitchell Duncan
Bssel El Mabsout
Ian Elmor Lang
Gautam Gala
Adrien Gauffriau
Golsana Ghaemi
Mario Günzel
Arne Hamann
Xinyu Han
Florian Heilmann
Denis Hoornaert
Mehdi Hosseinzadeh
Jeff Ichnowski
Tomasz Kloda
Leonidas Kosmidis
Kristin Krüger
Ching-Chi Lin
Felipe Lisboa
Claudio Mandrioli
Sean McBride
Enrico Mezzetti
Reza Mirosanlou
Tanmaya Mishra
Naresh Nayak
Geoffrey Nelissen
Catherine Nemitz
Federico Nesti
Luiz Neto
Sims Osborne
Marco Pagani
Runyu Pan
Paolo Pazzaglia
Sophie Quinton

Fatima Raadia
Jan Reineke
Tim Rheinfels
Carlos Rodriguez
Shahin Roozkhosh
Debayan Roy
Gero Schwäricke
Alejandro Serrano
Wenyuan Shao
Junjie Shi
Jayati Singh
Stefanos Skalistis
Parul Sohal
Pascal Sotin
Hamid Tabani
Stephen Tang
Dharmesh Tarapore
Niklas Ueter
Sergey Voronov
Aaron Willcock
Tyler Yandrofski
Patrick Meumeu Yomsi

# Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation

Michael Platzer
TU Wien, Institute of Computer Engineering, Austria

Peter Puschner
TU Wien, Institute of Computer Engineering, Austria

## Abstract

In this work, we present Vicuna, a timing-predictable vector coprocessor. A vector processor can be scaled to satisfy the performance requirements of massively parallel computation tasks, yet its timing behavior can remain simple enough to be efficiently analyzable. Therefore, vector processors are promising for highly parallel real-time applications, such as advanced driver assistance systems and autonomous vehicles. Vicuna has been specifically tailored to address the needs of real-time applications. It features predictable and repeatable timing behavior and is free of timing anomalies, thus enabling effective and tight worst-case execution time (WCET) analysis while retaining the performance and efficiency commonly seen in other vector processors. We demonstrate our architecture's predictability, scalability, and performance by running a set of benchmark applications on several configurations of Vicuna synthesized on a Xilinx 7 Series FPGA with a peak performance of over 10 billion 8-bit operations per second, which is in line with existing non-predictable soft vector-processing architectures.

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

**Keywords and phrases** Real-time Systems, Vector Processors, RISC-V

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.1

**Supplementary Material** *Software*: `https://github.com/vproc/vicuna`

## 1 Introduction

Worst-Case Execution Time (WCET) analysis, which is essential to determine the maximum execution time of tasks for real-time systems [46], has struggled to keep up with the advances in processor design. Numerous optimizations such as caches, branch prediction, out-of-order execution, and speculative execution have made the timing analysis of processing architectures increasingly complex [45]. As a result, the performance of processors suitable for real-time systems usually lags behind platforms optimized for average computational throughput at the cost of predictability. Yet, the performance requirements of real-time applications are growing, particularly in domains such as advanced driver assistance systems and self-driving vehicles [23], thus forcing system architects to use multi-core architectures and hardware accelerators such as Graphics Processing Units (GPUs) in real-time systems [13]. Analyzing the timing behavior of such complex heterogeneous systems poses additional challenges as it requires a timing analysis of the complex interconnection network in addition to analyzing the individual processing cores of different types and architectures [36, 9].

However, current trends motivated by the quest for improved energy-efficiency and the emergence of massively data-parallel workloads [8] have revived the interest in architectures that might be more amenable to WCET analysis [29]. In particular, vector processors are promising improved energy efficiency for data-parallel workloads [7] and have the potential to reduce the performance gap between platforms suitable for time-critical applications and mainline processors [29].

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 1; pp. 1:1–1:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Vector processors are single-instruction multiple-data (SIMD) architectures, operating on vectors of elements instead of individual values. The vector elements are processed simultaneously across several processing elements as well as successively over several cycles [2]. A single vector instruction can operate on a very large vector, thus amortizing the overhead created by fetching and decoding the instruction, which does not only increase its efficiency [4] but also means that complex hardware-level optimizations become less effective [29]. Therefore, vector processors can drop some of these optimizations and thus improve timing predictability without notable performance degradation.

While vector processors have the potential to greatly simplify timing analysis compared to other parallel architectures, existing vector processing platforms retain features that impact timing-predictability, such as out-of-order execution or banked register files [5]. Even if some vector architectures have simple in-order pipelines, they still exhibit timing anomalies (i.e., undesired timing phenomena which threaten timing predictability). Timing anomalies occur, for instance, when memory accesses are not performed in program order [16], such as when memory accesses by the vector unit interfere with accesses from the main core.

In this paper, we present a novel vector coprocessor addressing the needs of time-critical applications without sacrificing performance. Our key contributions are as follows:

1. We present a timing-predictable 32-bit vector coprocessor implemented in SystemVerilog that is fully compliant with the version 0.10 draft of the RISC-V vector extension [34]. All integer and fixed-point vector arithmetic instructions, as well as the vector reduction, mask, and permutation instructions described in the specification, have been implemented. Vicuna is open-source and available at `https://github.com/vproc/vicuna`.
2. We integrate our proposed coprocessor with the open-source RISC-V core Ibex [37] and show that this combined processing system is free of timing anomalies while retaining a peak performance of 128 8-bit multiply-accumulate (MAC) operations per cycle. The combined processing system runs at a clock frequency of 80 MHz on Xilinx 7 Series FPGAs, thus achieving a peak performance of 10.24 billion operations per second.
3. We evaluate the effective performance of our design on data-parallel benchmark applications, reaching over 90 % efficiency for compute-bound tasks. The evaluation also demonstrates the predictability of our architecture as each benchmark program always executes in the exact same number of CPU cycles.

This work is organized as follows. Section 2 introduces prior work in the domains of parallel processing and vector architectures. Then, Section 3 presents the design of our vector coprocessor Vicuna and Section 4 analyzes the timing behavior of our processing system. Section 5 evaluates its performance on several benchmark algorithms, and Section 6 concludes this article.

## 2 Background and Related Work

This section gives an overview of existing parallelized computer architectures and vector processors in particular and compares them to our proposed timing-predictable vector coprocessor Vicuna. Table 1 summarizes the main aspects.

### 2.1 Parallel Processing Architectures

In the mid-2000s, power dissipation limits put an end to the acceleration of processor clock frequencies, and computer architects were forced to exploit varying degrees of parallelism in order to further enhance computational throughput. A relatively simple approach is to

**Table 1** Performance and timing predictability of parallel computer architectures.

| Processor Architecture | Multi-Core CPU | General-purpose GPU | Domain-Specific Accelerators | Existing Vector Processors | Timing-Predictable Platforms | **Vicuna (Our work)** |
|---|---|---|---|---|---|---|
| General-purpose | ✓ | ✓ |  | ✓ | ✓ | ✓ |
| Efficient parallelism |  | ✓ | ✓ | ✓ |  | ✓ |
| Timing-predictable |  |  | ✓ |  | ✓ | ✓ |
| Max. OPs per sec ($\cdot 10^9$)<br>FPGA / ASIC | 2.2* / 1 200** | 3.2† / 35 000†† | 5 000‡ / 45 000‡‡ | 15§ / 128§§ | 2.4¶ / 49¶¶ | 10 / — |

* 16-core Cobham LEON3
** 344-core Ambric Am2045B
† FlexGrip soft GPU [1]
†† NVIDIA RTX 3090
‡ Srinivasan et al. [42]
‡‡ Google TPU [21]
§ 32-lane VEGAS [6]
§§ 16-lane PULP Ara [5]
¶ 15-core T-CREST Patmos [38]
¶¶ 8-core ARM Cortex-R82

replicate a processor core several times, thus creating an array of independent cores each executing a different stream of instructions. This multiple-instruction, multiple-data (MIMD) paradigm [11] is ubiquitous in today's computer architectures and has allowed a continued performance increase. Timing-predictable multi-core processors have been proposed for time-critical parallel workloads, most notably the parMERASA [43] and the T-CREST [38] architectures, which demonstrated systems with up to 64 and 15 cores, respectively. A similar timing-predictable multi-core architecture utilizing hard processing cores connected by programmable logic has been implemented recently on an Multiprocessor System-on-Chip (MPSoC) platform [14]. However, several of the workloads capable of efficiently exploiting this parallelism are actually highly data-parallel, and as a consequence, the many cores in such a system frequently all execute the same sequence of instructions [7]. The fetching and decoding of identical instructions throughout the cores represent a significant overhead and increase the pressure on the underlying network infrastructure connecting these cores to the memory system [28, 41]. Consequently, the effective performance of a multi-core system does not scale linearly as more cores are added. For the T-CREST platform, Schoeberl et al. report that the worst-case performance for parallel benchmark applications scales only logarithmically with the number of cores [38]. As an alternative to multi-core architectures, some timing-predictable single-core processors exploit parallelism by executing multiple independent hardware threads [26, 50], thus avoiding the overhead of a complex interconnection network. Yet, the scalability of this approach is limited since it does not increase the available computational resources.

An architecture that overcomes many of the limitations of multi- and many-core systems for highly parallel workloads are general-purpose GPUs (also referred to as GPGPUs) [31]. GPUs utilize data-parallel multithreading, referred to as the single-instruction multiple-threads (SIMT) paradigm [27], to achieve unprecedented energy-efficiency and performance. GPUs are used as data-parallel accelerators in various domains and have found their way into safety-critical areas such as autonomous driving [23, 13]. However, their use in hard real-time systems still poses challenges [9]. GPUs are usually non-preemptive, i.e., tasks cannot be interrupted, which requires software-preemption techniques to be used instead [13]. Also, contention among tasks competing for resources is typically resolved via undisclosed arbitration schemes that do not account for task priorities [10].

Recently, special-purpose accelerators emerged as another type of highly parallel platform that sacrifices flexibility and often precision [42] to achieve impressive performance for domain-specific tasks. For instance, the Tensor Processing Unit (TPU) [21] is capable of 65536 8-bit MAC operations in one cycle, achieving a peak performance of $45 \cdot 10^{12}$ operations per second at a clock frequency of 700 MHz. Due to their simple application-specific capabilities, the timing behavior of these accelerators is generally much easier to analyze [29]. While domain-specific accelerators achieve impressive performance for a small subset of applications, they are very inefficient at or even incapable of running other important algorithms, such as Fourier Transforms, motion estimation, or encryption with the Advanced Encryption Standard (AES). By contrast, a vector processor can execute any task that can be run on a conventional processor.

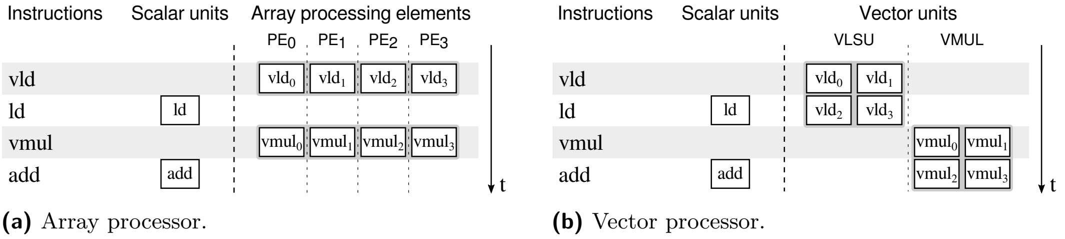
As an alternative to parallelizing tasks across several cores or threads, single-instruction multiple-data (SIMD) arrays have been added to several Instruction Set Architectures (ISAs). These are usually fixed-size arrays using special functional units, one for each element in the array, to apply the same operation to the entire array at once. However, array processors require that the computational resources are replicated for each element of the longest supported array [5].

## 2.2 Vector Processors

Vector processors are a time-multiplexed variant of array processors. Instead of limiting the vector length by the number of processing elements, a vector processor has several specialized execution units that process elements of the same vector across multiple cycles, thus enabling the dynamic configuration of the vector length [7]. Fig. 1 shows how an instruction stream with interleaved scalar and vector instructions executes on an array processor and a vector processor, respectively. In an array processor, the entire vector of elements is processed at once, and the processing elements remain idle during the execution of scalar instructions. In the vector processor, functionality is distributed among several functional units, which can execute in parallel with each other as well as concurrently with the scalar units.

Vector processors provide better energy-efficiency for data-parallel workloads than MIMD architectures [7] and promise to address the van Neumann bottleneck very effectively [4]. A single vector instruction can operate on a very large vector, which amortizes the overhead created by fetching and decoding the instruction. In this regard, vector processors even surpass GPUs, which can only amortize the instruction fetch over the number of parallel execution units in a processing block [5].

Several supercomputers of the 1960s and 1970s were vector processors, such as the Illiac IV [19] or the Cray series [35]. These early vector processors had functional units spread across several modules containing thousands of ICs in total. At the end of the century, they



**(a)** Array processor.

**(b)** Vector processor.

**Figure 1** Comparison of the execution patterns of array and vector processors. Instructions prefixed with a *v* operate on a vector of elements, while the rest are regular scalar instructions.

were superseded by integrated microprocessor systems, which surpassed their performance and were significantly cheaper [2]. While disappearing from the high-performance computing domain, vector processors have continued their existence as general-purpose accelerators in Field-Programmable Gate Arrays (FPGAs). Several soft vector processors have been presented, such as VESPA [48], which adds a vector coprocessor to a 3-stage MIPS-I pipeline, VIPERS [49], a single-threaded core with a vector processing unit, VEGAS [6], a vector coprocessor using a cacheless scratchpad memory, VENICE [39], an area-efficient improved version of VEGAS, or MXP [40], which added additional support for fixed-point computation.

In addition to FPGA-based accelerators, vector processors have also been explored as energy-efficient parallel computing platforms. Lee et al. [25] proposed a vector architecture named Hwacha, which is based on the open RISC-V ISA. The instruction set for Hwacha has been implemented as a custom extension. Despite sharing some features, it is incompatible with the more recent official RISC-V vector extension. One of the first vector processors based on the new RISC-V V extension is Ara, developed by Cavalcante et al. [5], as a coprocessor for the RISC-V core Ariane. Another recent architecture implementing the RISC-V V extension named RISC-V$^2$ has been proposed by Patsidis et al. [32].

While existing vector processors are less complex and easier to analyze than other parallel architectures, they still use speed-up mechanisms which are a source of timing anomalies, such as run-time decisions for choosing a functional unit [44], banked register files, and greedy memory arbitration [16]. By contrast, our proposed vector processor avoids such mechanisms, with negligible impact on its performance thanks to the vector processing paradigm's inherent effectiveness. Vicuna is free of timing anomalies and hence suitable for compositional timing analysis.

## 3 Architecture of Vicuna

This section introduces the architecture of Vicuna, a highly configurable, fully timing-predictable 32-bit in-order vector coprocessor implementing the integer and fixed-point instructions of the RISC-V vector extension. The RISC-V instruction set is an open standard ISA developed by the RISC-V foundation. It consists of a minimalist base instruction set supported by all compliant processors and several optional extensions. The V extension adds vector processing capabilities to the instruction set. RISC-V and the V extension are supported by the GNU Compiler Collection (GCC) and the LLVM compiler.

Vicuna is a coprocessor and must be paired with a main processor. We use the 32-bit in-order RISC-V core Ibex, developed initially as part of the PULP platform under the name Zero-riscy [37], as the main processor. Ibex is a small core with only two pipeline stages: an instruction fetch stage and a combined decode and execute stage. Ibex executes all non-vector instructions, which we refer to as scalar instructions.

Vicuna is connected to the main core with a coprocessor interface through which instruction words and the content of registers are forwarded from the main core to the coprocessor, and results can be read back. We added a coprocessor interface to Ibex to extend it with Vicuna. Instruction words are forwarded to the vector core via this interface if the major opcode indicates that it is a vector instruction. In addition to the instruction word, scalar operands from the main core's register file are also transmitted to the coprocessor since these are required by some vector instructions which use the scalar registers as source registers, such as for instance, a variant of the vector addition which adds a scalar value to every element of a vector or the vector load and store instructions which read the memory address from a scalar register.

An overview of the architecture of Vicuna and its integration with Ibex as the main core is shown in Fig. 2. Vicuna comprises a decoder for RISC-V vector instructions, which parses and acknowledges valid vector instructions. Once Vicuna's decoder has successfully decoded a vector instruction, it acknowledges its receipt and informs the main core whether it needs to wait for a scalar result. If the vector instruction produces no scalar result but instead only writes to a vector register or memory, then the main core can proceed with further instructions in parallel with the vector instruction's execution on the coprocessor. However, when a vector instruction writes back to a register in the main core, then the main core stalls until the coprocessor has completed that instruction. Only four RISC-V vector instructions produce a scalar result. Hence this scenario occurs rarely. Decoded vector instructions are placed in an instruction queue where they await execution on one of the vector core's functional units. Vicuna is a strictly in-order coprocessor: Vector instructions from the instruction queue are issued in the order they are received from the main core. A vector instruction is issued as soon as any data hazards have been cleared (i.e., any instructions producing data required by that instruction are complete) and the respective functional unit becomes available.

Since our main goal is to design a timing-predictable vector processor, we refrain from any features that cause timing anomalies, such as run-time decisions for choosing functional units [44]. Both cores share a common 2-way data cache with a least recently used (LRU) replacement policy, which always gives precedence to accesses by the vector core. Once a vector instruction has been issued for execution on one of the functional units, it completes within a fixed amount of time that depends only on the instruction type, the throughput of the unit, and the current vector length setting. For vector loads and stores, the execution time additionally depends on the state of the data cache, which is the only source of timing variability. However, in-order memory access is guaranteed for scalar and vector memory

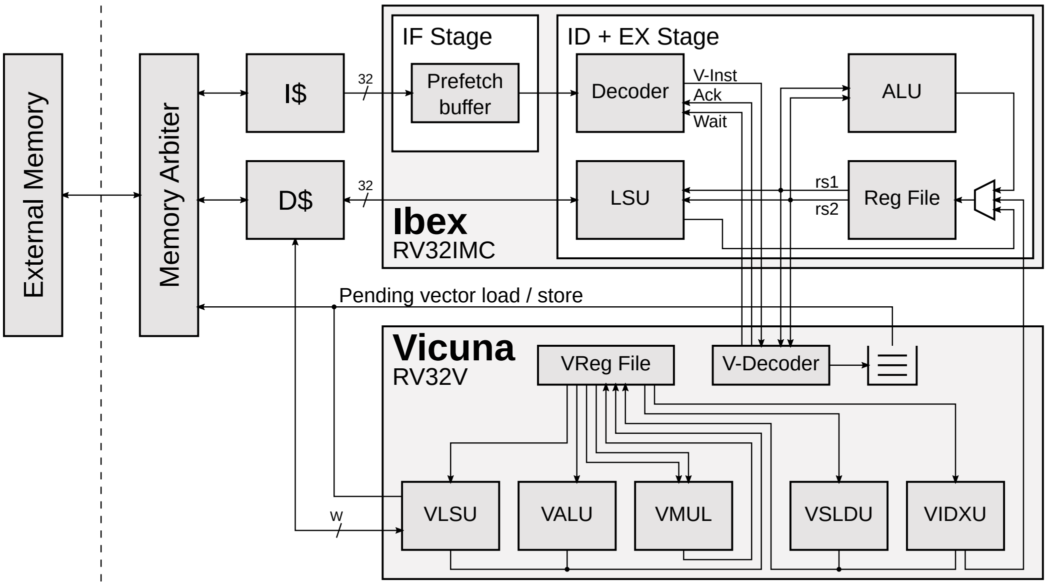


**Figure 2** Overview of Vicuna's architecture and its integration with the main core Ibex. Both cores share a common data cache. To guarantee in-order memory access, the memory arbiter delays any access following a cache miss by the main core until pending vector load and store operations are complete. When accessing the data cache, the vector core always takes precedence.

operations by delaying any access following a cache miss in the main core until pending vector load and stores are complete. Note that vector load and store instructions stall the main core for a deterministic, bounded number of cycles since no additional vector instructions can be forwarded to the vector core while the main core is stalled. This method is an extension of the technique introduced by Hahn and Reineke [15] for the strictly in-order core SIC. Due to the simple 2-stage pipeline of Ibex, conflicting memory accesses between its two stages become visible simultaneously. In that situation, the memory arbiter maintains strict ordering by serving the data access first.

Vicuna comprises several specialized functional units, each responsible for executing a subset of the RISC-V vector instructions, which allows executing multiple instructions concurrently. The execution units do not process an entire vector register at once. Instead, during each clock cycle, only a portion of the vector register is processed, which may contain several elements that are processed in parallel. Most array processors and several vector processors are organized in lanes. Each lane replicates the computational resources required to process one vector element at a time. In such a system, the number of lanes determines the number of elements that can be processed in parallel, regardless of the type of operation. By contrast, Vicuna uses dedicated execution units for different instruction types that each process several elements at once. The ability to individually configure the throughput for each unit improves the performance of heavily used operations by increasing the respective unit's data-path width (e.g., widening the data-path of the multiplier unit).

Some of the RISC-V vector instructions do not process the vector registers on a regular element-wise basis. Instead, they feature an irregular access pattern, such as indexed instructions, which use one vector register's values as indices for reading elements from another register, or the slide instructions, which slide all elements in a vector register up or down that register. Vicuna uses different functional units for each vector register access pattern, which allows us to implement regular access patterns more efficiently and hence to improve the throughput of the respective unit, while complex access patterns require more cycles.

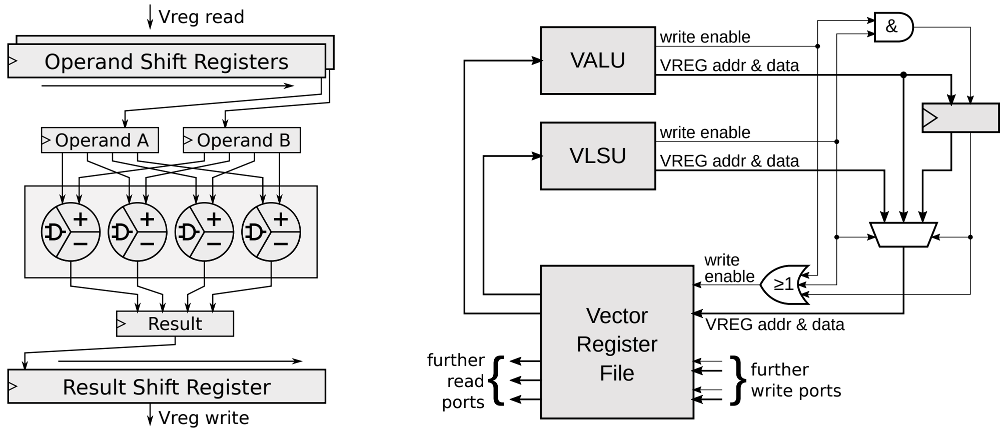
Vicuna comprises the following execution units:

- A *Vector Load and Store Unit* (VLSU) interfaces the memory and implements the vector memory access instructions.
- The *Vector Arithmetic and Logical Unit* (VALU) executes most of the arithmetic and logical vector instructions.
- A dedicated *Vector Multiplier* (VMUL) is used for vector multiplications.
- The *Vector Slide Unit* (VSLDU) handles vector slide instructions that move all vector elements up or down that vector synchronously.
- A *Vector Indexing Unit* (VIDXU) takes care of the indexing vector instructions. It is the only unit capable of writing back to a scalar register in the main core.

The VALU uses a fracturable adder for addition and subtraction, that consists of a series of 8-bit adders whose carry chains can be cascaded for wider operations. Four cascaded 8-bit adders perform four 8-bit, two 16-bit, or one 32-bit operation depending on the current element width. Similarly, the VMUL unit uses a fracturable multiplier to perform 8-bit, 16-bit, and 32-bit multiplications on the same hardware. Fracturable adders and multipliers are commonly used for FPGA-based vector processors. We base our implementation on the resource-efficient design that Chou et al. proposed for the VEGAS vector processor [6].

Selecting a relatively large sub-word from a large vector register consumes a substantial amount of logic resources. Therefore, we avoid sub-word selection logic for all functional units with a regular vector register access pattern. Instead, these units read the whole source vector registers into shift registers, as shown in Fig. 3 (a). The content of these is then shifted by the number of elements that can simultaneously be processed by the unit each cycle, thus making the next elements of the source vector register available to the processing pipeline. Similarly, the results are aggregated into another shift register that saves the computed elements until the entire vector is complete, upon which the whole vector register is written back to the register file. The amount of combinatorial logic resources consumed by the shift registers is less than those that are required by an index-based subword selection (they do, however, require some extra flip-flops for buffering the whole vector register).

Vicuna's vector register file contains 32 vector registers of configurable width. Multiple read and write ports are required in order to supply the execution units operating in parallel with operands and consume their results. We take advantage of the functional unit's shift registers, which fetch entire vector registers at once and accumulate results before storing a whole register, to implement both read and write port multiplexing. Each functional unit has a dedicated read port used to fetch the operand registers sequentially, storing them in shift registers from where they are consumed iteratively. This adds one extra cycle when fetching two operand registers but avoids the need for two read ports on each unit. As the only exception, the VMUL unit has two read ports to better support the fused multiply-add instruction, which uses three operands. Also, write ports are shared between units using the circuitry shown in Fig. 3 (b). Due to the accumulation of results in shift registers prior to write-back, a unit cannot write to the vector register file for two subsequent cycles. Hence, whenever a collision between two units occurs on a shared write port, one unit takes precedence and writes its result back first while the other unit writes its result into a temporary buffer, from where it is stored to the register file in the subsequent cycle. A second write request from the first unit cannot immediately follow the previous write. Hence this delayed write-back is guaranteed to succeed. Regardless of whether the write-back is delayed by one cycle or not, any data hazards of operations on units not taking precedence on their shared write port are cleared one cycle after the operation completes to maintain predictable instruction timings while accounting for a potentially delayed write-back.



**(a)** Organization of the vector ALU. Operand registers are read sequentially into shift registers and consumed over several cycles by processing a fixed-width portion each cycle. Results are again accumulated into a shift register before write-back.

**(b)** The VALU and VLSU share a common write port, with the VLSU always taking precedence. In case of a collision, the value and address of the VALU write request are temporarily saved and written to the vector register file in the next cycle. Neither unit can write for two subsequent cycles. Hence the delayed write always succeeds.

**Figure 3** Reading and writing whole registers from the vector register file avoids subword selection logic and allows multiplexing of read and write ports without affecting timing predictability.

Although multiplexing of both read and write ports is used to reduce the required number of ports, the vector register file must still provide several concurrent ports. We decided against banked registers, which allow concurrent access to registers of different banks but introduce interdependencies between execution units which are a potential source of timing anomalies in case two registers within the same bank are accessed simultaneously. Since a large flip-flop-based register file does not scale well, we implemented it as multi-ported RAM. The design has been inspired by work from Laforest et al. [24], who investigated ways of constructing efficient multi-ported RAMs in FPGAs. We implemented it as an XOR-based RAM since this allows selectively updating individual elements of a vector register for masked operations.

## 4 Timing-Predictability

In this section, we analyze the timing-predictability of Vicuna and argue that it is free of timing anomalies, thus enabling compositional timing analysis.

Timing predictability and timing compositionality are both essential properties to avoid the need for exhaustively exploring all possible timing behaviors for a safe WCET bound estimation. In particular, timing compositionality is necessary to safely decompose a timing analysis into individual components and derive a global worst case based on local worst-case behavior [18]. The presence of timing anomalies can violate both timing predictability and compositionality.

A timing anomaly can either be a counterintuitive timing effect or a timing amplification. Counterintuitive timing anomalies occur whenever the locally better case leads to a globally worse case, such as a cache hit leading to increased global timing, thus inverting the expected behavior. Amplification timing anomalies occur when a local timing variation induces a larger global timing variation. While counterintuitive timing anomalies threaten the timing predictability, amplification timing anomalies affect the timing compositionality [20].

Counterintuitive timing anomalies can occur, for instance, when an execution unit is selected at run-time rather than statically [44]. In-order pipelines can also be affected by this kind of anomalies for instructions with multi-cycle latencies [3]. While vector instructions executed within Vicuna can occupy the respective functional unit for several cycles, there is only one unit for each type of instruction, and hence there is no run-time decision involved in the choice of that unit. The execution time of all vector instructions is completely deterministic, thus avoiding counterintuitive timing anomalies.

Amplification timing anomalies can be more subtle to discover, as recently shown by Hahn et al. [17], who identified the reordering of memory accesses on the memory bus as another source for timing anomalies. The presence of amplification timing anomalies is due to the non-monotonicity of the timing behavior w.r.t. the progress order of the processor pipeline [15].

We show that Vicuna is free of amplification timing anomalies by extending the formalism introduced by Hahn and Reineke [15] for their timing-predictable core SIC to our vector processing system. A program consists of a fixed sequence of instructions $\mathcal{I} = \{i_0, i_1, i_2, \dots\}$. During the program's execution, the pipeline state is a mapping of each instruction to its current progress. The progress $\mathcal{P} := \mathcal{S} \times \mathbb{N}_0$ of an instruction is given by the pipeline stage $s \in \mathcal{S}$ in which it currently resides, as well as the number $n \in \mathbb{N}_0$ of cycles remaining in that stage. For our processing system, comprising the main core Ibex and the vector coprocessor Vicuna, we define the following set of pipeline stages:

$$\mathcal{S} = \{pre, IF, ID{+}EX, VQ, VEU, post_S, post_V\}$$

Analogous to the pipeline model used by Hahn and Reineke [15], we use the abstract stages *pre* and *post* to model instructions that have not yet entered the pipeline or have already left the pipeline, respectively. However, we distinguish between completed regular (scalar) instructions and completed vector instruction by dividing the *post* stage into $\mathit{post}_{\mathcal{S}}$ and $\mathit{post}_{V}$, respectively. *IF* is the main core's fetch, while *ID+EX* denotes its combined decode and execute stage. The vector coprocessor is divided into two abstract stages: *VQ* represents the vector instruction queue, and *VEU* comprises all the vector execution units. Vector instructions awaiting execution in the vector queue remain in program order, and once a vector instruction has started executing on one of the vector core's functional units, it is no longer dependent on any other instruction since there are no interdependencies between the individual vector units. Hence we do not need to explicitly model each of the concrete stages in the vector core.

Guaranteeing the strict ordering of instructions requires the following ordering $\sqsubset_{\mathcal{S}}$ of these pipeline stages:

$$\mathit{pre} \sqsubset_{\mathcal{S}} \mathit{IF} \sqsubset_{\mathcal{S}} \mathit{ID{+}EX} \begin{array}{l} \sqsubset_{\mathcal{S}} \mathit{post}_{\mathcal{S}} \\ \sqsubset_{\mathcal{S}} \mathit{VQ} \sqsubset_{\mathcal{S}} \mathit{VEU} \sqsubset_{\mathcal{S}} \mathit{post}_{V} \end{array}$$

Non-vector instructions exit the pipeline after the *ID+EX* stage, while vector instructions enter the vector queue and eventually start executing on a vector execution unit. An instruction that has fewer remaining cycles in a stage or is in a later stage than another instruction has made more progress. Hence, for two instruction with current progress $(s, n), (s', n') \in \mathcal{P}$ respectively, an order on the progress is defined as:

$$(s, n) \sqsubseteq_{\mathcal{P}} (s', n') \Leftrightarrow s \sqsubset_{\mathcal{S}} s' \vee (s = s' \wedge n \geq n')$$

The cycle behavior of a pipeline is monotonic w.r.t. the progress order $\sqsubseteq_{\mathcal{P}}$, if an instruction's execution cannot be delayed by other instructions making more progress. For this property to hold, an instruction's progress must depend on previous instructions only and never on a subsequent instruction [20]. Instructions are delayed by stalls in the pipeline. Hence any pipeline stage must only be stalled by a subsequent stage.

The vector execution units cannot stall, except for the vector load and store unit in case of a cache miss. Due to the strict ordering of memory accesses, the vector core cannot be delayed by a memory access of the main core. Hence the *VEU* stage cannot be stalled by any other stage. The vector queue holds instructions that await execution on a vector unit. Thus the *VQ* stage can only be stalled by the *VEU* stage. The *ID+EX* stage, in turn, can be stalled by an ongoing memory access of the vector core (the *VEU* stage), by a vector instruction writing back to a scalar register, when a vector instruction has been decoded, but the vector queue is full, or during memory loads and stores. Loads and stores are executed while the *IF* stage fetches the next instruction. Hence in case of an instruction cache miss on the subsequent instruction, a memory access by the *ID+EX* takes precedence over the *IF* stage. Finally, the *IF* stage can be stalled by the *ID+EX* or by a memory access of the vector core. Therefore, any pipeline stage of our processing system can only be stalled by a subsequent stage. Hence, the progress order $\sqsubseteq_{\mathcal{P}}$ of instructions is always maintained, and instructions can only be delayed by previous instructions, but not by subsequent ones. Consequently, the cycle behavior of our architecture is monotonic and hence free of timing anomalies, which in turn is a sufficient condition for timing compositionality [20].

# 5 Evaluation

This section evaluates our vector coprocessor's performance by measuring the execution time of parallel benchmark applications on a Xilinx 7 Series FPGA with an external SRAM with a 32-bit memory interface and five cycles of access latency. We evaluate a small, medium, and fast configuration of Vicuna with vector register lengths of 128, 512, and 2048 bits, respectively. Table 2 lists the parameters for each configuration, along with the peak multiplier performance and the maximum clock frequency.

The performance of parallel computer architectures on real-world applications is often degraded by various bottlenecks, such as the memory interface. While a large number of parallel cores or execution units might yield an impressive theoretical performance figure, efficiently utilizing these computing resources can be challenging. The roofline model [47] visualizes the performance effectively achieved by application code w.r.t. a processor's peak performance and memory bandwidth. The model shows the theoretical peak performance in operations per cycle in function of the arithmetic intensity, which is the ratio of operations per byte of memory transfer of an application. According to the roofline model, an algorithm can be either compute-bound or memory-bound [30], depending on whether the memory bandwidth or the computational performance limits the effectively achievable performance. The computational capability of a core can only be fully utilized if the algorithmic intensity of an application is larger than the core's performance per memory bandwidth.

Fig. 4 shows the roofline performance model of each of the three configurations of Vicuna, along with the effectively achieved performance for three benchmark applications, namely weighted vector addition, matrix multiplication, and the 3 × 3 image convolution. The dashed lines show each configuration's performance boundary, i.e., the maximum theoretical performance in function of arithmetic intensity. The horizontal part of these boundaries corresponds to the compute-bound region, where the throughput of the multipliers limits the performance. The diagonal portion of the performance boundary shows the memory-bound region, where the memory bandwidth limits the performance. Applications with a high arithmetic intensity are compute-bound, while memory-intensive applications with a low arithmetic intensity are memory-bound. Markers indicate the effectively achieved performance for each benchmark program.

The first benchmark is AXPY, a common building block of many Basic Linear Algebra Subroutine (BLAS). AXPY is defined as $Y \leftarrow \alpha X + Y$, where $X$ and $Y$ are two vectors, and $\alpha$ is a scalar. Hence, this algorithm adds the vector $X$ weighted by $\alpha$ to the vector $Y$. We implement AXPY for vectors of 8-bit elements. For a vector of length $n$, it requires $n$ 8-bit MAC operations and $3n$ bytes of memory transfer, which gives the algorithm an arithmetic intensity of $1/3$, thus placing it in the memory-bound region for all three configurations.

**Table 2** Configurations of Vicuna for evaluation on a Xilinx 7 Series FPGA. Note that for larger configurations, the maximum clock frequency decreases slightly as these require more resources which complicates the routing process.

| Config. Name | Configuration Parameters | | | 8-bit MACs per cycle | Clock frequency (MHz) |
|---|---|---|---|---|---|
| | Vector Reg. Width (bit) | Multiplier Data-Path Width (bit) | Data-Cache Size (kB) | | |
| Small | 128 | 32 | 8 | 4 | 100 |
| Medium | 512 | 128 | 64 | 16 | 90 |
| Fast | 2048 | 1024 | 128 | 128 | 80 |

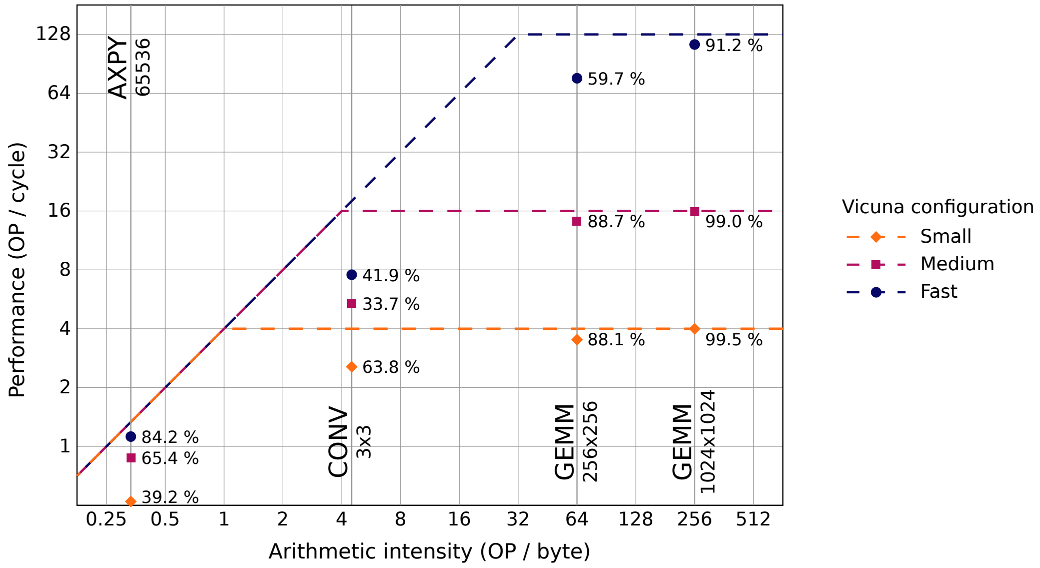


**Figure 4** Roofline plot of the performance results for the benchmark algorithms for each of Vicuna's three configurations listed in Table 2. The dashed lines are the performance boundaries of each configuration, and the markers show the measured effective performance. The percentages indicate the ratio of effective vs. theoretical performance.

The next benchmark program that we consider is the generalized matrix multiplication (GEMM) $C \leftarrow AB + C$, which adds the product of two matrices, $A$ and $B$, to a third matrix, $C$. The arithmetic intensity of this algorithm depends on the size $n \times n$ of the matrices. It requires loading each of the matrices $A$, $B$, and $C$ and storing the result, which corresponds to a minimum of $4n^2$ values that must be transferred between the core and memory. The matrix multiplication itself requires $n^3$ MAC operations. We again use 8-bit values, which gives an arithmetic intensity of $n/4$ MACs per byte transferred. We evaluate Vicuna's performance for two matrix sizes, $256 \times 256$ and $1024 \times 1024$, with an arithmetic intensity of 64 and 256, respectively, which are heavily compute-bound.

Finally, we use the $3 \times 3$ image convolution, which is at the core of many convolutional neural networks (CNNs). This algorithm loads an input image, applies a $3 \times 3$ convolution kernel, and then stores the result back to memory. Hence, each pixel of the image must be transferred through the memory interface twice, once for loading and once for storing. A total of 9 MACs are applied per pixel. Thus the arithmetic intensity is 4.5.

The benchmark programs have been executed on all three configurations of Vicuna, and the execution times were measured with performance counters. Table 3 lists the recorded execution times. For all measurements, both data and instruction caches were initially cleared. The results show that the performance of Vicuna scales almost linearly w.r.t. the maximum throughput of its functional units, which is consistent with the capabilities observed in high-performance vector processors. For highly compute-bound applications, such as the matrix multiplication of size $1024 \times 1024$, the multipliers are utilized over 90 % of the time for the fast configuration and over 99 % of the time for the smaller variants.

The resource usage of Vicuna is similar to that of other FPGA-based vector processors. Fig. 5 shows a radar chart that compares the fast configuration of Vicuna to the VESPA [48] and the VEGAS [6] architectures (we compare configurations that have the same theoretical peak performance of 128 8-bit operations). Other FPGA-based vector architectures, such

**Table 3** Execution time measurements of the benchmark applications for each configuration.

| Benchmark | Execution time in CPU cycles on the respective configuration | | |
|---|---|---|---|
| | Small | Medium | Fast |
| AXPY | 108 985 | 58 693 | 41 989 |
| CONV | 214 486 | 92 852 | 61 719 |
| GEMM 256 × 256 | 4 758 824 | 1 164 797 | 665 596 |
| GEMM 1024 × 1024 | 268 277 942 | 67 467 224 | 9 182 492 |

as VIPERS or VENICE, have only demonstrated smaller configurations and thus are not included in this comparison. While the amount of logic resources consumed by Vicuna is similar to that of the other soft vector processors, its minimum clock period is larger. This is primarily due to the latency of the vector register file's read and write ports. VESPA can only execute one operation at a time and does not support a fused multiply-add instruction, thus requiring much fewer register file ports than Vicuna. VEGAS replaces the vector register file with a scratchpad memory with only two read and write ports. Despite its lower clock frequency, Vicuna achieves a higher effective performance than VESPA and VEGAS because of its ability to execute several operations in parallel, which allows it to better utilize its computational resources. For VEGAS, Chou et al. report an execution time of 4.377 billion cycles for a 4096 × 4096 matrix multiplication on a 32-lane configuration, which corresponds to a multiplier utilization of only 49 %. Vicuna achieves an efficiency of over 90 % for compute-bound workloads.

The efficiency of Vicuna is more in line with recent ASIC-based vector architectures, such as Cavalcante et al.'s Ara [5] and Lee et al.'s Hwacha [25]. Both of these architectures achieve over 90 % utilization of computational units, with Ara reaching close to 98 % for a 256 × 256

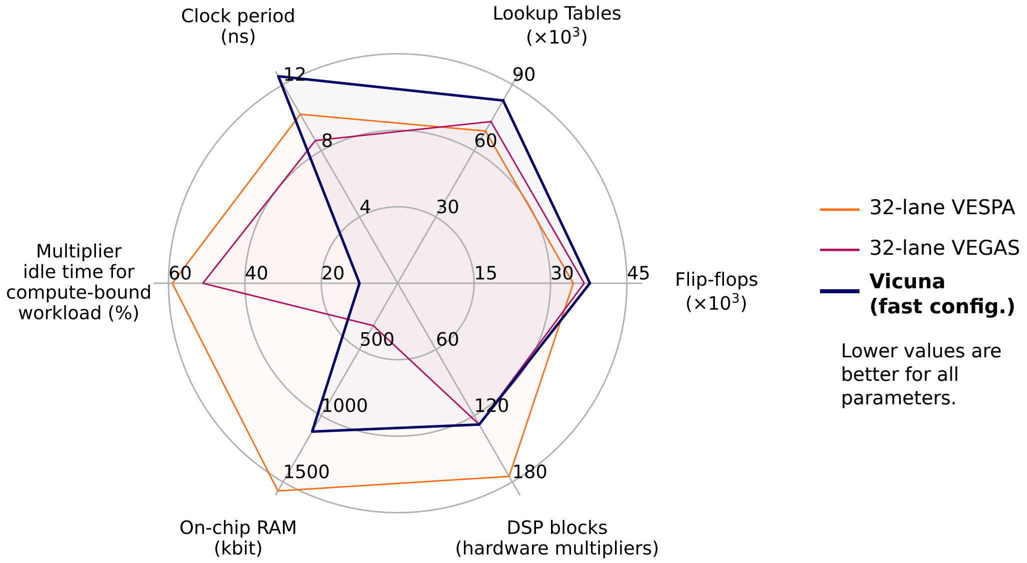


**Figure 5** Resource utilization and performance of the FPGA-based vector processors Vicuna, VESPA, and VEGAS (each configured for a peak performance of 128 8-bit operations per cycle).

matrix multiplication on a configuration with 16 64-bit lanes. Yet, both Ara and Hwacha use features that are a source of timing anomalies. Ara resolves banking conflicts for its banked vector register file dynamically with a weighted round-robin arbiter that prioritizes arithmetic operations over memory operations. Therefore, run-time decisions are involved in the progress of instructions, and slow memory operations can be delayed by subsequent arithmetic instructions. Hence, Ara likely exhibits both counterintuitive and amplification timing anomalies [44]. While Hwacha sequences the accesses of vector register elements in a way that avoids banking conflicts, it uses an out-of-order write-back mechanism and consequently also suffers from timing anomalies. In addition, none of the existing vector processors that we investigated maintains the ordering of memory accesses, particularly when the main core and the vector core both access the same memory. Thus all these architectures are plagued by amplification timing anomalies [16].

A feature distinguishing Vicuna from other vector processors is its timing-predictability and compositionality. Vicuna is free of timing anomalies, enabling compositional timing analysis required for efficient WCET estimation in real-time systems. While the performance figures for Vicuna were obtained via measurements instead of a timing analysis, the predictable nature and low timing variability of Vicuna, as well as the absence of data-dependent control-flow branches in the benchmark programs, implies that their execution time is constant (assuming that the cache is initially idle). Hence, the measured execution times in Table 3 are equal to the respective WCET. Repeating the measurements with varying input data does not alter the timing and always yields the same execution times.

In contrast to timing-predictable multi-core architectures, Vicuna's performance scales significantly better. The performance of multi- and many-core systems typically does not scale linearly with the number of cores since contention on the underlying network connecting these cores to the memory interface becomes a limiting factor [28, 41]. This is particularly true in real-time systems where tasks require guarantees regarding the bandwidth and latency available to them [22, 33]. Schoeberl et al. found that the worst-case performance of the T-CREST platforms scales only logarithmically with the number of cores [38]. Similar results have been reported for the parMERASA multi-core architecture [12]. By contrast, the fast configuration of Vicuna achieves over 90 % multiplier utilization for compute-bound workloads, thus scaling almost linearly with the theoretical peak performance.

The combination of timing-predictability, efficiency, and scalability for parallel workloads makes Vicuna a prime candidate for time-critical data-parallel applications. Besides, Vicuna uses the RISC-V V extension as its instruction set, rather than custom extensions, as do most vector processors, which eases its adoption.

## 6 Conclusion

The performance-enhancing features in modern processor architectures impede their timing-predictability. Therefore, the performance of architectures suited for time-critical systems lags behind processors optimizing for high computational throughput. However, the increasingly demanding tasks in real-time applications require more powerful platforms to handle complex parallel workloads.

In this work, we presented Vicuna, a timing-predictable, efficient, and scalable 32-bit RISC-V vector coprocessor for massively parallel computation. We have integrated Vicuna with the Ibex processor as the main core and demonstrated that the combined processing system is free of timing anomalies, thus enabling compositional timing analysis.

The inherent efficiency of the vector processing paradigm allows us to drop common micro-architectural optimizations that complicate WCET analysis without giving rise to a significant performance loss. Despite its timing predictability, the effective performance of Vicuna scales almost linearly w.r.t. the maximum throughput of its functional units, in line with other high-performance vector processing platforms. Therefore, our vector coprocessor is better suited for time-critical data-parallel computation than the current timing-predictable multi-core architectures.

## References

1 K. Andryc, M. Merchant, and R. Tessier. FlexGrip: A soft GPGPU for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 230–237, December 2013. doi:10.1109/FPT.2013.6718358.

2 Krste Asanovic. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, CA, USA, 1998.

3 Mihail Asavoae, Belgacem Ben Hedia, and Mathieu Jan. Formal Executable Models for Automatic Detection of Timing Anomalies. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2018.2.

4 S. F. Beldianu and S. G. Ziavras. Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Transactions on Computers*, 64(3):805–817, 2015. doi:10.1109/TC.2013.2295820.

5 Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1 GHz+ scalable and energy-efficient RISC-V vector processor with multi-precision floating point support in 22 nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP:1–14, December 2019. doi:10.1109/TVLSI.2019.2950087.

6 Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 15–24, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950413.1950420.

7 Daniel Dabbelt, Colin Schmidt, Eric Love, Howard Mao, Sagar Karandikar, and Krste Asanovic. Vector processors for energy-efficient embedded systems. In *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*, MES '16, page 10–16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2934495.2934497.

8 J. Dean. The deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 8–14, February 2020. doi:10.1109/ISSCC19947.2020.9063049.

9 G. A. Elliott and J. H. Anderson. Real-world constraints of GPUs in real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 2, pages 48–54, 2011. doi:10.1109/RTCSA.2011.46.

10 Glenn A. Elliott and James H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74, 2012. doi:10.1007/s11241-011-9140-y.

11 Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972. doi:10.1109/TC.1972.5009071.

12 Martin Frieb, Ralf Jahr, Haluk Ozaktas, Andreas Hugl, Hans Regler, and Theo Ungerer. A parallelization approach for hard real-time systems and its application on two industrial programs. *Int. J. Parallel Program.*, 44(6):1296–1336, December 2016. doi:10.1007/s10766-016-0432-7.

13 V. Golyanik, M. Nasri, and D. Stricker. Towards scheduling hard real-time image processing tasks on a single GPU. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 4382–4386, 2017. doi:10.1109/ICIP.2017.8297110.

14 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:25, Dagstuhl, Germany, May 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.27.

15 S. Hahn and J. Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–481, 2018. doi:10.1109/RTSS.2018.00060.

16 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 299–308, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2997465.2997471.

17 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. *Toward Compact Abstractions for Processor Pipelines*, pages 205–220. Springer International Publishing, 2015. doi:10.1007/978-3-319-23506-6_14.

18 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

19 R. M. Hord. *The Illiac IV: The First Supercomputer*. Springer-Verlag Berlin Heidelberg GmbH, 1982.

20 M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee. Formal semantics of predictable pipelines: a comparative study. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 103–108, 2020. doi:10.1109/ASP-DAC47756.2020.9045351.

21 Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. doi:10.1145/3140659.3080246.

22 Nassima Kadri and Mouloud Koudil. A survey on fault-tolerant application mapping techniques for network-on-chip. *Journal of Systems Architecture*, 92:39–52, 2019. doi:10.1016/j.sysarc.2018.10.001.

23 Junsung Kim, Ragunathan (Raj) Rajkumar, and Shinpei Kato. Towards adaptive gpu resource management for embedded real-time systems. *SIGBED Rev.*, 10(1):14–17, 2013. doi:10.1145/2492385.2492387.

24 Charles Eric Laforest, Zimo Li, Tristan O'rourke, Ming G. Liu, and J. Gregory Steffan. Composing multi-ported memories on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3), September 2014. doi:10.1145/2629629.

25 Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202, September 2014. doi:10.1109/ESSCIRC.2014.6942056.

26 Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, page 137–146, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1450095.1450117.

27 Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008. doi:10.1109/MM.2008.31.

28 Radu Marculescu, Umit Y. Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(1):3–21, January 2009. doi:10.1109/TCAD.2008.2010691.

29 Tulika Mitra. Time-predictable computing by design: Looking back, looking forward. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3316781.3323489.

30 G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, March 2014. doi:10.1109/ISPASS.2014.6844463.

31 John Owens, Mike Houston, David Luebke, Simon Green, John Stone, and James Phillips. GPU computing. *Proceedings of the IEEE*, 96:879–899, May 2008. doi:10.1109/JPROC.2008.917757.

32 Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch. Sirakoulis, and Giorgos Dimitrakopoulos. RISC-V2: A scalable RISC-V vector processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, September 2020. doi:10.1109/ISCAS45731.2020.9181071.

33 Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Hard real-time application mapping reconfiguration for NoC-based many-core systems. *Real-Time Systems*, 55:433–469, 2019. doi:10.1007/s11241-019-09326-y.

34 RISC-V International. *Working draft of the proposed RISC-V V vector extension*, January 2021. Version 0.10. URL: https://github.com/riscv/riscv-v-spec.

35 Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978. doi:10.1145/359327.359336.

36 S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 220–229, 2015. doi:10.1109/CODESISSS.2015.7331385.

37 P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, September 2017. doi:10.1109/PATMOS.2017.8106976.

38 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. 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.

39 Aaron Severance and Guy Lemieux. VENICE: A compact vector processor for FPGA applications. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–5, 2011. doi:10.1109/HOTCHIPS.2011.7477515.

40 Aaron Severance and Guy Lemieux. Embedded supercomputing in FPGAs with the vectorblox MXP matrix processor. In *2013 International Conference on Hardware/Software Codesign*

*and System Synthesis (CODES+ISSS)*, pages 1–10, 2013. doi:10.1109/CODES-ISSS.2013.6658993.

41 Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Comput. Surv.*, 50(2), 2017. doi:10.1145/3057267.

42 Sudarshan Srinivasan, Pradeep Janedula, Saurabh Dhoble, Sasikanth Avancha, Dipankar Das, Naveen Mellempudi, Bharat Daga, Martin Langhammer, Gregg Baeckler, and Bharat Kaul. High performance scalable FPGA accelerator for deep neural networks, 2019. URL: https://arxiv.org/abs/1908.11809.

43 Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernandez, Francisco Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka. Parallelizing industrial hard real-time applications for the parmerasa multicore. *ACM Trans. Embed. Comput. Syst.*, 15(3), May 2016. doi:10.1145/2910589.

44 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 295–303, 2005. doi:10.1109/QSIC.2005.49.

45 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

46 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.

47 Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. doi:10.1145/1498765.1498785.

48 Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, page 61–70, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1450095.1450107.

49 Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core CPU accelerator. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, page 222–232, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1344671.1344704.

50 M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. Flexpret: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 101–110, 2014. doi:10.1109/RTAS.2014.6925994.

# A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic

Denis Hoornaert[1] ✉
TU München, Germany

Shahin Roozkhosh[1] ✉
Boston University, MA, USA

Renato Mancuso ✉
Boston University, MA, USA

## Abstract

The sharp increase in demand for performance has prompted an explosion in the complexity of modern multi-core embedded systems. This has lead to unprecedented temporal unpredictability concerns in Cyber-Physical Systems (CPS). On-chip integration of programmable logic (PL) alongside a conventional Processing System (PS) in modern Systems-on-Chip (SoC) establishes a genuine compromise between specialization, performance, and reconfigurability. In addition to typical use-cases, it has been shown that the PL can be used to observe, manipulate, and ultimately manage memory traffic generated by a traditional multi-core processor.

This paper explores the possibility of PL-aided memory scheduling by proposing a Scheduler In-the-Middle (SchIM). We demonstrate that the SchIM enables transaction-level control over the main memory traffic generated by a set of embedded cores. Focusing on extensibility and reconfigurability, we put forward a SchIM design covering two main objectives. First, to provide a safe playground to test innovative memory scheduling mechanisms; and second, to establish a transition path from software-based memory regulation to provably correct hardware-enforced memory scheduling. We evaluate our design through a full-system implementation on a commercial PS-PL platform using synthetic and real-world benchmarks.

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

**Keywords and phrases** Memory Scheduling, PLIM, FPGA, Memory Management, Bandwidth Regulation, MemGuard, Coloring, Bank Partitioning, Real-time, Multicore, Safety-critical

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.2

**Supplementary Material** The SchIM sources are available at
*Software*: `https://github.com/denishoornaert/MemorEDF`

**Funding** *Denis Hoornaert*: Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.
*Renato Mancuso*: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. 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

It is undeniable that the massive increase in expectation on the performance of next-generation cyber-physical systems has deeply impacted the way we design modern embedded and real-time systems. High-resolution, high-bandwidth sensors such as lidars, and depth cameras on the one hand, and data-intensive processing workload such as machine-learning applications

[1] These authors contributed equally.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 2; pp. 2:1–2:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

on the other hand, have exacerbated the push for high-performance embedded platforms. Following this performance *moving target*, chip manufactures have significantly scaled up clock speeds, CPU count, and heterogeneity. For instance, the on-chip integration of powerful graphic processing units (GPUs) has been the characterizing factor in the NVIDIA Tegra series of embedded systems-on-a-chip (SoC).

In this context, an embedded architectural paradigm that is surging in popularity among manufacturers, researchers, and industry practitioners is the PS-PL organization. This class of embedded platforms integrates on the same die (1) traditional full-speed embedded CPUs and (2) programmable logic constructed using field-programmable gate array (FPGA) technology. This organization naturally defines two macro-domains, namely the Processing System (PS) and the Programmable Logic (PL), hence the name. PS-PL platforms establish a good trade-off between specialization, raw performance, and mission-specific re-configurability. The current generation of commercially available PS-PL platforms is dominated by ARM-based products offered by, most notably, Intel [12] and Xilinx [37]. A pilot large-scale, high-performance PS-PL system is the Enzian platform [3] being rolled out by ETH Zurich[2]. Furthermore, a RISC-V-based solution has been recently made available by Microsemi with their PolarFire SoC [18].

From a real-time perspective, the co-existence of traditional CPUs and a tightly-coupled block of PL has more profound implications than expected. Clearly, it is possible to define custom accelerators in PL and to relieve the main CPUs of some of the heavy data-processing workload. However, more interestingly, recent studies have highlighted the possibility of using the PL also as a way to manage the memory traffic originated from the main CPUs [13, 29]. Such a possibility opens the doors for memory traffic inspection and control at the level of individual transactions; which in turn promises to unlock provable determinism for the real-time workload.

In this paper, we embrace the concept of PL-aided memory traffic management and propose an infrastructure to develop, test and evaluate memory scheduling policies. Specifically, we propose a component, called the Scheduler In-the-Middle– or SchIM, for short – that can be instantiated in the PL to enforce a set of configurable scheduling policies on individual memory transactions generated by the CPUs in the PS.

The overarching goal of the proposed SchIM is twofold. First, we want to provide a playground for researches to test promising novel memory scheduling ideas for multi-core platforms, much like LITMUS$^{\mathrm{RT}}$ [7] fostered research on CPU scheduling techniques. Second, we want our SchIM to act as an intermediate stepping stone for industrial applications where strong determinism over memory performance is required. The SchIM can be used to analyze the behavior of realistic workload in a multitude of what-if memory management use-cases. We note that such kind of analysis was previously possible only through full-system simulation or by synthesizing the entire SoC on FPGA – that is, with a soft-core implementation.

In short, this paper makes the following contributions. (1) We demonstrate that a configurable module could be interposed between the cores and the memory controller to perform transaction-level scheduling in commercial PS-PL platforms; (2) we propose a design for a memory scheduling infrastructure that focuses on extensibility and runtime reconfigurability; (3) we address important issues to correctly account and regulate CPU-generated traffic when a shared last-level cache is present; (4) we design and implement two pilot memory scheduling policies as a proof-of-concept on the potential of our SchIM; and (5) we perform a full system integration and implementation on a commercial PS-PL embedded platform to evaluate the behavior of the SchIM with synthetic and realistic workload.

[2] Also see `http://enzian.systems/`

## 2 Related Work

There is a broad consensus that memory resources represent the main performance bottleneck in modern multi-core processors. The observation has sparked a host of research works addressing the problem from multiple angles [17]. In this context, the works representing the inspiration for our SchIM fall in two macro-categories, namely **hardware-based** and **software-based** techniques for main memory traffic management.

The first category includes a large body of works aimed at achieving better and/or more predictable performance by advancing novel hardware redesigns. The works in [22–24] strive to construct high-performance and fair memory schedulers. The addition of software-controlled memory deadlines and transactional semantics where explored in [33] and [10], respectively. Next, the work by Åkesson et al. [1, 2] and Paolieri et al. [25] attains timing predictability through careful scheduling of SDRAM commands. Finally, the MEDUSA DRAM controller [9, 34] implements a two-tiers scheduler at the DRAM controller to ensure predictability when accessing memory areas where access time strongly impact application performance. Finally, the hardware designs proposed in [8, 26, 42] put their emphasis on main memory bandwidth partitioning; clever dynamic pipelining is further explored in [20] to better balance average performance and determinism.

Among the software-based techniques are the mechanisms that stemmed from MemGuard, originally proposed in [41] and that rely on broadly available performance counters to regulate the bandwidth extracted by individual CPUs. Later extensions to jointly consider regulation and cache partitioning [38] and to expose control over memory bandwidth as a lockable resource [39] were proposed. Software-based memory throttling has also been implemented at the hypervisor-level [21, 30]. Remarkably, the work in [30] combines regulation mechanisms for CPU and embedded accelerators through the ARM QoS extensions [4].

In addition to the two categories surveyed above, perhaps the most closely related works are those that explored memory isolation techniques in PS-PL platforms. The work in [11] demonstrated that the PL-side can be used to define private memory storage, control, and bus units to strongly isolate high-criticality workload. A number of techniques developed as part of the FRED framework [6] put an emphasis on memory traffic arbitration and management for in-PL accelerators [27, 28]. The AXI HyperConnect [27] is perhaps the component most similar to the SchIM in terms of high-level design. However, both are substantially different as the SchIM is designed to manage embedded CPUs' memory traffic.

Compared to the literature reviewed above, what sets this work apart are the following aspects. (1) Our SchIM applies to existing PS-PL commercial systems without introducing any hardware modification; (2) it allows management in the PL of memory traffic originated by the embedded CPUs residing in the PS; (3) it provides the framework to test the feasibility and performance of custom memory scheduling policies; and (4) it is designed such that multiple schedulers can coexist, be activated, and configured at runtime.

## 3 Background Concepts

In this section, we introduce some fundamental concepts necessary to understand the overall system design and the class of platforms targeted by this work.

### 3.1 Hybrid Multi-Core Platforms with Programmable Logic

This work targets the aforementioned class of embedded multi-core platforms with programmable logic – i.e., PS-PL platforms. In such platforms, the PS encompasses a multi-core processor with a multi-level cache hierarchy and a main memory (DRAM) controller. A

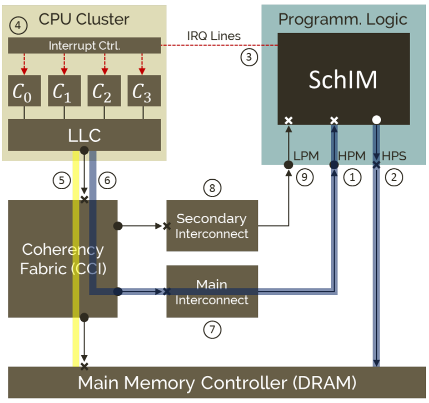


**Figure 1** PS-PL interconnect block diagram.

simplified block diagram for a reference PS-PL organization is illustrated in Fig. 1. The figure considers a platform with four CPUs denoted as $C_0, C_1, C_2$, and $C_3$.

A key feature in PS-PL platforms is the presence of high-performance communication channels between the two domains. These come in the form of data exchange interfaces and interrupt lines. Data exchange channels follow a master-slave paradigm. Specifically, high-performance masters (HPM, Fig. 1①) and high-performance slaves (HPS, Fig. 1②) send and receive transactions to and from the PL, respectively. Additionally, there exist programmable interrupt request (IRQ) lines (see Fig. 1③) that can be driven by the PL and are connected to the interrupt controller (Fig. 1④) inside the PS. As we discuss in Section 5.7, the presence of PS-PL interrupt lines is crucial to building PL-assisted memory traffic regulation.

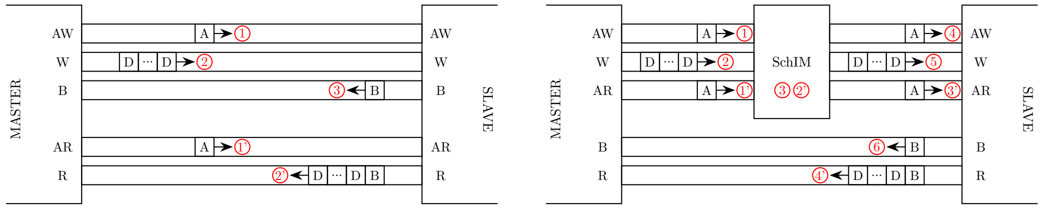
Note also that there might exist PS-PL data ports that are routed through a secondary interconnect (Fig. 1⑧). These can generally sustain less throughput compared to HPS ports; hence we refer to them as low-performance masters (LPM, Fig. 1⑨). LPM ports are useful to perform memory-mapped configuration of PL modules.

## 3.2 Programmable Logic In-the-Middle

In this work, we leverage the ability to route main memory traffic originated by the CPUs through the PL. This technique is known as Programmable Logic In-the-Middle, or PLIM for short. PLIM was originally proposed in [29]. To fully grasp how PLIM can be achieved, one needs to understand how memory accesses are routed in PS-PL platforms.

Any CPU-generated memory access that results in an LLC miss is routed directly to main memory if its physical address falls within the aperture, say the address range $[A, B]$ handled by the DRAM controller. We refer to this as the *normal route*, depicted in Fig. 1⑤ and highlighted in yellow.

Conversely, generic memory access resulting from an LLC cache miss will be sent on an HPM port if the corresponding physical address falls within another range, say $[C, D]$. One can then insert (1) a lightweight layer of virtualization to map all the physical addresses of a guest OS to the PL, i.e., to fall in the range $[C, D]$; and (2) an address translator in the PL that re-bases request physical addresses to access main memory and relays back the data payload to the requesting CPU(s). In other words, one can find a constant $k$ such that $C = A + k$. Then, the translator in the PL, upon receiving any request at address $x \in [C, D]$ will issue a main memory request at the address $(x - k)$ through the HPS port and provide



**(a)** Standard AXI communication Scheme.

**(b)** AXI communication mediated with SchIM.

the response to the CPU. The PLIM technique introduces a secondary memory route for reaching the DRAM, called the *PL loop-back*, or simply *loop-back*, which is highlighted in blue in Fig. 1 ⑥. Memory transactions on the loop-back route typically traverse the main interconnect, as depicted in Fig. 1 ⑦. The advantage of PLIM is that transactions on the loop-back route can be inspected, blocked, re-routed, and in general managed by custom re-programmable logic. Importantly, switching from the direct to the loop-back route can be done dynamically at runtime so that the overhead of PLIM can be avoided if deemed detrimental for the application under analysis.

In this paper, we leverage the PLIM approach to perform memory scheduling, hence, we call our module the Scheduler In-the-Middle, or SchIM for short.

## 3.3 Advanced eXtensible Interface (AXI)

The vast majority of PS-PL platforms currently available are ARM-based. This is also the case for the platform we used for our evaluation, namely the Xilinx Zynq UltraScale+ MPSoC. Thus, we briefly introduce the communication protocol used for on-chip communication in ARM-based SoCs, namely the Advanced eXtensible Interface (AXI). The AXI is an open specification bus protocol [5] used for high-bandwidth data exchanges between on-chip subsystems – such as cache controllers, memory controllers, DMAs, PL modules. It is also used in the PS-PL platforms of reference to exchange data on the HPM and HPS ports.

The AXI protocol is based on the master-slave duality. A master AXI interface can initiate transactions toward a connected slave interface. The latter responds master-initiated requests. Masters and the slaves communicate with each other through five different channels named AW (address write), W (write), B (write acknowledgment), AR (address read) and R (read), as illustrated in Fig. 2a.

A write transaction begins with an address phase ① where the channel AW is used to transmit the transaction’s meta-data, such as the destination address, the transaction ID, and the cacheability attributes the type/length of the burst, and so on. Upon completing this phase, follows the data phase ②, which consists of the transmission of the data payload to be written through the W channel. The response phase ③ concludes a successful write transaction and occurs on the B channel.

The transmission of a read transaction is carried out in a similar way. The address phase ①’ is transmitted through the equivalent AR channel and is directly followed by the data phase ②’. A response initiated by the slave follows where the read data is transferred over the R channel. The protocol is asynchronous because different phases of different transactions can interleave on any AXI bus segment. Hence, multiple outstanding transactions can be emitted by a single master and the receipt of out-of-order responses is possible.

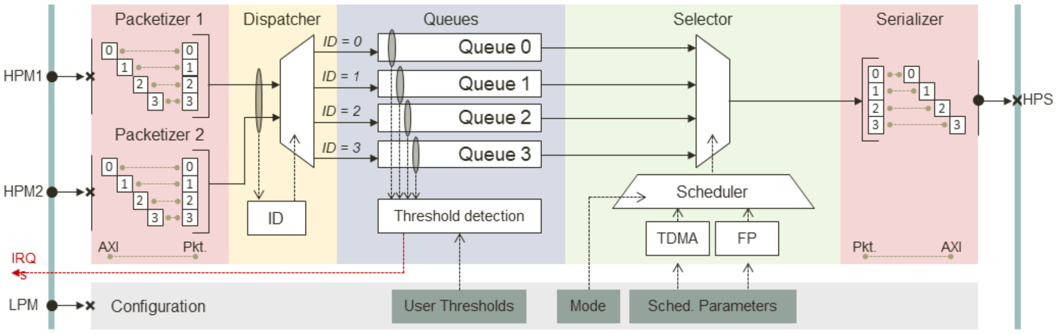


**Figure 3** SchIM internal organization connected to the PS via the HPM, LPM and HPS ports.

## 4 Design Goals and Overview

In this section, we introduce the proposed SchIM design and describe the overarching goals of this work. We then provide a bird's-eye view of the SchIM organization and principles of operation.

### 4.1 Design Goals

As briefly surveyed in Section 2, there have been numerous proposals for better memory controllers and approaches to manage memory traffic in modern multi-core embedded platforms. With respect to the existing literature, the purpose of this work is twofold. First, we want to demonstrate that scheduling CPU-originated memory traffic at the granularity of individual transactions is possible in PS-PL platforms. Second, and more importantly, we want to provide an infrastructure that is generic and extensible enough for the broader research community to adopt and foster a new chapter on PL-assisted memory scheduling. With this in mind, we establish the following goals.

**Extensible memory scheduling infrastructure.** First and foremost, the SchIM has been designed with modularity and extensibility in mind. We separate the functionalities that concern handling, queuing, selection, and forwarding of memory requests inside our infrastructure. Moreover, we design our SchIM to be able to support multiple memory scheduling policies simultaneously. A simple, standardized interface is provided to define new memory scheduling policies without impacting the design of the rest of the SchIM. We discuss in Section 5.5 the generic interface provided by the SchIM to implement a new memory scheduling policy.

**Runtime configuration and transparency.** We want the SchIM to be a robust supporting infrastructure to evaluate, compare, and contrast memory scheduling policies. As such, we strive to provide (1) runtime reconfigurability and (2) operational transparency. It is possible to rapidly identify desirable configuration parameters by allowing memory scheduling policies to be switched at runtime. Besides, an adopted policy can be tuned according to the workload criticality and memory intensiveness. For this purpose, the SchIM exposes a memory-mapped configuration interface where all the operational parameters can be changed at runtime. At the same time, we want to ensure that the applications and the (real-time) operating system under analysis do not need to be modified to use the SchIM. Hence, we propose using a thin virtualization layer to selectively route memory traffic through the SchIM without changes to the binary of OS kernel and applications.

**Realistic performance with experimental policies.** One of the limiting factors of research on memory scheduling policies is the ability to construct evidence of performance improvements with the realistic workload. Proposing a new memory scheduling policy is traditionally done with either a simulated setup or with a full-system soft-core implementation. Both cases have their drawbacks. The former gives a great deal of flexibility but achieving clock-level accuracy requires simulating many SoC components whose details might not be publicly available. In addition, simulated setups that propose custom hardware designs cannot be directly adopted on real platforms without being first synthesized in hardware. Full soft-core-based SoC implementations suffer from two shortcomings. First, they run at relatively low frequencies and thus can extract only a fraction of the available DRAM bandwidth. Secondly, they are typically based on processor IPs that do not feature the same Instructions Set Architecture (ISA) as widely available COTS, which further limits the practical impacts of these works.

As reported in [29], re-routing the traffic of the core cluster through the PL-side comes at a cost in terms of extra latency and reduced bandwidth. Nonetheless, as PS-PL platforms mature and the interplay of PL and memory resources improves, a SchIM-like design could be the way to go for mission-reconfigurable, upgradable embedded systems.

## 4.2 Design Overview

As previously mentioned, the SchIM leverages the PLIM approach. CPU-originated main memory transactions are re-routed through the programmable logic and scheduled by the SchIM according to a flexible and configurable policy. The result is that the timing of memory transactions generated by real-time applications can be carefully determined and reasoned upon. Because the SchIM follows a PLIM approach, transactions can be selectively sent to the SchIM for scheduling. However, it is always possible to dynamically exclude the SchIM and route transactions directly to the main memory. Toward this paper's incentive, we consider a setup in which SchIM handles all the CPU-generated memory transactions.

Fig. 1 provides an overview of the location of the SchIM in the reference platform, while its internal organization is visible in Fig. 3. Application memory requests reach the SchIM the aforementioned HPM ports. Without loss of generality, we consider a SchIM instance with two arrival lanes, which are labeled as $\texttt{HPM}_1$ and $\texttt{HPM}_2$ in Fig. 3. The SchIM then forwards the received transactions towards main memory through the `HPS` interface. A more detailed view of the SchIM module is provided in Fig. 3 where the same convention is used to identify input and output ports. In addition, as shown in Fig. 3, a fourth `LPM` port is used to configure the SchIM from the PS.

The SchIM is composed of a number of sub-modules grouped into three different domains, namely (i) the *interfacing domain*, (ii) the *queuing domain*, and (iii) the *scheduling domain*.

**The interfacing domain** encompasses the sub-modules to interface the core logic of the SchIM with the rest of the system using the AXI protocol. This is comprised of three sub-modules. These are (i) the *packetizer(s)*, (ii) the *serializer*, and (iii) the previously mentioned *configuration* interface.

The PS-facing end of the **packetizer** offers an AXI slave port to accept new incoming transactions. Upon receipt, this module transforms each transaction into an equivalent *packet* that can be queued and scheduled by SchIM. Packetization of AXI transactions is necessary to be able to store transactions that are serial by nature. A standard AXI transaction is composed of one address phase (AR or AW channel) followed by a data phase (R or W channel), which can be itself composed of multiple successive bursts.

In many ways, the **serializer** is the dual module of the packetizer. Its purpose is to transform the packets that encode CPU-generated memory requests back into AXI-compliant transactions. As such, the serializer offers a master port to the rest of the system to be routed to the main memory controller.

**The queuing domain** handles how packets are stored between receipt and retrasnmission. This domain is comprised of (i) the *dispatcher* module, (ii) the *transaction queues*, and (iii) the *selector* module.

The use of **multiple transaction queues** is necessary to differentiate the traffic of the CPUs and perform scheduling. As such, the SchIM associates a queue to each of the active cores – four in the platform of reference. The queues implemented in the SchIM not only act as a holding space for in-flight memory transactions. They also (a) provide information to the scheduling domain regarding their current state, and (b) they can generate a congestion control signal to the associated CPU core.

Congestion control is vital because memory transactions originated at the LLC controller follow the same route to the SchIM regardless of the originating CPU. The total number of outstanding transactions that the cores can emit exceeds the queuing elements' capacity on the loop-back route. Hence, priority inversion arises if a low-priority CPU's memory traffic is (temporarily) held. Latter is due to the uncontrolled queue buildup, which provokes a head-of-line blockage. Importantly, what described is true also for the normal route and it is a direct consequence of the best-effort nature of traditional multi-core memory buses. The SchIM allows the user to specify a configurable threshold on the occupancy of the queues that, when reached, issues a regulation signal to the corresponding CPU. We describe in greater detail how congestion control was implemented on the target platform in Section 5.7.

As suggested by Fig. 3, transactions are categorized and enqueued based on the source of traffic. The **dispatcher** module performs the matching between an incoming transaction and the destination queue. Similarly, transactions are dequeued by the **selector** module and sent directly to the output of the SchIM following the scheduling domain's resolutions.

**The scheduling domain** encompasses all the sub-modules that enable arbitration of transactions issued by the different cores of the PS. The modules in this domain are intended to be generic for extensibility, albeit the first set of two template schedulers is provided as a proof of concept. The scheduling policies currently implemented in the SchIM are Fixed Priority (FP) and Time Division Multiple Access (TDMA). Each of the parameters required by the implemented policies – such as the priorities and the periods – can be adjusted at runtime via the configuration interface.

The FP scheduler allows associating a priority value to each of the transaction queues. Pending transactions at the queues are then forwarded out of the SchIM following the user-defined priority order. The TDMA scheduler allows associating a transmission time slot to each of the queues expressed in PL clock cycles. The module then builds a schedule by concatenating the per-core slots so that only pending transactions from one queue at a time are forwarded by the SchIM.

## 5 SchIM Design and Implementation

A full-system implementation was carried out on a Xilinx ZCU102 development system, which is based on a Xilinx Zynq UltraScale+ XCZU9EG PS-PL SoC. The PS comprises four ARM Cortex-A53 CPUs that share a unified 1 MB LLC. The PS includes a DDR4-2666 controller connected to a 4 GB DDR4 memory module. There are two high-performance master interfaces (HPM1 and HPM2); and a third interface routed through the low power domain (LPM). The PL is capable of driving up to 16 interrupt requests lines towards the PS interrupt controller. We hereby provide key details on the operation of our SchIM in the target platform. These include complementary software stack, memory traffic accounting, regulation to prevent head-of-line blocking, and programming model.

## 5.1 Software Stack

As mentioned in Section 4.1, we want to ensure that the SchIM can be used with no modification to the OS and the applications under analysis. For this reason, we rely on a thin virtualization layer that can be used to redirect memory traffic from the direct route to the loop-back route (see Section 3.2). For this purpose, we use the open-source Jailhouse [16] partitioning hypervisor[3] Jailhouse does not boot the target machine. Instead, it relies on a standard Linux kernel to perform the initial boot sequence. When enabled from a Linux driver, Jailhouse dynamically virtualizes the original OS. In line with its partitioning-only philosophy, Jailhouse has a small footprint and enforces virtualization-aided partitioning of essential resources like CPUs, interrupts, main memory, I/O devices. It does not perform any virtual-CPU scheduling.

Following Jailhouse's nomenclature, a resource partition is called a *cell*, while guest OS's are referred to as *inmates*. An inmate can be either a bare-metal application, an RTOS or a full-fledged OS like Linux. Jailhouse uses ARM hardware Virtualization Extensions (VE) to offer a set of Intermediate Physical Address (IPA) to its inmates that is compatible with the way they have been compiled. Jailhouse then maps IPA ranges of different cells to configurable Physical Addresses (PAs) – stage-2 translation. By changing the configured stage-2 mapping, it is possible to dynamically re-route via the loop-back the memory traffic generated by each inmate.

As described below, some modifications were necessary to the mainline Jailhouse code for our full system implementation[4].

## 5.2 Altered communication scheme

In order to achieve the objective of re-ordering transactions, one must alter the standard AXI communication scheme explained in the Section 3.3. To this end, the SchIM is interposed between the master (HPM) and the slave (HPS) as depicted in Fig. 2b. As shown in Fig. 2b, only the phases initiated by the masters (i.e., address phase on AW and AR and the data phase on W) are intercepted for re-ordering by the SchIM. The introduction of the SchIM has a direct consequence on the overall communication scheme. Unlike the response phases on channels R and B that remain unchanged, the address and write data phases are handled following a store-and-forward scheme. Consequently, a write transaction will start exactly as in the standard AXI scheme with its address phase ① and data phase ②. These two transactions are buffered within the SchIM's queues (③) and only relayed following the internal memory scheduler's logic. This release of the transaction leads to the initialization of two new addresses and data phase ④, and ⑤. Finally, the response phase ⑥ goes directly from the slave to the master without being intercepted. For read transactions, the same modifications apply to the address phase ①' which is buffered (②') for some time before being re-emitted in ③'. Just like for write acknowledgments writing, the read response phase ④' is not intercepted by the SchIM.

## 5.3 Queueing Domain

At the heart of the queueing domain, lies the queues. They work as FIFOs. However, instead of inserting the new data at the back of the queue, the new data is always inserted as close as possible to the front of the queue. This mechanism helps avoiding gaps within the queues prevents the loss of few clock cycles that would be required to move the data from the back to the front. From the authors' experiments, saving clock cycles in SchIM is vital to keep the final bandwidth as high as possible.

[3] The source code is available at `https://github.com/siemens/jailhouse.git`.

[4] The modified Jailhouse sources are available at `https://github.com/rntmancuso/jailhouse-rt`.

Furthermore, the queues have been designed to deal with three constraints. Firstly, the queues store both read and write packets such that the order at which transactions arrived is guaranteed. This implies that all the queue slots have the same size regardless of whether they contain read or write packets. Secondly, due to the altered communication scheme (see Section 5.2), each slot needs to be large enough to store both the address phase payload and the corresponding data of an AXI write transaction (678 bits). The depth of each queue is determined by considering the worst-case scenario. The latter consists of having to handle the maximum number of outstanding read and write transactions simultaneously. Our SchIM instance on the considered Xilinx UltraScale+ platform was configured with queues that are 16 slots in-depth. Indeed, the HPM ports in this platform cannot handle more than eight transactions of each type [37].

## 5.4 LLC-SchIM Interface and Traffic Accounting

As illustrated in Fig. 1, the considered system features an LLC shared between the four cores of the PS. For a non-cacheable read (resp., write) memory access, which CPU represents the source of the traffic is carried in the ID bits of the corresponding AR (resp., AW) AXI transaction. But for cacheable memory accesses, which is the norm for application workload, this is not the case. This is mainly because cache controllers typically use a write-back strategy. In this case, a read or write cache miss causes up to two events: (1) a cache refill and (2) a cache eviction. The cache refill is carried out with a read AXI transaction. If the line being evicted was previously written (dirty), then the eviction causes a write AXI transaction. It follows that, while read AXI transactions have an easily identifiable source, write transactions do not. Indeed, a CPU $x$ might be causing the eviction of a line previously allocated and modified by CPU $y$. Hence, accounting (and scheduling) the resulting write transaction as if it originated from CPU $x$ would be incorrect.

To ensure fair accounting for both read and write traffic, we rely on cache partitioning through coloring. As studied in a number of previous works, cache coloring is easy to implement at the hypervisor level [15, 21, 32]. In our system setup, we leverage the support Jailhouse already provides. The standard support has been extended to support booting a Linux inmate over colored memory. Cache partitioning allows us to establish a 1-to-1 relationship between any read/write transaction traversing the SchIM and the originating CPU. Moreover, with cache coloring in place, the SchIM uses the color bits in the address of the memory transactions (AR and AW channels) – instead of the AXI ID bits – to differentiate between the traffic of the various cores.

Finally, recall that the SchIM forwards transactions between HPM and HPS ports. These ports follow the asynchronous AXI protocol that allows issuing multiple outstanding AR and AW transactions. The protocol dictates that any outstanding transaction must have a unique AXI ID. This property is crucial to be able to match received responses with outstanding requests. Unfortunately, a potential mismatch between the bit-width of the AXI ID emitted at the HPM ports and the bit-width of AXI ID accepted by the HPS ports. For instance, in the platform of reference, the HPMs emit 16-bit AXI IDs, while the HPS AXI ID bit-width is 6 bits. Therefore, the SchIM also acts as an AXI ID translator.

## 5.5 Scheduling Interface and Implemented Policies

All the memory schedulers included in the scheduling domain share a common interface to ease the integration of a new scheduler. In terms of input signals, a generic scheduler module must define (1) a manual reset signal that can be triggered through the configuration port; (2) a vector of bits where each bit indicates whether the associated queue is empty; and (3) a signal indicating if the last scheduled transaction as been consumed. Alongside these inputs, the scheduling modules also have access to all the configuration registers listed in Table 1. In terms of outputs a SchIM scheduler must define (1) a signal to the selector indicating the queue considered for scheduling; and (2) a signal stating whether the current scheduling decision is valid. We hereby review the initial set of memory scheduling policies implemented in the SchIM.

### 5.5.1 Fixed Priority

The FP scheduling module aims at enforcing strict prioritization of cores' memory traffic. The priority ordering is explicitly defined by the user through the configuration port. While the SchIM instance used in this paper only has four queues, 16 different levels of priority are offered as the considered platform supports up to 16 different colors. This is useful if an hypervisor that supports vCPU scheduling is used. In this case, the SchIM allows assigning different priorities to different partitions sharing the same physical CPU. The core-to-priority assignment must be strict, meaning that two cores cannot be assigned the same priority.

The FP scheduling module only needs two pieces of information. That is (1) the priority associated with each queue and (2) whether a given queue contains at least one buffered transaction. The module logic always selects the queue with the highest priority. Lower priority queues are considered when higher priority queues do not have transactions. This is done by internally setting the user-defined priority of a queue as 0 when the corresponding queue is empty.

### 5.5.2 Time Division Multiple Access

The TDMA memory scheduler is a non-work conserving policy that operates by defining a per-core time *slot* during which the core has exclusive access to main memory. The slots are expressed in PL clock cycles, to maximize granularity. The configuration port can be used to specify and change the slots specifications at runtime.

The implementation of the module uses a counter register to track the time elapsed in the current TDMA primary frame – defined as the sum of all the cores' slots. It is reset to 0 at the beginning of a new major frame. Using the time-tracking register, the module determines to which core the current slot belongs, and forwards the information to the queue selector. This is done by summing up the length of all the previous slots, and determining if the current time falls within the interval of the considered core's slot.

## 5.6 Programming Model

The parameters that compose the programming interface of the SchIM are summarized in Table 1. The `base` address referenced in the table can be set when the SchIM is deployed in the PL. By default, this is set to `0x800000000`. All the parameter registers are 32 bit wide, except for the priorities of the FP scheduler. In this case, the priority values are encoded using 8 bits. The last "Mode" register allows a user to select the active memory scheduler.

## 5.7 PL-to-PS Feedback

Each of the HPM ports interfacing the PS and the PL sides (HPM1 and HPM2) have two dedicated queues for read and write transactions. Since transactions are being buffered inside SchIM as well as in these port buffers, head-of-line blocking can happen. Head-of-the-line blocking is harmful for performance; and can cancel out the benefits of transaction scheduling performed by the SchIM. For instance, in the case of a non work-conserving policy (e.g.,

**Table 1** Available SchIM configuration registers.

| Parameter | Associated Core | Address |
|---|---|---|
| TDMA slots | $C_0$ | `base+0x00` |
| | $C_1$ | `base+0x04` |
| | $C_2$ | `base+0x08` |
| | $C_3$ | `base+0x0C` |
| User Thresholds | $C_0$ | `base+0x10` |
| | $C_1$ | `base+0x14` |
| | $C_2$ | `base+0x18` |
| | $C_3$ | `base+0x1C` |
| FP Priorities | $C_0$ \| $C_1$ \| $C_2$ \| $C_3$ | `base+0x20` |
| Reserved | | |
| Mode | N/A | `base+0x38` |

TDMA), if the HPM port queue gets filled with transaction coming for the same core, no other transaction will be able to reach the SchIM and thus be considered for scheduling. This implies that no transaction would be scheduled until the end of the active core's TDMA slot. On the other hand, for work-conserving policies (e.g., FP) in the presence of head-of-line blocking, the decisions being taken by SchIM would directly depend on the order at which transactions are emitted by the HPM port buffer.

In both cases, one must prevent the cores from saturating the HPM port buffers. In order to avoid such situation, we implemented a feedback scheme aimed at slowing down the cores when necessary. As we mentioned in the context of Fig. 3, the SchIM's queues are associated a programmable threshold. Whenever the queue occupancy reaches (or exceeds) the associated threshold, a per-core interrupt line is asserted from the PL to the PS side. When received, the interrupt is treated by the platform software as a *fast interrupt request* (FIQ) and directly handled by the hypervisor – invisible to any guest OS. The advantage of using FIQs instead of regular IRQs is the significantly reduced handling latency [31]. Minor modifications to the TrustZone monitor were necessary to correctly configure FIQ handling. To minimize overhead, the installed FIQ handler only executes two assembly instructions. These are (1) a `dsb` memory barrier that stops the core until all the outstanding memory transactions have been completed, and (2) a `eret` instruction to exit the FIQ context. There is not need to save/restore any register because FIQs have banked syndrome/status registers and because no general purpose register is modified in the handler.

Ideally, the available space in the HPM buffers should be shared evenly between the cores. Since each HPM port has a buffer with a depth of 8+8 transactions, each core should occupy at most 2 slots in each buffer. Unfortunately, our experiments highlighted that the control over amount of transactions buffered by each core is imperfect. Often times, the selected threshold is exceeded by up to two transactions. This is the main reason why we propose a dual-ported SchIM which uses both the available HPM ports. Indeed, by assigning two cores on each of the ports, the ideal threshold on maximum amount buffered transactions can be doubled. The increase provides enough room to compensate for imperfections in the micro-regulation performed with PL-to-PS FIQ delivery.

# 6 Evaluation

The present section aims at evaluating the behavior of the SchIM on the target platform, its overhead and benefits. First, in subsection 6.1, we review our experimental setup. Thereafter, we assess the overhead introduced by the SchIM in Section 6.2. Section 6.3 explores the impact of the PL-to-PS feedback on the control and the performance. In Section 6.4, an in-depth analysis of the SchIM's behavior is presented. Finally, an evaluation of the temporal behavior of a set of real-world benchmarks operating through the SchIM is provided in Section 6.5.

## 6.1 Experimental Setup

The SchIM has been evaluated using synthetic benchmarks (or *Memory Bombs*), real benchmarks selected from the San Diego Vision Benchmark Suite (SD-VBS) [35] and a combination of the two. Specifically, seven memory-intensive benchmarks have been selected, i.e. *stitch*, *texture synthesis*, *disparity*, *tracking*, *localization*, *mser* and *sift*. For our runs, we have considered all the intermediate input sizes ranging from SQCIF (128×196 pixels) to VGA (640×480 pixels). When running any benchmark, we use the cache coloring mechanism implemented in the Jailhouse hypervisor [32] to partition the LLC evenly amongst the 4 cores and to prevent our measurements from being affected by inter-core cache interference. As a result, each benchmark operates on 1/4 of the total cache space – 256 KB. As extensively discussed in [14, 40], it is also important to avoid inter-core DRAM bank conflicts, which can cause the arbitrary re-ordering of transactions originating from different cores. This is accomplished by (1) configuring the DRAM controller to disable DRAM bank interleaving; and (2) by performing static cache bleaching [11, 29] at the SchIM's output to re-compact accesses to colored pages into contiguous DRAM accesses. In this platform, there are a total of 16 DRAM banks of 256 MB each. Thanks to bleaching, we can assign the full size of 4 banks (i.e., 1 GB) to each core, instead of being restricted to only 1/4 of that due to non-overlapping color and bank address bits.

To evaluate the capabilities of the SchIM, two memory routes for the traffic generated by the cores are compared. The first serves as baselines, whereas, the last one is the one under analysis and involves the SchIM module. The first path consists in the cores directly accessing the main memory. As illustrated in Fig. 1, the traffic simply goes through the *Main Interconnect* before arriving at the DDR controller. This path is referred to as the *normal route*. Secondly, we consider the case where the SchIM module is deployed and in use to schedule memory traffic generated by the CPUs in the PL. Cores 0 and 1 target HPM1 aperture, while cores 2 and 3 target HPM2. In our analysis, the SchIM is used in all the available modes, i.e., FP and TDMA.

Note that in the case of the *normal route*, combining both a strict cache partitioning and strict bank partitioning could not be applied. In fact, as a direct consequence of the address coloring and in the absence of a bleacher, only 1/16 of each 1 GB wide memory allocation can be used by each core. The resulting reduced space of 64 MB is not enough for running Linux. Consequently, in the case of the *normal route*, the cores have been split into two groups of two, where each group targets independent sets of banks. This configuration allows the cache to be partitioned using address coloring.

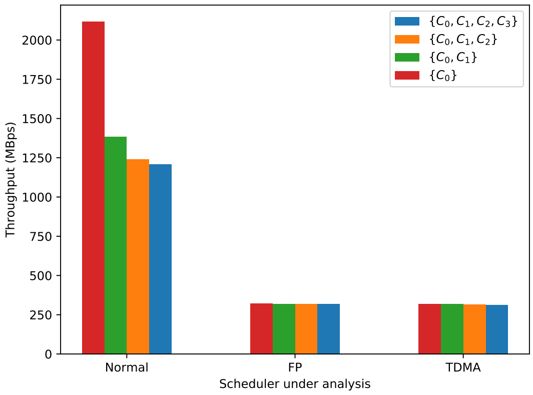


**Figure 4** Bandwidth in MBps for different path under increasing set of cores contending.

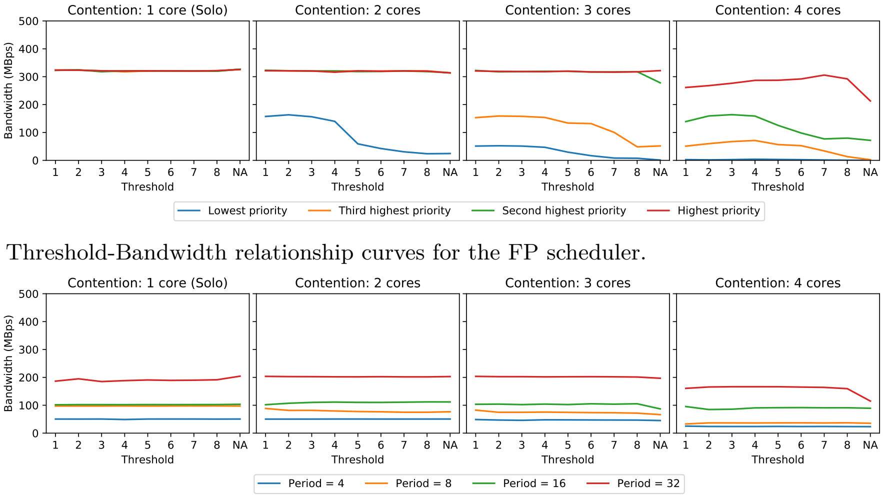
## 6.2 Platform Capabilities and performance degradation

Intuitively and as discussed in [29], redirecting the traffic coming from the cores to the PL side incurs a performance hit. In spite of the lower frequency at which the SchIM operates (250 MHz), the theoretical throughput when using both the HPM lanes should be around 8 GBps. We observe, however, that the achievable throughput through the HPM ports is a fraction of what we measured by accessing the main memory through the *normal route* (2116.5 MBps and 1207.41 MBps for solo and full contention by 3 other cores, respectively). We further provide a discussion on the bandwidth drop when transactions are routed through the PL in Section 7. For the sake of completeness, we quantify in Fig. 4 the maximum bandwidth achieved through the PL – and hence through the SchIM. Nevertheless, it is important to remember that the absolute figures are strictly platform dependent.

In Fig. 4, we have computed the throughput of one *core under analysis*, here core 0 (noted $C_0$) when a synthetic memory-intensive application is deployed on an increasing number of cores denoted with the same notation. The first bar cluster ("Normal") refers to the throughput measured via the normal route. The other two clusters capture the observed bandwidth when traffic is routed through and managed by the SchIM. One cluster is provided for each of the implemented memory scheduling policies, namely – from left to right – FP and TDMA. As expected, there is a sharp reduction (around 75%) in terms of absolute bandwidth. Importantly, however, two aspects need to be highlighted. First, the bandwidth achieved through the SchIM is still remarkably high and allows studying the behavior of the realistic workload under custom memory scheduling policies, which is the primary goal of this research. Second, it emerges that the implemented FP and TDMA policies are capable of protecting the core under analysis from inter-core interference, while this is not the case when going through the normal route.

## 6.3 PL-to-PS feedback performance impact

As mentioned in Section 5.7, the PL-to-PS feedback enables our SchIM to regulate the HPM ports buffer occupancy to prevent head-of-line blocking. Since this feedback directly throttles the desired core, the selection of an adequate threshold is important to preserve the balance between control and performance. Therefore, in Fig. 5, we have explored the sensitivity to the threshold for each of the proposed schedulers under different levels of contention. The thresholds in use range from 1 to 8 and even include the case where the feedback mechanism



**(a)** Threshold-Bandwidth relationship curves for the FP scheduler.

**(b)** Threshold-Bandwidth relationship curves for the TDMA scheduler.

**Figure 5** Figures showing the impact of the threshold in use on the final bandwidth experinced by the cores for the offered schedulers.

is disabled (noted *NA*). The contention is created by up to four co-running cores emitting write transactions. For each parameter applied to a scheduler (i.e., fixed priority or TDMA slot), the co-running cores are assigned the most demanding parameters available (i.e., the highest priority for FP or the biggest TDMA slot).

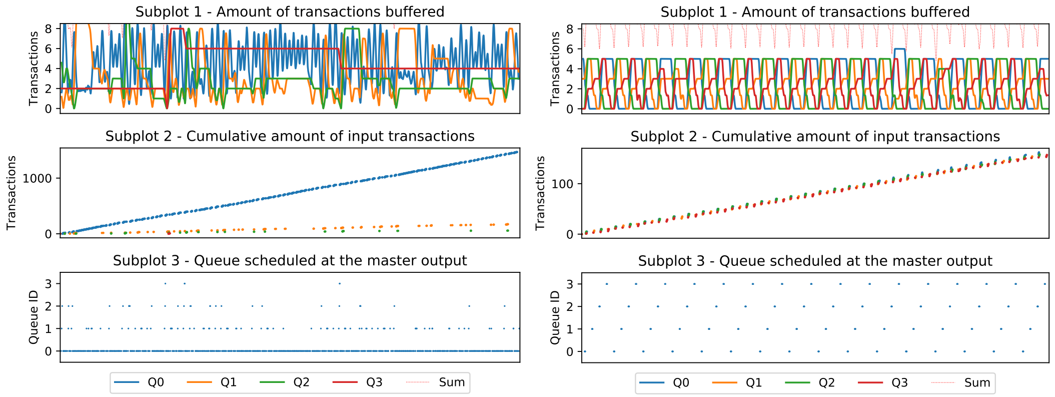
In the case of the FP scheduler (Fig. 5a), one can observe that when running alone, the threshold has no influence on the throughput. However, as soon as co-runners are added, the cores start to experience a decrease in throughput. Fig. 5b shows that the TDMA scheduler is not impacted considerably by the threshold with respect to the throughput. Globally, the scheduler manages to preserve a constant throughput regardless of the contention and the assigned slot.

Nonetheless, under high contention, one can observe that the throughput of each core is affected. The fourth inset of Fig. 5a and Fig. 5b illustrate the importance of the threshold and the PL-to-PL feedback mechanism as a a considerable drop of throughput can be observed for the highest priority of FP and for a TDMA period of 32.

Considering these experiments, setting the threshold to four for all the schedulers seems to bring the best trade-off between control and performance. However, this value cannot be blindly applied to all cases as this experiment is performed for a sequential and contiguous access pattern.

## 6.4 Internal Behaviour of SchIM

The next objective is to verify the correct behavior of the schedulers at the granularity of a clock cycle by observing the inputs, the outputs and the internal signals and registers of the SchIM module. This is made possible thanks to the *Integrated Logic Analyzer* (or ILA) provided by Xilinx [36]. The latter IP can be directly implemented on the PL side, alongside the SchIM, and is able to probe the signals and to store them in a local memory. For this experiment, a group of relevant internal signals have been probed and captured during a window of 16384 contiguous clock cycles. Then, the information has been extracted by post-processing the data. To characterize the behavior of the two different policies, the



**(a)** FP with ordering $C_0 \succ C_1 \succ C_2 \succ C_3$.

**(b)** TDMA with slots of 256 clock cycles.

**Figure 6** Trace snapshots of SchIM for FP (6a), TDMA (6b).

ILA has been instrumented to collect (i) the amount of transactions being buffered in the queues at each clock cycle (inset 1 in Fig. 6a and Fig. 6b) (ii) the rate at which queues receive new transactions from the cores cluster (inset 2 in Fig. 6a and Fig. 6b) and (iii) the queues ID of each transaction forwarded by the SchIM module (inset 3 in Fig. 6a and Fig. 6b).

For the Fixed Priority trace snapshot displayed in Fig. 6a, the following strict priority ordering has been considered: $C_0 \succ C_1 \succ C_2 \succ C_3$ where the $\succ$ operator means that the left argument has a strictly higher priority than the right argument. In this experiment, a regulation threshold of 3 for each core has been used. As emphasized by the inset 2 in Fig. 6a, the FP scheduler is able to prioritize the traffic of one core at the expense of the others according to the priorities assignment. Furthermore, one can observe that the rate at which the queues receive new transactions from their associated core is proportional to the priority level in the priority ordering. Finally, the third inset in Fig. 6a confirms the correct behavior of the FP policy.One can see that the cores with the highest priority also feature the highest density of transactions at the output of the SchIM.

The trace snapshot displayed in Fig. 6b has been obtained by configuring the SchIM module in TDMA mode. For the sake of clarity, a slot of 256 clock cycles has been set for each core. Besides, the threshold of each core has been set to 4 to create sharp transitions. The insets 2 and 3 of Fig. 6b clearly show the behavior expected from a TDMA schedule. In fact, one can clearly see in the latter that transactions originating from one core are only being repeated out of the SchIM module during a well-defined and periodic time slot of 256 clock cycles. In the inset 2 of Fig. 6b, we can observe a similar pattern, with transactions arriving only during the TDMA slot associated with their queue (and indirectly core). Globally, the rate at which queues receive transactions is steady and constant.

## 6.5 Memory Isolation

On the platform considered for this set of experiments, the Xilinx ZCU102 development board, we denote three main sources of inter-core performance interference: (1) cache contention, (2) DRAM bank conflicts, and (3) the congestion and saturation of the memory controller. Despite their orthogonality, the two first sources are tackled respectively via the integration of page coloring in the hypervisor and static bleaching in the SchIM. On the other hand, since the SchIM provides fine-grained control over the timing and ordering of transactions originating from the application cores as they reach the memory controller. Thus, the SchIM

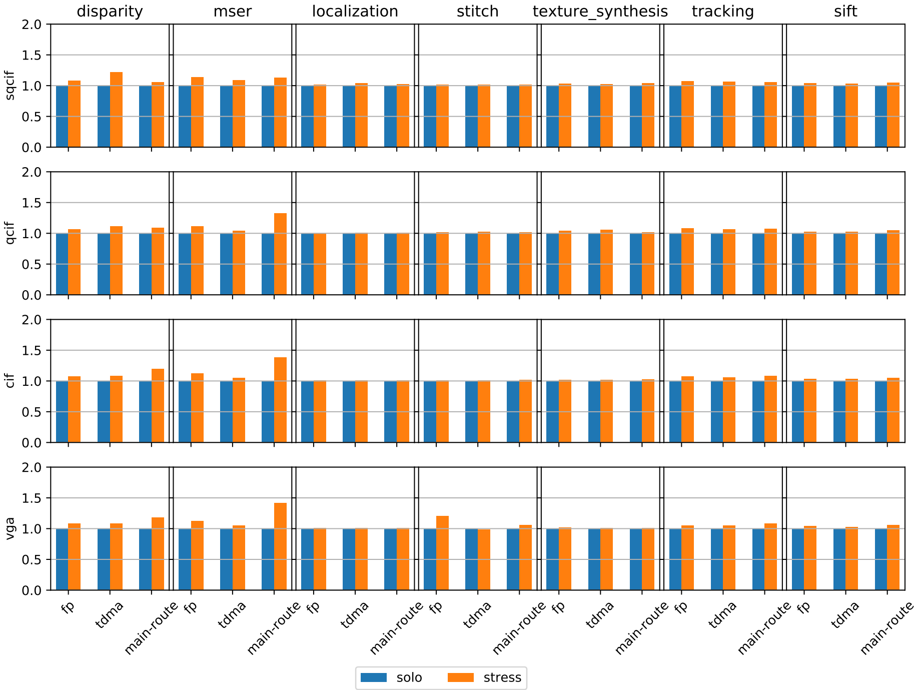


Figure 7 Normalized execution time for each benchmark and input size for *Solo* and *Stress*. Each column denotes a given benchmark of the SD-VBS suite, while each row denotes a specific input size (in increasing order from top to bottom).

brings memory bandwidth management into the PL, and provides not only regulation but a generic infrastructure to experiment with custom bandwidth management techniques, both work-conserving and non-work-conserving.

The evaluation setup considered for this experiment is identical to the one presented in Section 6.1. The routes going through the PL and using our SchIM (i.e., FP and TDMA) benefit from both cache partitioning and bank partitioning. On the other hand, the *normal route* uses cache partitioning and sees its cores divided into two sets targeting each a different group of private banks.

To evaluate the capability of our SchIM with respect to its ability to ensure performance isolation between the cores, a set of experiments involving SD-VBS benchmarks were designed. Here, we compare the execution time of an application on a given core when running alone (referred to as *Solo*) and when running alongside interfering synthetic benchmarks (write memory bombs) on all the other cores (referred to as *Stress*). For each combination of a route to main memory (i.e., the *normal route* or the *SchIM route*) and scheduler, the result obtained for *Stress* is normalized with respect to the equivalent configuration in *Solo*. The results obtained on the considered benchmarks are listed in Fig. 7. The results in the Fig. 7 are the aggregation (arithmetic average) of 30 different runs in the same configuration. Each bar cluster of the Fig. 7 insets represents one of the aforementioned configuration for *Solo* and *Stress*. The height of each bar denotes its normalized execution time.

For this set of experiments, the FP scheduler was configured such that the core under analysis (i.e., the one running the benchmark) has the highest priority and a threshold of 8. The other cores are assigned lower priorities and thresholds matching their priority order (i.e., 4, 2, 1). Under TDMA scheduling, the core under analysis has a slot of 512 clock cycles and a threshold of 14 while the co-runners are assigned slots of 32 and 16 clock cycles with thresholds of 4 and 1.

The *normal route* is used as a baseline for this experiment because no scheduling is performed in this configuration. The Fig. 7 highlights the sensitivity of both *disparity* and *mser* to inter-core interference on the *normal route*. This is especially the case for large input sizes such as *cif* and *vga*. On the other hand, *texture synthesis* and *localization* do not suffer from inter-core interference. Globally, the TDMA scheduler always manages to preserve the isolation of the core, having execution times under *Stress* similar or smaller than the *normal route*. This is particularly visible for *qcif*, *cif* and *vga* input sizes of *disparity* and *mser*. Similarly, the FP scheduler is also capable of ensuring sound isolation of the core under analysis.

## 7 Discussion and Limitations

By design, the PLiM module proposed in this paper, the SchIM, centralizes the memory traffic and its scheduling. A centralized design makes sense on the specific target platform because there exist only one memory controller and thus a single path between the LLC and the DRAM controller. In systems where multiple paths between the processing units and the memory controllers exist, for instance when multiple controllers and channels are present, a decentralized design is to be preferable to better exploit the available memory parallelism. In such platforms, a possible avenue could be instantiating multiple SchIM modules, roughly one per channel, and introducing appropriate out-of-band signaling between the modules for coordination off the critical path.

As we mentioned in Section 6.1, our setup includes the Jailhouse partitioning hypervisor. While the SchIM module does not strictly require the PS side to use a hypervisor, Jailhouse has been extensively used for the evalution as it provides convenient features to control physical memory allocation. For instance, the support for page coloring has been used to both partition the LLC space and to easily identify the owner of each memory transactions in the SchIM (as presented in Section 5.4). However, instead of enforcing cache partitioning, one could instead identify the ownership of memory transactions by extracting a different subset of address bits. For instance, if the physical memory allocated to different partitions is not interleaved, then the most significant bits of the address can be used to perform traffic accounting. In addition, the IPA address virtualization is convenient to transparently redirect the memory traffic of the application partitions through the PL side, even if they are initially booted through the normal route. Finally, the cores throttling mechanism (see Section 5.7) via the FIQs can be implemented at EL3 (Secure Monitor) or in the individual guest OS's instead (EL1). Implementing FIQ handling in the hypervisor (EL2), however, has the advantage of not requiring any change in the guest OS's, as well as not requiring a full switch into secure mode compared to an implementation at EL3.

On the same note, provided that the FIQ lines are not used by the inmates, the feedback regulation mechanism is entirely transparent to the guest OS's (or even for bare-metal applications) and introduces minimum overhead. The Linux kernel do not use FIQs, and the same goes for typical RTOS's. Nonetheless, it must be acknowledged that defining a FIQ handler to be used for CPU throttling might interfere with (and be interfered by) the latency of FIQ handling in guest OS's that rely on the same functionality. This is mainly because FIQ handling is non-preemptive. We also recognize that the PL-to-PS feedback mechanism is relatively coarse. Inset 1 of Fig. 6b highlights this problem. Even though all the queues have been assigned a threshold of 4, the threshold is often exceeded. The worst-case being queue 3 exceeding the threshold by 2 on the right-hand side of the plot. This problem can be attributed to the reaction time of the FIQ routine, and to the fact that jumping to the FIQ handler itself might cause a few memory transactions depending on the cache state. Currently, the thresholds used for FIQ-based regulation require to be fine-tuned manually by the user. Future extensions of the SchIM will explore the implementation of schedulers capable of dynamically adapting the thresholds to maximize performance and improve isolation.

The loss in bandwidth caused by routing transactions through the PL is important and a serious drawback against the adoption of the SchIM. Our experiments in Section 6.2 have shown that rerouting the traffic through the PL has a cost. As illustrated in Fig. 4, up to 2100 MBps can be extracted from the *normal route* whereas any route through the PL only achieves around 320 MBps. In contrast, a back-of-the-envelope calculation reveals that for a PL operating at 250 MHz (the SchIM frequency), and with a bus width of 128 bits, a full-duplex throughput of approximately 3.7 GBps can be sustained. This calculation is in line with the reported throughput in an experiment conducted in [19], in which PL-originated transactions targeting the DRAM passed through the one of the HP ports. This suggests that the PL-to-DRAM route can sustain a much higher throughput than what has been experimentally observed in our evaluation setup, where transactions originate from the PS side. In light of these considerations, we can conclude that the source of the bandwidth loss can be imputed to the bus segments connecting the CPU cluster to the HPM ports. A focused study is necessary to narrow down the exact reason for the performance drop. Nonetheless, vendor-imposed bandwidth throttling, PS-to-PL clock-domain crossing delays, and shallow FIFOs at the HPM ports and/or at the main PS-side interconnect represent plausible reasons. We anticipate that due to the platform-specific nature of this issue, the raw performance of the SchIM will substantially vary across different SoCs.

## 8 Conclusion

In the present article we introduced the SchIM, a memory transactions scheduler framework that can be integrated with commercially available platforms featuring a tightly coupled processing system and programmable logic. A full-system implementation in a commercially available PS-PL platform has been detailed, which encompasses the accompanying software stack and the platform-specific integration steps.

Through a set of experiments, we assessed the capabilities of the framework and demonstrated the correct behavior of the proposed scheduling policies, namely Fixed Priority and Time Division Multiple Access. Finally, we showed using a suite of real-world benchmarks that the SchIM is capable of enforcing strong temporal isolation despite heavy memory contention.

The authors see the proposed SchIM as a stepping stone to propose, test and validate novel memory scheduling policies to be tested on embedded platforms with realistic performance and complex workload. For this reason, the SchIM has been designed to be open-source and with extensibility in mind. Especially, we strongly envision that the SchIM could represent a stepping-stone toward profile-based memory traffic scheduling.

## References

1 B. Akesson. *Predictable and composable system-on-chip memory controllers.* PhD thesis, Technische Universiteit Eindhoven, School of Electrical Engineering, 2010. doi:10.6100/IR658012.

2 B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, 2007.

3 G. Alonso, T. Roscoe, D. Cock, M. Ewaida, Kaan Kara, Dario Korolija, D. Sidler, and Ze ke Wang. Tackling hardware/software co-design from a database perspective. In *Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, Netherlands, January 2020.

4 ARM. ARM® CoreLink™ QoS-400 Network Interconnect Advanced Quality of Service, 2013. Accessed on 09.01.2020.

5 ARM. AMBA AXI and ACE Protocol Specification. Technical report, ARM, 2019. URL: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf.

6 A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016. doi:10.1109/RTSS.2016.010.

7 J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS$^{\mathrm{RT}}$ : A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126, 2006. doi:10.1109/RTSS.2006.27.

8 F. Farshchi, Qijing Huang, and H. Yun. BRU: Bandwidth regulation unit for real-time multicore processors. *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375, 2020.

9 F. Farshchi, P. Kumar, R. Mancuso, and H. Yun. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:25, Barcelona, Spain, July 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2018.1.

10 C. Ferri, A. Marongiu, B. Lipton, R. Bahar, T. Moreshet, L. Benini, and M. Herlihy. SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 39–48, 2011.

11 G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing mixed criticality applications on modern heterogeneous MPSoC platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

12 Intel, Corp. Intel's Stratix 10 FPGA: Supporting the smart and connected revolution, October 2016. Accessed on 09.01.2020. URL: https://newsroom.intel.com/editorials/intels-stratix-10-fpga-supporting-smart-connected-revolution/.

13 A. K. Jain, S. Lloyd, and M. Gokhale. Microscope on memory: MPSoC-enabled computer memory system assessments. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 173–180, 2018. doi:10.1109/FCCM.2018.00035.

14 H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014. doi:10.1109/RTAS.2014.6925998.

15 H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

16 J. Kiszka, V. Sinitsin, H. Schild, and contributors. Jailhouse Hypervisor. Accessed on 09.01.2020. URL: ttps://github.com/siemens/jailhouse.

17 C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.*, 52(3), 2019. doi:10.1145/3323212.

18 Microsemi — Microchip Technology Inc. PolarFire SoC - Lowest Power, Multi-Core RISC-V SoC FPGA, July 2020. Accessed on 09.01.2020. URL: https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga.

19 S. Min, S. Huan, M. El-Hadedy, J. Xiong, D. Chen, and W. Hwu. Analysis and optimization of I/O cache coherency strategies for SoC-FPGA device. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 301–306, 2019. doi:10.1109/FPL.2019.00055.

20 R. Mirosanlou, M. Hassan, and R. Pellizzoni. DRAMbulism: balancing performance and predictability through dynamic pipelining. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94, 2020. doi:10.1109/RTAS48715.2020.00-15.

21 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018.

22 O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160. IEEE, 2007.

23 O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *2008 International Symposium on Computer Architecture*, pages 63–74. IEEE, 2008.

24 K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222. IEEE, 2006.

25 M. Paolieri, E. Quinones, F. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters*, 1(4):86–90, 2009.

26 N. Rafique, W. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258. IEEE, 2007.

27 F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo. AXI HyperConnect: A predictable, hypervisor-level interconnect for hardware accelerators in FPGA SoC. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:10.1109/DAC18072.2020.9218652.

28 F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo. Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs. *ACM Trans. Embed. Comput. Syst.*, 18(5s), 2019. doi:10.1145/3358183.

29 S. Roozkhosh and R. Mancuso. The potential of programmable logic in the middle: Cache bleaching. In *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*, Sydney, Australia, April 2020.

30 P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: a system-wide framework for memory bandwidth profiling and management. In *41st IEEE Real-Time Systems Symposium (RTSS 2020)*, Houston, TX, USA, December 2020.

31 ST Microelectronics Inc. Real-time performance using FIQ interrupt handling in SPEAr MPUs, January 2010. Accessed on 10.01.2020.

32 M. Solieri T. Kloda, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019)*, pages 1–14, Montreal, Canada, April 2019. doi:10.1109/RTAS.2019.00009.

**33** H. Usui, L. Subramanian, K. Chang, and O. Mutlu. Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–28, 2016.

**34** P. Valsan and H. Yun. MEDUSA: A predictable and high-performance DRAM controller for multicore based embedded systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 86–93. IEEE, 2015.

**35** S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009.

**36** Xilinx. Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide. Technical report, Xilinx, 2016. URL: `https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf`.

**37** Xilinx. Zynq UltraScale+ Device Technical Reference Manual. Technical report, Xilinx, 2019. URL: `https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf`.

**38** M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356, 2019. `doi:10.1109/RTAS.2019.00036`.

**39** H. Yun, W. Ali, S. Gondi, and S. Biswas. BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2017.

**40** 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, 2014. `doi:10.1109/RTAS.2014.6925999`.

**41** 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

**42** Y. Zhou and D. Wentzlaff. MITTS: Memory inter-arrival time traffic shaping. *ACM SIGARCH Computer Architecture News*, 44(3):532–544, 2016.

# Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC

**Alejandro Serrano-Cases**
Barcelona Supercomputing Center (BSC), Spain

**Juan M. Reina**
Barcelona Supercomputing Center (BSC), Spain

**Jaume Abella**
Barcelona Supercomputing Center (BSC), Spain
Maspatechnologies S.L, Barcelona, Spain

**Enrico Mezzetti**
Barcelona Supercomputing Center (BSC), Spain
Maspatechnologies S.L, Barcelona, Spain

**Francisco J. Cazorla**
Barcelona Supercomputing Center (BSC), Spain
Maspatechnologies S.L, Barcelona, Spain

## Abstract

The interference co-running tasks generate on each other's timing behavior continues to be one of the main challenges to be addressed before Multi-Processor System-on-Chip (MPSoCs) are fully embraced in critical systems like those deployed in avionics and automotive domains. Modern MPSoCs like the Xilinx Zynq UltraScale+ incorporate hardware Quality of Service (QoS) mechanisms that can help controlling contention among tasks. Given the distributed nature of modern MPSoCs, the route a request follows from its source (usually a compute element like a CPU) to its target (usually a memory) crosses several QoS points, each one potentially implementing a different QoS mechanism. Mastering QoS mechanisms individually, as well as their combined operation, is pivotal to obtain the expected benefits from the QoS support. In this work, we perform, to our knowledge, the first qualitative and quantitative analysis of the distributed QoS mechanisms in the Xilinx UltraScale+ MPSoC. We empirically derive QoS information not covered by the technical documentation, and show limitations and benefits of the available QoS support. To that end, we use a case study building on neural network kernels commonly used in autonomous systems in different real-time domains.

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

**Keywords and phrases** Quality of Service, Real-Time Systems, MPSoC, Multicore Contention

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.3

**Funding** This work has been partially supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB; the European Union's Horizon 2020 research and innovation programme under grant agreement No. 878752 (MASTECS) and the European Research Council (ERC) grant agreement No. 772773 (SuPerCom).

## 1 Introduction

Satisfying the increasing computing performance demands of critical software applications requires Multi-Processor System-on-Chip (MPSoC) devices that incorporate diverse computing elements [42, 59]. Distributed interconnects are also required on the MPSoC for fast communication between masters (e.g. CPUs) and slaves (e.g. on-chip memories and memory controllers). For instance, the Zynq UltraScale+ MPSoC [59], which we refer to as ZUS+ in

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 3; pp. 3:1–3:26
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

this work, comprises two CPU clusters, with CPUs with different power and performance points, a Graphics Processing Unit (GPU), a Field Programmable Gate Array (FPGA) that allows synthesizing specific accelerators, and an AXI4-based distributed interconnect.

Complex MPSoCs accentuate the problem of multicore contention, i.e. controlling the interference co-running tasks generate on each other. In an MPSoC, tasks can interact in many hardware resources and controlling how such resources are shared becomes a necessary precondition to derive useful timing bounds. This can be achieved via software-controlled hardware mechanisms like cache partitioning (e.g. provided in the NXP T2080 [27]) to prevent tasks from evicting each other's cache data, and hardware-thread prioritization in simultaneous multithreading (SMT) IBM [15] and Intel [31] processors. Hardware QoS mechanisms like these help controlling multicore contention: by properly configuring the hardware QoS mechanisms, the system software (RTOS or hypervisor) can favor the execution of specific tasks, reducing the slowdown they suffer due to contention, at the cost of increasing the impact of contention on (less time-constrained) co-runner tasks. This offers a rich set of platform configurations that allow the end-user to better adapt to the criticality and timing constraints of the running application workload.

In this paper, we analyze the hardware support for QoS in the ZUS+, which is assessed for on-board computing in avionics [58]. The ZUS+ offers a rich set of QoS mechanisms implemented in different hardware IP blocks of the interconnect and the memory controller. The number, diversity, and complexity of those mechanisms are, at a first sight, simply overwhelming: up to 4 different hardware IP components in the ZUS+ are QoS improved. Some of those components are instantiated several times resulting in (i) over 30 different QoS points that control the flow of traffic in the interconnect and the access to the slaves; and (ii) millions of possible QoS configurations. However, QoS can only work effectively if the QoS points work coordinately. Otherwise, a QoS point down the path from the source to the destination can cancel out all the prioritization benefits achieved through previous QoS points. This calls for a detailed analysis of the different QoS mechanisms and their dependencies to reach a global predictability goal. In this line, our contributions are:

**Individual QoS mechanisms.** (Section 3) We analyze several QoS-enabled IP components from 2 different IP providers instantiated in the ZUS+: the Arm NIC-400 [4], and its QoS-400 [5] and QVN-400 [6] extensions, the Arm CCI-400 [9], and the Synopsys uMCTL2 [55] DDR memory controller. We describe the main QoS features in each of these components as building blocks of the analysis performed in the rest of this work.

**Coordinated QoS mechanisms.** (Section 4) Following the individual analysis of QoS-enabled IP blocks, we analyze how QoS mechanisms can work coordinately to achieve a global goal, e.g. favoring the traffic of the Real-time Processing Unit (RPU) over the Application Processing Unit (APU). This analysis, which is not provided in the ZUS+ or its IP blocks' technical reference manuals, presents key insights to fully master the QoS support in the ZUS+. In particular, (i) we show that some QoS features, especially when provisioned by different IP providers, can be fundamentally incompatible and hence, cannot be deployed together towards reaching a common predictability goal; (ii) for compatible QoS features in different IP blocks, we show the particular range of QoS configuration values that can be used to prevent that one feature cancels out the benefits brought by another. In doing so, we introduce the new concepts of *QoS domain* and *QoS domain mapping*; and (iii) we also show the missing information about QoS mechanisms in the technical manuals of the ZUS+.

**Characterization.** (Section 5) Driven by the analysis in Section 4, we perform controlled experiments to characterize QoS mechanisms in different IP blocks, with a view to determining some of the design choices made by Xilinx when instantiating Arm IP blocks as they are not documented in the corresponding technical manuals. Also, we note that all four A53 cores in the APU share a single QoS-enabled port to the interconnect that allows controlling the aggregated traffic but not per-core traffic, which in practice prevents having several applications in the APU if they have different QoS needs. We unveil how QoS and packet routing can be combined to overcome this limitation, allowing two applications to run in the APU with heterogeneous QoS requirements.

**Case Study.** (Section 6) We focus on a composite deployment scenario comprising several applications, each one potentially subject to different predictability requirements, to show how hardware QoS configuration is a central element of the platform configuration to ensure applications meet their timing constraints. We use representative neural network kernels to show that, by deploying specific QoS setups, the time constraints of the different applications can be accommodated while other metrics, like average performance, can be improved. This is very useful in different domains for platform configuration selection, referred to as intended final configuration (IFC) in CAST-32A [18] in the avionics domain.

The rest of this work is organized as follows. Section 2 introduces the most relevant related works. Section 3 to Section 6 cover the main technical contributions of this work, as described above. Last but not least, Section 7 provides the main conclusions of this work and discusses future research directions.

## 2 Background and Related Works

Multicore contention is at the heart of the complexities for the adoption of MPSoCs in high-integrity systems (e.g. avionics and automotive). This has impacted domain-specific safety standards and support documents [18, 2, 32] and led to the proliferation of academic and industrial studies on modeling multicore interference [46].

**Contention Modelling.** Contention Modelling is one of the main multicore-contention related research lines covering COTS chips for avionics [40] and automotive [22]. Analytical approaches aim at bounding the contention impact on shared hardware resources, initially focusing on the timing interference in shared on-chip buses [52, 19, 20] and later extended to include other shared resources. Solutions have been proposed to make Advanced Microcontroller Bus Architecture (AMBA) protocols time-composable [33], and to achieve a fair bandwidth allocation across cores considering requests with different durations [50]. Other works target more complex interconnects, bounding their worst-case traversal time [35, 26], focusing on Network on Chips (NoCs) specifically [49, 21, 17, 57, 13], and modeling contention with network calculus [34, 47]. For the DDR memory, some authors build on static analyses to derive bounds to the latencies of memory requests considering other potential requests in the memory controller [29], as well as information about tasks and requests simultaneously [30]. For cache memories, contention has been modeled statically, as surveyed in [36], as well as analyzed with measurements on COTS multicores, targeting the coherence protocol [53]. The tightness and precision of analytical approaches are challenged by the complexity of the hardware and software under analysis. For this reason, other approaches are proposed to exploit specific application semantics or dedicated hardware and software support.

**Application Semantics.** Several works have been advocating the enforcement of predictable application semantics where task memory operations are only allowed to happen in dedicated phases (e.g., read-compute-write). This enables the computation of tighter contention bounds and the formulation of contention-aware co-scheduling approaches [45, 44, 12, 14]. While unquestionably effective, not all applications can support an execution semantics allowing a reasonable and clear separation into phases.

**Exploiting hardware support for QoS in COTS.** For simultaneous multi-threading processors some authors have exploited existing fetch policies to allocate core resources to threads in the context of HPC applications running for IBM POWER- processors [15] and Intel processors [31]. In real-time systems, other authors have focused on an individual Arm QoS element and a specific example (memory traffic from accelerators) to show that QoS mechanisms could be effectively leveraged for a better application consolidation [54]. Other authors evaluate the throughput of DDR memory on a ZUS+, including the impact of one QoS parameter in the memory controller [37]. In our work, we analyze/characterize the specific realization of Arm QoS IPs in the ZUS+ SoC and consider how to orchestrate multiple QoS mechanisms for an effective QoS management. In the short and mid term, we foresee chip providers will further support advanced QoS features and mechanisms such as, for instance, the Memory System Resource Partitioning and Monitoring (MPAM) in Arm-V8 architectures [8], which is under evaluation by industry in the real-time domain [23].

**Software-only solutions.** Software-only solutions for contention control do not require specific hardware support for either enforcing task segregation or providing a given level of QoS guarantees. These techniques leverage information on set and bank indexing in caches and memory, and hypervisor/RTOS allocation support to force different tasks to be mapped to different cache sets and DDR memory banks/ranks [28, 38]. Other solutions focus on controlling the access to shared resources (e.g. memory) as a way to control the maximum contention an offending task can generate on its co-runners [60] and also to guarantee performance of critical tasks while dynamically minimizing the impact on best effort tasks [1].

**Specific hardware proposals.** Specific hardware proposals for contention control include some general resource management policies [39]. The number of resource-specific proposals is high and covers a wide variety of mechanisms including changes in the arbitration and communication protocols [33], memory bandwidth regulation [24], support for cache partitioning [38] (in some cases building on existing programmable logic in the SoC [51]), control contention bounds [16], exploit AMBA AXI bandwidth regulation for accelerators in FPGAs [43].

In this work, we do not propose hardware support for contention control, but build on that provided by default by the MPSoC integrator. Unlike previous works that focus on centralized QoS control, we address the challenge of understanding, characterizing, and showing the limitations and benefits of a distributed QoS system like the one in the ZUS+.

## 3 Analysis of the QoS Mechanisms in the Zynq UltraScale+ MPSoC

The ZUS+ integrates several computing and memory components, all connected by a distributed interconnect fabric, see Figure 1. The main computing elements are the quad-core Arm Cortex-A53 APU, the dual-core Arm Cortex-R5 RPU, the Arm Mali-400 GPU, and

**Table 1** Main QoS-related terms used in this work.

| Term | Definition |
|---|---|
| QoS mechanism | Specific hardware mechanisms in a QoS-enabled block to control QoS |
| QoS slot (point) | Refers to the instantiation of a QoS-enabled block or mechanism |
| QoS feature | Specific QoS characteristic implemented by a QoS mechanism |
| QoS value | Specific value given to a QoS feature |
| QoS setup | Set of values for the QoS features of all QoS points under evaluation |

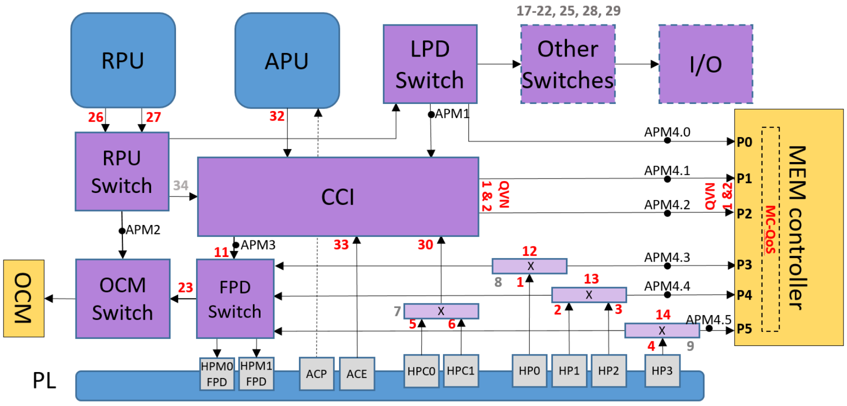


**Figure 1** Simplified ZUS+ block diagram emphasizing APU/RPU/PL and OCM/DDRC.

the accelerators that can be implemented in the Programmable Logic (PL). The memory system comprises several on-chip units like the On-Chip Memory (OCM) and the controller of the DDR SDRAM, which is the main off-chip memory. The interconnect comprises top-level switches, namely, the Cache Coherent Interconnect (CCI), the low-power domain (LPD) switch, full-power domain (FPD) switch, and the OCM switch; and a high number of second-level switches, highlighted as "x" in Figure 1. In this work, we focus on communication from the APU, RPU, and the PL to the OCM and the DDR DRAM. Other blocks that are not the focus of this work are not developed in the figure, e.g., IP blocks related to I/O are abstracted as "Other Switches" and "I/O". APM are the Xilinx AXI performance monitoring point blocks used to collect statistics on the packets sent over an AXI link (reads, writes, amount of data transferred, etc.).

In terms of third party IPs, the ZUS+ equips a distributed AMBA AXI4 [11] interconnect, with switches based on the Arm NIC-400 [4] and its QoS-400 extension [5]. The CCI, instead, is based on the Arm CCI-400 [9] and equips similar features to the Arm QVN-400 [6] IPs. The memory controller builds on the Synopsys uMCTL2 [55]. Each block provides hardware support for QoS, which we analyze here. To support our discussion, Table 1 introduces the main terms we use in this work.

## 3.1 QoS support per IP-block

The ZUS+ technical documentation [59] provides very limited information about the functional behavior of the underlying IP blocks on which it builds. Hence, we start by analyzing the information on supported QoS features that can be obtained from each IP's technical specification. How each IP is instantiated in the ZUS+ is covered in Section 4.3.

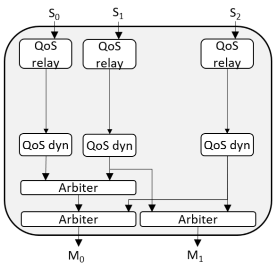


Figure 2 Arm QoS-400 QoS relay and dynamic QoS.

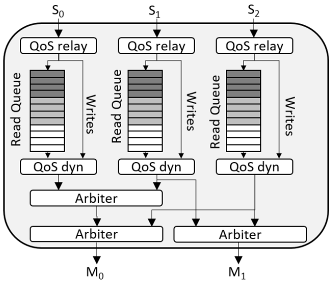


Figure 3 QoS features in the Arm CCI-400.

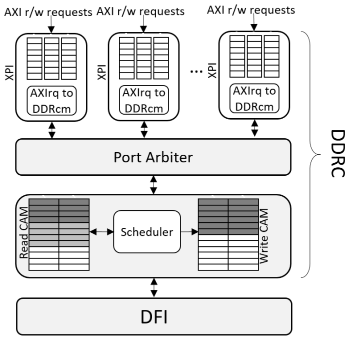


Figure 4 Block Diagram of the DDR memory controller.

**Arm AXI4 [11].** AXI4 presents 5 communications channels, three from master to slave (address read, address write, and write data) and two from slave to master (read data and write response). In the read address and write address channels, AXI4 implements QoS-specific signals by supporting two 4-bit QoS identifiers for read (ARQOS) and write (AWQOS) transactions, indistinctly referred to as AXQOS. Transaction initiators (masters) set the QoS value for each read/write request. QoS values range from 0 (default or neutral QoS) to 15, with higher values meaning higher priority. We refer to this feature as static QoS.

**Arm NIC-400 [4].** On arrival to a NIC-400, every transaction allocates a QoS value by assigning (i) a fixed value statically defined when the NIC-400 IP is integrated into the SoC; (ii) a programmable value provided via NIC-400 control registers; or (iii) the QoS value received from the attached master. We call this feature QoS relay. As the transaction traverses internal switches in the NIC-400, static QoS values are used to decide which transaction has to be served first. In these arbitration points, the transaction with the highest value is prioritized using Least Recently Granted (LRG) as a tie-breaker.

**Arm QoS-400 [5].** Arm QoS-400 is an extension to the NIC-400 that provides additional QoS features, remarkably three dynamic QoS regulation mechanisms: outstanding transaction that limits the maximum number of read, write, or read+write in-flight transactions allowed; transaction rate that dynamically increases QoS value of transactions if the latency is greater than the target and vice versa; and transaction latency that controls the period between successive request handshakes dynamically increasing/decreasing QoS values when the observed period is greater/lower than the target [5] . The regulation builds on three parameters: the Peak, Burstiness and Average. The average controls how many transactions need to be made within a period of time. When not achieved this amount of transactions due to the system congestion, then the regulator allows performing a limited set of transactions (burstiness) to restore the average. In addition, the control can be configured to limit this transactions issue to not overuse the shared resources (Peak). As an illustrative example, Figure 2 shows a block diagram of the QoS features in a Arm NIC-400 interconnect block encompassing QoS extensions with 3 slaves and 2 master ports, and 3 arbiters.

**Arm QVN-400 [6].** Arm QVN-400 is an extension to the CoreLink NIC-400. The QVN protocol creates virtual networks by using tokens to control transaction flows. QVN extends the common AXI channels with extra signals ensuring that a transaction can always be accepted at its destination before a source sends it. The number of VN (virtual network) is defined during the IP implementation. QVN enables transactions on virtual networks with different QoS values to avoid a blocking (less priority) transaction in a queue (Head-Of-Line).

**Arm CCI-400 [9].** Arm CCI-400 has similar features to the QoS relay, QoS dynamic and QoS static features in the QoS-400, and QVN in the QVN-400. A unique feature of the CCI-400 is that each master interface implements a *read queue* with several slots reserved for high priority requests, other for high+medium priority requests and the rest for low-priority requests (see Figure 3). The QoS values considered as high, medium, low are configurable. So is the number of reserved entries medium and high priority requests. We call this feature CCIrq.

**Memory Controller.** It comprises the DDR-controller (DDRC) that schedules and converts AXI requests (AXIrq in Figure 4) into DDR commands (DDRCm in Figure 4) and the DDR-PHY (DFI) that translates the requests into specific signals to the target DDR device. The DDRC dynamic scheduling optimizes both bandwidth and latency using a programmable QoS controller to prioritize the DFI requests, allowing out-of-order execution.

Six AXI ports (XPI) receive traffic, i.e. flow of AXI requests, going to the DDRC. XPIs are referred to as P0-P5 in Figure 1. In each XPI, the DDRC translates and classifies AXI transactions into a set of DDR commands. In each port, different queues temporarily store transactions depending on their type (read/write and request/data), see Figure 4. Read transactions are classified into low, high, and video traffic classes (LPR, HPR, and VPR, respectively), while write transactions are classified into low (or normal) and video (LPW/NPW and VPW, respectively) traffic classes. Commands with VPR/VPW behave as low priority when the command has not expired (i.e. there is not a transaction timeout). Once expired, the command are promoted to a priority higher than the HPR/NPW commands.

Once the transactions make their entry on the DDRC and their translation into DRAM commands are generated, those commands are stored into the counter addressable memories (CAMs). A read CAM and a write CAM are shared by all ports in a way that the maximum number of entries that can be allocated to a traffic class can be limited.

The Port Arbiter (PA), which is shared among all ports, selects the command to be sent to the CAMs based on several levels of arbitration, as shown next. *Operation type*: reads are prioritized while there are VPR expired or there are reads and no expired VPW. Writes are executed when there are no reads and if there are expired VPW and no expired VPR. The expiration period can be configured via setting timeouts for VPR/VPW [1]. Also, ports can be individually flagged as "urgent" to force all its request to be processed immediately. *Channel*: the PA prioritizes commands from higher priority classes: HPR has higher priority than LPR/VPR on the read channel and NPW/VPW has the same initial priority on the write channel, with VPR/VPW prioritized if they time out. *AXQOS*: in the next layer, priorities are given per command based on AXQOS signals. *Tie breaker*: in the bottom tier conflicts are resolved using round-robin arbitration.

This nominal behavior is affected by port throttling based on the occupancy of CAMs. When the available entries for HPR/LPR in the read CAM is below an HPR/LPR threshold, low-latency(HPR)/best-effort ports can be throttled. Likewise, if the available entries for NPW in the write CAM is below a threshold best-effort ports can be throttled.

[1] Note that there is a port "aging" feature that is set at boot time and is explicitly recommended not to be used with AXQOS: "aging counters cannot be used to set port priorities when external dynamic priority inputs (arqos) are enabled". Hence, we do not enable this feature in our experiments.

When issuing commands from CAMs to the DFI, command reordering is allowed to favor page hits, potentially causing out-of-order execution of the commands. A regulator limits the issue to up to 4 out-of-order commands. When it is disabled, no restriction is applied, resulting in no control in the number of out-of-order command executed. In our setup for predictability reasons, we limit it to its minimum value, 4. Also, HPR and LPR partitions in read CAM and the write CAM can enter a "critical" state if they spend more than a given number of cycles without issuing a command to the DFI.

## 4 Interaction Among QoS-enabled IP Blocks

We faced two main challenges in our attempt to orchestrate the different and distributed QoS mechanisms implemented in the ZUS+.

1. Xilinx provides very limited information about the QoS-enabled blocks it integrates into the ZUS+ and, instead, refers the reader to the technical manuals of each IP provider. However, the latter provides implementation-independent descriptions rather than details on the particular implementation options selected for the ZUS+ IP blocks. As a result, we could not find the specific implementation options for some IP blocks (Section 4.3) and had to derive them empirically instead (Section 5).
2. Xilinx provides almost no information on how the different QoS mechanisms – coming from different IP providers – can work coordinately. However, to effectively exploit all QoS features in view of a common predictability goal, it is necessary to properly configure all the QoS points from the different masters to the slave. For instance, in the ZUS+ a request from the RPU to the DDR, see Figure 1, crosses: a static QoS point in the RPU switch; read queue priority, QoS relay, and dynamic QoS in the CCI; QVN between the DDRC and the DRAM controller; and the multilayer arbitration in the DRAM controller which involves XPI, Port Arbiter, and CAMs.

This section tackles those issues by providing key insights and unveiling details, not documented in any technical reference manual, about the instantiation of QoS-enabled blocks in the ZUS+ and the interaction among them. This required cross-matching information in the technical manuals of the different IP providers and covering the conceptual holes found in the documentation by analyzing dozens of processor registers that control the operation of the QoS. The outcome of the analysis is the central element to guide the experimental part [2].

Overall, this section encompasses two well-differentiated parts. First, an engineering effort to derive missing information on QoS-enabled IP blocks, which is complemented with specific characterization and reverse-engineering experiments in Section 5. And second, a structured attempt to orchestrate the different QoS mechanisms by introducing concepts like QoS domains and QoS domain mapping. The former is more ZUS+ dependent, while the latter sets the basis for a methodology for analyzing the QoS support in other MPSoCs.

### 4.1 QoS domains and mappings

In order to capture the interactions between different QoS-enabled IP blocks, we define the concept of *QoS domain* as a set of QoS-enabled IP devices, or elements thereof, under which request prioritization is carried out using the same QoS abstraction (i.e. QoS values that vary

[2] It is worth noting that the analysis in this section required several months of effort by hardware experts. In fact, deriving the information in this section has taken longer than the experimentation part itself.

over the same range and have the same meaning). We also define the *QoS domain mapping* abstraction to capture the interaction among QoS domains and how the priorities levels in different domains are related. In the ZUS+, we differentiate the following QoS domains.

- **AXQOS**. Prioritization of AXI requests based on AXI QoS (ARQOS and AWQOS), classified in the previous Section as static QoS.
- **CCIrq**. Prioritization of read requests arriving at the slave interfaces, based on three levels (high, medium, low), with requests in the high tier being reserved some entries in the read queue, high+medium being reserved another set of entries, and low-priority requests using the rest of the entries in the queues.
- **QVN**. Prioritization using the virtual network ID. The master id determines for each transaction the virtual network it is mapped to.
- **DDRCreg**. Prioritization over *traffic regions*. On every port of the DDR controller (P0-P5) two regions [3] are defined, respectively referred to as region0 and region1.
- **DDRCtc**. Prioritization over *traffic classes*. The read channel is associated one traffic class: high-priority (HPR), low priority (LPR), or video priority (VPR). For the write channel the traffic class is normal write priority (NPW) or video priority write (VPW).

From these QoS domains, we define the following QoS domain mappings:

- **AXQOS-CCIrq** allows defining the set of static QoS values assigned to the high, medium, and low priorities. QoS values from 0 to $i$ mapped to the low priority, from $i$ to $j$ mapped to the medium priority, and from $j$ to 15 mapped to the high priority (with $0 < i < j < 15$). As this is defined per slave port, the same static QoS of requests arriving via different slave interfaces will be assigned to different priorities.
- **AXQOS-DDRCreg**. On every port, AXI requests are mapped to regions based on AXQOS, i.e. those lower than a threshold are mapped to region0 and the rest to region1.
- **DDRCreg-DDRCtc**. In each DDRC port, one traffic class (HPR, LPR, VPR) can be assigned to read channel in region0/region1 and one traffic class (NPW/VPW) can be associated to write channel in region0/region1.
- **AXQOS-DDRCtc**. It combines the previous two. For the read channel the static QoS (AXQOS) values are mapped to region0/1, which are then mapped to HPR/LPR/VPR traffic classes. For the write channel, also the static QoS (AXQOS) values are mapped to region0/1, which are mapped to either NPW or VPW.

There is no explicit mapping for AXQOS-QVN, CCIrq-QVN, DDRCreg-QVN, and DDRCtc-QVN, as we capture later in this section. As a result, if both QoS domains in those pairs are activated, different QoS features could be working towards opposing objectives, thus defying the potential benefits of hardware support for QoS.

## 4.2 Incompatible QoS features and Incongruous QoS Values

We have detected several QoS features, either in the same or different QoS domains, that are simply incompatible given their nature. As a result, simultaneously enabling them can result in unknown results in terms of predictability and isolation.

- **INCOMP1** Arm's dynamic QoS mechanisms transaction rate and transaction latency are incompatible with the QoS mechanisms in the DDR controller by Synopsis. Both QoS-400 and CCI-400 implement these dynamic QoS mechanisms. The source of the problem

[3] As explained later in this section regions help mapping static QoS, which ranges from 0 to 15, and Traffic Classes (low priority, high priority, and video).

lies in that these mechanisms change per-request static QoS priorities dynamically by overwriting static priority (AXQOS) settings. Hence, the hardware, without any software guidance, determines the QoS value of each request. This confronts with the use made of static QoS priority to split requests into classes or groups in the DDR controller: a given flow of requests that leaves the master with a given static QoS value can arrive at the target – after crossing a dynamic QoS mechanism – with requests having different and variable priorities. Despite a QoS range register controls the range of variation allowed for the dynamic QoS mechanisms, for this feature to be effective, the range must be so that the requests to be prioritized get higher priority than requests from other flows. Otherwise, dynamic QoS would have no effect. The net result, however, is that requests from the flow being prioritized can arbitrarily take different static QoS values, and hence they can be mapped to any region and traffic class in the memory controller. This makes dynamic QoS and the memory controller QoS fundamentally incompatible.

- **INCOMP2** The QoS relay for an IP block in the path from a master to a destination can overwrite the QoS set by the master. This can be done either with an IP integration time value in the QoS-400/CCI-400 block or a configurable value set in the control registers causing that all mappings and prioritization based on AXQOS can be lost regardless of the QoS set by the master. When the QoS value is hardwired at IP integration time, it can effectively become an incompatible feature with other QoS mechanisms that vary AXQOS values. Instead, when configurable via a control register, it becomes a feature to be properly set to avoid incongruities.

For compatible QoS features, there are a set of mutually incongruous QoS configurations whose combined effect can heavily affect or even cancel out the expected QoS behavior. This, in turn, can prevent achieving an overall predictability goal.

- **INCONG1** The lack of explicit mapping for AXQOS-QVN, CCIrq-QVN, DDRCreg-QVN, and DDRCtc-QVN makes that requests arriving at the CCI can have high AXI QoS priority while being assigned to a low-priority virtual channel (and vice versa). Likewise, requests from different sources going to the CCI can be mapped to different VNs; however, they can be mapped to the same CAMs in the DDRC so one flow with lower VN priority can stall the other, as the VN control is done at the port (XPI) level.
- **INCONG2** Traffic class in ports and channels. When the number of entries for HPR/LPR in the read CAM is below a HPR/LPR threshold, low-latency/best-effort ports (respectively) can be throttled. Likewise, when write CAM entries for NPW is below a threshold, best-effort ports can be throttled. However, nothing prevents ports to be setup as video while they issue HPR/LPR/NPW requests, causing CAM-based port throttling not to achieve its expected effect.
- **INCONG3** On arrival to a CCI slave port, read requests can be assigned few read queue entries (e.g. they are assigned to the low priority), while they are prioritized with QVN.
- **INCONG4** In the DDR controller, requests arriving via the two ports connected to the CCI, can be mapped to VPR/HPR hence being prioritized, while on the CCI the same requests can be assigned a low priority in the read queue, which will ultimately result in a low priority assignation.

## 4.3 QoS-Enabled IP Block Instantiation

The descriptions of the QoS features of each IP block in Section 3 come from IP providers and are agnostic to the particular instantiation of the IP block on a specific SoC. Those IP blocks have configuration options to be fixed at integration time, which hence are not described in

**Table 2** Main QoS points in the path from the APU/RPU to the DDRC/OCM.

| Type | IDs | Description |
|---|---|---|
| QoS-400 | 2, 3 | Prioritizes requests from PL ports HP1 and HP2 |
| QoS-400 | 12, 13, 14 | Prioritizes requests from HP0 (mem. port P3), HP1-HP2 (P4), & HP3 (P5) |
| QoS-400 | 30, 32, 33 | Prioritizes requests from the APU, HPC0-HPC1, & ACE |
| QoS-400 | 23, 26, 27 | Prioritizes requests from the 2 RPUs and the FPDswitch to the OCM |
| QVN-400 | QVN1/2 | Prioritizes requests to the DDR that pass through the CCI |
| CCI-400 | CCI-400 | Handles requests traversing the CCI-400 |
| DDRC | MC-QoS | Handles ports, CAMs, and other QoS mechanisms in the mem. controller |

the IP provider information. Unfortunately, nor they are in the Xilinx documentation [59]. This section describes the instantiation of QoS-enabled IP blocks in the ZUS+, capitalizing on the missing (unknown) information and observed limitations in QoS control.

There are more than 30 QoS points in the ZUS+. Table 2 lists those related to the access to the OCM and DDR from the APU, RPU, and PL. The first four rows correspond to static QoS mechanisms. QoS points based on AXI QoS are identified with numbers in the Figure 1, with values in light grey showing QoS-400 points that do not control the access to the DDR/OCM from the APU/RPU/PL and hence we do not cover. Static QoS points are referred to as “QoSpi” in the text where “i” is the QoS point id. For instance, QoSp8 controls the traffic generated from the display port and QoSp9 the traffic from FPD DMA. The types of QoS-enabled IP blocks are identified as “CCI-400”, “QVN” (QoS virtual networks), and “MC-QoS” (DDRC with QoS from Synopsis).

**QoS missing information.** A subset of the QoS features of some IP blocks are to be fixed at IP integration time by the integrator (Xilinx). However, several of these decisions are not described in Xilinx documentation and hence must be assessed empirically, as we do in Section 5 for the first two below.

- **UNKN01** There is no control register to select the behavior of QoS relay feature for NIC-400 second-level switches, that is, all switches but the FPD switch, the OCM switch, and the LPD switch. Nor is it documented whether there is some default behavior.
- **UNKN02** AXI3 FIFO queues are used to connect the PL with the Processing System (PS) and dealing with the clock and power region conversion. These FIFOs are 16-entry deep and independent for reads and write transactions. The implementation is AXI3 compliant and hence does not provide some of the AXI4 protocol signals, like the QoS signals. The ZUS+ documentation does not clarify whether and how the requests from the PL to memory keep the static QoS set in the PL ports. However, the field `FABRIC_QOS_EN` in the registers `RDCTRL` and `WRCTRL` in the `AFIFM` module seems to control this feature.
- **UNKN03** The CCI-400 provides no feature to control the number of slots to reserve to high and medium priority requests in the read queue of each slave. We conclude that either this feature is not implemented or the split of the queue is carried out with default, not controllable values. In any case, it is not a configurable QoS feature.

**QoS limitations.** From the instantiation of QoS-enabled blocks in the ZUS+ we derive the following limitation.

- **LIMIT01** All requests from the four A53 cores are routed via the only port between the APU and the CCI. Hence, the same QoS is assigned to requests from all 4 cores. QoSp32 helps controlling the aggregated traffic from all cores but not per-core traffic.

**Table 3** QoS features analyzed and fixed to deal with inconsistencies and incongruities.

| Feature | Description |
|---|---|
| (1) Static QoS | Enabled. All requests in the same flow have the same static QoS. |
| (2) Dynamic QoS | Disabled as it is incompatible with the QoS domains in the DDRC (INCOMP1) |
| (3) Outst. Transact. | Enabled |
| (4) QVN | Disabled as it was not possible to relate it to other QoS domains: AXQOS, DDRC, ... (INCONG1 and INCONG3) |
| (5) CCI read queue | It is not configurable. It is either not implemented or configurable (preventing INCONG4, and UNKN02) |
| (6) Urgent Port | Disabled not to override traffic class prioritization |
| (7) DDRC QoS | Enabled as it is central to achieve predictability goals. The particular parameters used are described later in Section 6 (Table 5). |
| (8) Traffic Class in ports & channels | We keep the same traffic class in the read/write channels and keep it congruent with the port type. We use: (VR/VW,V), (HPR/NPW,LL), (LPW/NPW,BE). |
| (9) Command reordering | DRAM command reordering is limited to the minimum value (4) to limit the impact on predictability |
| (10) CAM exhaustion | Fixed to the default value in the Xilinx provided setup |

The same limitation has been identified for other NXP SoCs integrating Arm IPs [54]. In this work, we show how such limitation can be pragmatically overcome through other routing mechanism since there are two ports (P0 and P1) the A53 can use to access the DDR.

## 4.4 Putting it All Together: Key Insights of the Analysis

The main outcomes of the analysis performed in this section relate to:

1. the particular QoS setups that make sense to experimentally evaluate, i.e. for which it has not been determined that they are fundamentally incompatible;
2. the range of values to prevent incongruities in the expected QoS behavior;
3. a set of QoS mechanisms that require empirical evidence to be validated/rejected as, from the analysis, it was not possible to determine the particular setup (values) selected in their instantiation in the ZUS+; and
4. a set of QoS-related open challenges that cannot be solved from the analysis, e.g. dealing with the fact that all A53 cores share a single port to the CCI (QoSp32 in Figure 1).

A decision we take for this work is to set static QoS priorities at the level of requests flows, see (1) in Table 3. For instance, we keep the same QoS for all requests from a source like the APU to the destination like the DDRC. This is in contrast to changing static QoS at the request level that, although possible, it would heavily complicate modeling and characterization usually performed in real-time systems. We also disable the urgent feature as it disruptively overwrites the nominal behavior based on traffic classes (6).

We discard for our evaluation the dynamic QoS features transaction rate and latency (2) in the CCI-400, NIC-400 as we conclude they are incompatible with the QoS features in the DDR, hence preventing INCOMP1 from arising. We analyze the outstanding transaction (3) dynamic QoS feature, but we conclude it provides limited benefits as the R5 cores are in-order and hence allow one in-flight load/store [10] and the A53 [7] ones allow a maximum of 3 loads in flight. The QVN feature (4) is disabled as we cannot map it to other QoS domains, which can have unexpected results, affecting the predictability/isolation goals (effectively preventing INCONG1 and INCONG3). The CCIrq feature is not configurable or not implemented (5), so we cannot set incongruous QoS values for it, preventing INCONG3 and INCONG4. The multi-layer arbitration in the DDRC is evaluated maintaining the type

**Table 4** Routing in the CCI.

| Setup | APM4.1 | | APM4.2 | |
|---|---|---|---|---|
| | ReadTC | WriteTC | ReadTC | WriteTC |
| **Default** | 64 | 64 | 64 | 64 |
| **ForceP1** | 128 | 128 | 0 | 0 |
| **ForceP2** | 0 | 0 | 128 | 128 |

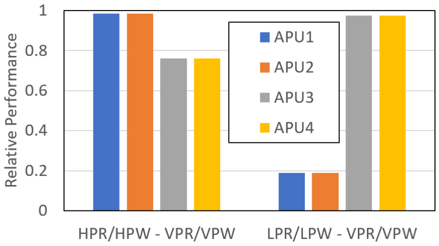


**Figure 5** QoS among APU cores.

of port and QoS-traffic class mapping per port congruent (8) preventing INCONG2; fixing reordering to its minimum value of 4 (9); and using the default CAM exhaustion (critical) mechanism.

The features to be empirically assessed include how to provide different QoS to different A53 cores (LIMIT1), and the determination of the QoS relay mechanism in second level switches (UNKN01 and INCOMP2) in the AXI3 FIFO queues (UNKN02).

## 5 QoS mechanisms characterization

Next, we characterize some QoS mechanisms by addressing undocumented design choices made by Xilinx when instantiating Arm IP blocks, and the limitation of the distributed QoS mechanisms in the ZUS+ introduced in Section 4.3.

### 5.1 Experimental Environment

We perform experiments on a Xilinx ZCU102 board that is equipped with a Zynq Ultrascale EG+ MPSoC. We run no operating system (bare-metal) or any external code, except for the First Stage Boot Loader (FSBL) provided by Xilinx toolchain (Vitis-2019.2), reducing non-hardware sources of interference. In fact, when executing several time the same experiments, we observe negligible execution time variability.

We run a low-overhead software configurator and a software collector. The former configures at boot time and during operation more than 60 SoC registers controlling the operation of the distributed QoS mechanisms. The latter provides measurements from several internal counters, including A53 and R5 performance counters and counters in the AXI Performance Monitors (APM).

**Benchmarks.** In this section, we use a set of benchmarks that generate intense read/write traffic to the OCM/DDR from the R5 and A53 cores by missing in each core's cache(s). The PL has been customized using the Xilinx Vivado tool to synthesize HDL designs, integrate multiples IPs, and generate the platform bitstream. We build on the AXI Traffic Generators (ATG) provided by Xilinx to generate read/write traffic to stress the target slave devices (OCM and DDR). To that end, we instantiate one or several ATGs per PL port so that we can vary the intensity of the generated read/write traffic.

### 5.2 Unveiling QoS features in the ZUS+

We empirically unveil relevant undocumented QoS features in the ZUS+. The same features will be further exploited in Section 6 to support the deployment scenario in our case study.

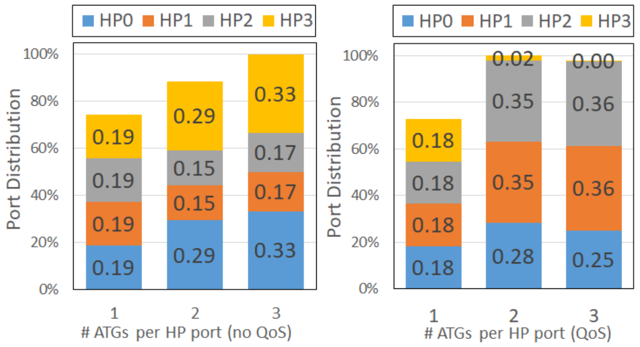


**Figure 6** DDR transaction distribution under the same and different QoS setups.

**A53 prioritization (LIMIT1).** As introduced in Section 4, the APU has a single port to the CCI that acts as the master for all requests from all four A53 cores to the CCI. This, in theory, prevents different QoS for the A53 cores, only allowing controlling their aggregated traffic. This challenges the use of the ZUS+ in critical systems since all applications in the APU are forced to have the same priority.

We circumvent this limitation by exploiting a characteristic that we have discovered empirically in our default configuration: while the traffic from the APU to the DDR uses ports P1 and P2 of the DDR, addresses in the same 8KB boundary are mapped to the same DDR controller port (P1 or P2), with P1 and P2 8KB address segments interleaved. We validated this feature by developing a benchmark that performs 128,000 read accesses and 128,000 write accesses to addresses mapped to different 8KB regions. We used the monitoring counters in APM 4.1 and 4.2, see Figure 1. As shown in Table 4, in the default setup accesses evenly distribute on P1 and P2. If we force the benchmark to use 8KB chunks mapped to P1 (ForceP1) requests are sent only to P1. The same happens if we force the benchmark to use address regions mapped to P2 (ForceP2).

In order to assess whether we can achieve different service for two A53 cores, we run four copies of a read benchmark, each of which runs in a A53 core (APU1-4). The first two are mapped to P1 and the other two to P2 as described above. In this experiment, all 4 benchmarks miss systematically in all data cache levels, so interference occurs almost exclusively in the access to DDR memory, i.e. benchmarks suffer almost no extra L2 miss when run simultaneously. In a first experiment, we put traffic class for read/write requests on P1 and P2 as HPR/HPW and VPR/VPW, respectively, with the latter having a high timeout. In a second experiment, we put traffic class as LPR/LPW - VPR/VPW, respectively. As shown in Figure 5, for the former experiment (left bars) APU1-APU2 get high relative performance (execution time in the 4-core experiment vs. execution time when each pair of benchmarks runs in isolation). This occurs since APU3-APU4 get priority only when their timeout expires every 1024 cycles. In the latter experiment (right bars), APU1-APU2 first compete with APU3-APU4 with the same priority, and whenever the timeout of the latter expires, APU3-APU4 get prioritized. Overall, the APUs mapped to the same port get the same relative performance, whereas those in different ports can have different relative performance. This confirms that our solution combining routing and QoS can offer different predictability guarantees to two different A53 cores.

**PL priorities (UNKN01 and UNKN03).** In these experiments, we aim at confirming (i) that FIFO queues in the PL, which use AXI3, effectively forward the static QoS value we set in each PL port (UNKN03), and (ii) that the QoS relay approach in the second level switches, which connect the ports to the memory, effectively forwards the input's AXQOS provided by the master (UNKN01). To that end, we configure from 1 to 3 of ATGs per each PL port: HP0 (mapped to memory port P3), HP1 and HP2 (mapped to P4), and HP3 (mapped to P5). Increasing the number of ATGs per port also increases the traffic to the DDR until reaching saturation. The traffic from HP0 and the display port (not shown in Figure 1) go to the same switch; so does the traffic of HP1 and HP2; and the traffic of HP3 and the FPD DMA (not shown in Figure 1). In this experiment, we focus on QoSp2 and QoSp3 that control HP1 and HP2, respectively; and QoSp12, QoSp13, and QoSp14 that control the traffic from the three switches to the OCM or the DDR. In ports P3, P4, and P5 we map static QoS priorities 0-3 to region0 and 4-15 to region1. We also map region0 to LPR/NPW traffic class and region1 to VPR/VPW and enable a request timeout for VPX requests so that they get prioritized.

As it can be seen in Figure 6 (left plot), under the same QoS setup (0,0,0,0), as we increase the ATGs per port from 1 to 3, the bandwidth usage increases, achieving the expected bandwidth allocation for the latter scenario: even bandwidth distribution with 1/3 of the bandwidth for HP0 (memory port P3) and HP3 (memory port P5) and 1/6 for HP1 and HP2 as they share the same port to memory (P4). Figure 6 (right plot) shows results for the setup (3,7,7,3), i.e. lower priority for HP0 and HP3. For 1 ATG per port, we see no impact of the QoS mechanism as each ATG can send as many requests per unit of time as in an isolation setup. With 2 or 3 ATGs per port, we see how effectively HP1 and HP2 get more bandwidth than HP0 and HP3. Both tasks contending for the central DDR get most of the bandwidth ($\tilde{3}5\%$ each), which matches their maximum bandwidth usage when run in isolation. For 2 and 3 ATGs per port, we also see that the ports with lower priority, HP0/P3 and HP3/P5, enjoy an uneven bandwidth despite both receive the same type of traffic and the configuration for both ports is the same. Our hypothesis is that HP1+2/P4 improves its performance due to a change of region from NPW to VPW. In contrast, HP0/P3 and HP3/P5 get unbalanced traffic due to the round-robin arbiter, which seems to arbitrate HP0/P3 before HP3/P5 and by the time it has to grant access to HP3/P5 a request in HP1+2/P4 gains higher priority, hence delaying HP3/P5 requests systematically.

**APU and RPU to OCM.** In our deployment scenario (Section 6), the APU and the RPU issue read/writes requests to the OCM to handle control variables. While this is unlikely to cause performance issues, we empirically show the impact of sharing the OCM and the potential benefits of using QoS hardware support to control it. In this experiment, the APU and RPU perform transactions to the OCM. RPU1 and RPU2 are first prioritized in the RPU switch (QoSp26 and QoSp27) and the request winning that arbitration competes with the requests arriving from the APU – when active – in the OCM switch (QoSp23).

The left chart in Figure 7 shows that, for reads, the APU suffers a maximum slowdown of 1.03x (i.e. 3%) due to the contention in the OCM, whereas RPU1/RPU2 suffer no slowdown. This occurs because the R5 [10] implements an in-order pipeline and the A53 [7] allows at most 3 loads in flight. Hence, since tasks run almost as in isolation, QoS has no room for improvement. For writes (right chart), when running the two RPUs alone without the APU (referred to as RPUx2), we observe that RPUs do not generate enough pressure on the OCM, as for loads. When adding the APU (RPUx2,APU), the APU suffers a 1.5x slowdown. This occurs because A53 cores are out-of-order cores that, thanks to the use of store/write buffers, support in-flight write requests, increasing the pressure on the target. However, this also makes APU's high-frequency write requests to be more sensitive to contention. Increasing the static priority of any of the RPUs, setups (7,0,0) and (0,7,0), reduces the slowdown

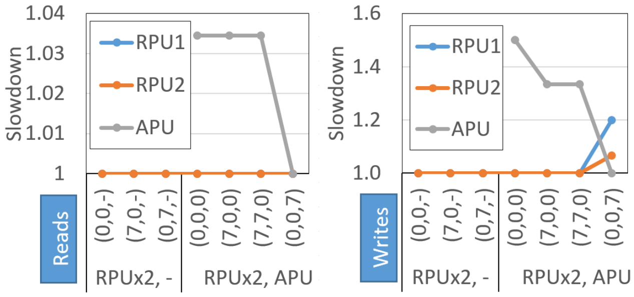


**Figure 7** Impact of static QoS when the RPU/APU target the OCM.

on the APU down to 1.3x. When both RPUs have low priority, (0,0,7) the APU reduces its slowdown to zero (1.0x). This also causes a non-homogeneous impact on RPU1/RPU2, suffering a slowdown of 1.2x and 1.05x, respectively. As before, it seems that RPU1 and RPU2 are arbitrated using a round-robin arbiter that arbitrates RPU2 before RPU1 after APU accesses are served, and since the access patterns repeat, this small difference magnifies and leads to those different slowdowns for each RPU.

Overall, despite some contention can occur in some corner situations (RPUs and APUs making writes to the OCM), the OCM is not a bottleneck in our deployment scenario as it is used mainly for control/synchronization variables. Hence, the potential slowdown is minimum and no QoS mechanism is needed to control contention.

**Summary.** We unveiled how to combine routing and QoS so that up to two A53 cores can be provided different QoS service. We also showed that FIFO queues in the PL and the second-level switches relay the static QoS received from the master starting the transaction. Finally, we showed that QoS is not needed for the OCM in our deployment scenario.

## 6 Case Study: IFC selection

In this section we build on the analysis and characterization in previous sections to show the benefits of the hardware QoS support of the ZUS+ to increase the chances of finding valid platform setups. In avionics, this is referred to as selection of the intended final configuration (IFC) in CAST-32A [18]. In our case, the IFC includes the setting of QoS mechanisms so that time constraints of each process are met as required by CAST-32A.

**Deployment Scenario.** We address a deployment scenario in which the MPSoC is configured to host a set of mixed-criticality applications, organized into several software partitions (SWP). Such scenario is representative, for example, of multicore partitioned systems in the avionics domain [3, 41, 18]. Depending on the specific resource and time partitioning approach, SWPs may be allowed to execute in one or multiple computing elements, either exclusively or in parallel with other partitions. In this respect, we focus on a relatively flexible, performance-oriented deployment configuration where three SWPs are executed in parallel on the ZUS+. Each software partition comprises several processes that execute in the RPU, APU and PL. The OCM is used for exchanging control data while the DDR is used as main memory for sharing compute data. SWP1 runs one process on a R5 core, another in a A53 core, and uses the PL for acceleration. We refer to them as RPU1, APU1, and PL1, respectively. The processes of SWP2 are mapped in the same manner and are referred to as RP2, APU2, and PL2. Finally, SWP3 basically runs on the PL (PL3) though it has a small control process that runs on an A53 core.

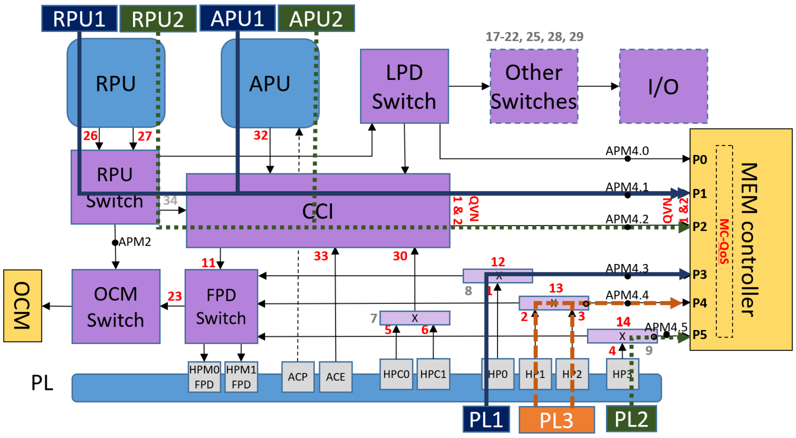


**Figure 8** Routing and port mapping in our deployment scenario.

**Providing Guaranteed Service.** The specific QoS configuration is meant to meet the diverse predictability requirements of each SWPs. We focus on three hierarchically ordered predictability goals. First, provide guaranteed service to SWP1, i.e. preventing that SWP1 receives no service due to the load generated by other SWPs. Second, provide guaranteed service to SWP2 as long as SWP1 leaves enough resources to that end. And third, in all scenarios SWP3 is provided best effort (average performance centric setup). We achieve the required guarantees by deploying QoS setups with specific values fixed for some QoS features while factoring in the outcome of the analysis (Table 3).

1. The traffic of different SWPs does not share the same memory port. As it can be seen in Figure 8, APU1/RPU1 share memory port P1 and PL1 uses P3 (solid blue line); APU2/RPU2 share P2 and PL2 uses P5 (green dotted line), whereas PL3 uses P4. Note that, as PL3 is assumed to be a number crunching accelerator, it uses two ports in the PL (HP1 and HP2) to support more traffic from/to memory.
2. Requests from SWP1 have the highest static QoS or the same as SWP2 (in the latter case, round-robin is used for arbitration, ensuring that both SWPs get service).
3. We map SWP1 requests to video traffic class (VPR and VPW) and all the ports it uses, P1 and P3, are also set as video traffic class. For SWP2, we use high priority traffic class for reads/writes (HPR/NPW) and low-latency type for P2 and P5. SWP3 is mapped to the low priority traffic class and the port it uses, namely P4, is set as best effort.

Under this set of constraints in the QoS setup, SWP1 requests have the highest priority when their associated timeout expires. When they are not expired, SWP2 requests have the highest priority. The values for other QoS parameters can be varied to adjust the service provided to the needs of the particular processes. This includes the following, see Table 5: The number of entries in the CAMs for each traffic class: (i) high-priority and low-priority thresholds for the read CAM and the (ii) normal priority threshold for the write CAM. (iii) The timeout for VPR/VPW traffic class on each port that can be increased when tasks have low utilization, i.e. the ratio between their execution time and deadline is low and vice versa. (iv) The traffic class of port/channels. (v) The static QoS of APU/RPU in each SWP. While they both remain mapped to the same traffic class, we can assign either APUi or RPUi higher static QoS to adjust their latency as needed. We explore 3 different QoS values to provide three different prioritization levels. In particular, we use QoS values 3, 7, and 10. Any other three different values can be used. And (vi) the outstanding transactions.

**Table 5** QoS values explored in this work.

| Feature | Description |
|---|---|
| read CAM | High/Low priority threshold [0, 1, 2, ..., 32][0, 1, 2, ..., 32] |
| write CAM | Normal priority threshold [0, 1, 2, ... , 32] |
| Timeout | [1, 8, 16, 32... 1024] |
| Traffic Class | 3 classes available for each channel/port |
| Channels/Ports | SWP1 always at highest priorities and SWP3 at relative lowest ones |
| Static QoS | 3 values so that SWP1 has the highest priority and SWP3 the lowest |
| OT | Outstanding Transactions: 4-16 |

**Kernels.** We create several workloads from kernels used in many applications in critical systems. These kernels, which run in the APU and the RPU, are: (i) Matrix Multiplication (`MM`) is one of the most common kernels for many functionalities like object detection or path planning in autonomous navigation [4]; (ii) Matrix Transpose (`MT`) is another quite common matrix operator and often used along with MM; (iii) Rectifier (`ReLu`) is an activation function in neural networks defined as the positive value of its argument; (iv) the Image-to-Columns (I2C) function for transforming raw RGB images into matrices in the format needed by neural networks; and (v) vector-multiply-add (`VMA`) that is a type of linear algebra operator. In the PL we run several instances of the ATG performing reads or write bursted transactions (`ATGr` and `ATGw`) to match burst-oriented accelerators transfers. PL1 instantiates 1 ATG, PL2 2 ATGs, and PL3 4 ATGs to generate asymmetric traffic demands.

We focus on the setup presented above with three SWPs. We compose several workloads from the kernels: WRKLD1 `(MM,VMA,ATGr)(MT,I2C,ATGr)(ATGr)` that runs MM, VAM as APU1 and RPU1 respectively; MT and I2C as APU2 and RPU2, respectively; and `ATGr` used as PL1/2/3; and WRKLD2 `(MM,I2C,ATGw)(ReLu,I2C,ATGw)(ATGw)`. When creating a workload, we allocate memory of these kernels properly to ensure they use either P1 or P2, see Section 5.2. Also, note that these workloads put high pressure on DDR memory, with 7 ATGs in the PL (1, 2, and 4 respectively instantiated for PL1, PL2, PL3), 2 A53 cores, and 2 R5 cores sending requests simultaneously to the DDR memory system.

## 6.1 Malleability

We start assessing the *malleability* of the QoS mechanisms in the ZUS+ for several workloads. Malleability measures whether the used QoS setups effectively bias the execution of those tasks with higher priority, though this causes the other tasks to suffer more contention interference. Without this property, the use of QoS would be ineffective. For two different workloads Figure 9 reports the relative performance of each process with respect to the scenario in which its SWP runs in isolation. A relative performance of X% means a slowdown of (100/X), e.g. 50% relative performance means 2x slowdown. In particular, Figure 9 shows the impact of changing the timeout for video requests (VPR/VPW) when both SWP1 and SWP2 are mapped to the video traffic class, while SWP3 is mapped to the low-priority class. As we decrease the timeout of all video ports from 1024 by half until reaching 2, the performance of SWP1/SWP2 (RPU1/APU1/PL1 and RPU2/APU2/PL2) processes increases at a similar pace, while PL3 relative performance sharply decreases when the timeout goes from 1024 to 256 and remains around 10% for lower VPR/VPW timeout values.

[4] Matrix multiplication is the central part of machine learning libraries like YOLOv3 [56] and account for 67% of YOLO's execution time [25].

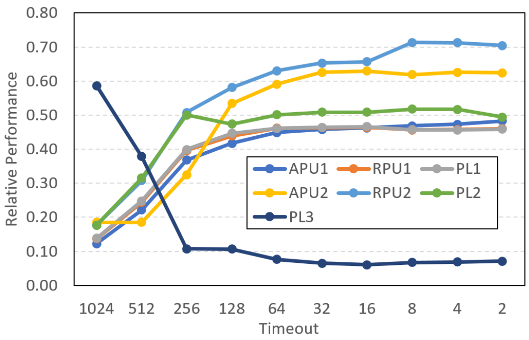


**Figure 9** Impact of changing the duration of the timeout period

.

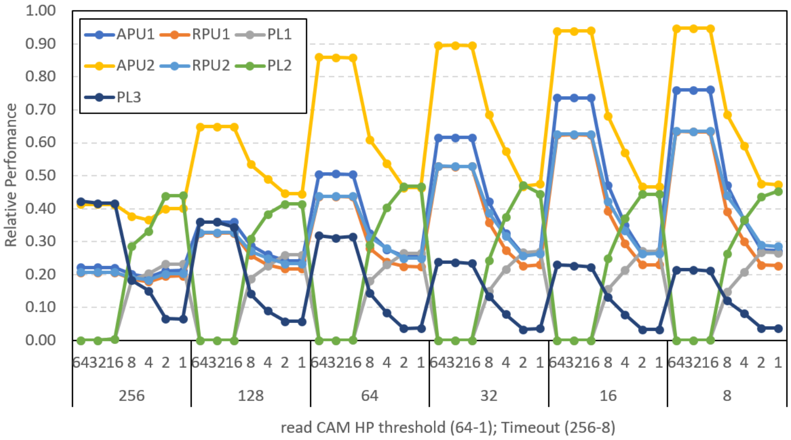


**Figure 10** Malleability for different QoS setups varying read CAM entries for HPR and timeout.

Figure 10 captures a scenario in which the processes of the workload vary their relative performance under the combined effect of decreasing the timeout and decreasing the number of read CAM entries for HPR (where SWP3 is mapped to). For each group of 7 configurations, we see increased performance for all computing units except PL3 when decreasing CAMs. Timeouts for VPR/VPW, on which SWP1/SWP2 are mapped to, decrease across 7-setup groups from left to right, bringing the combined effect in which SWP1/SWP2 relative performances increase within each 7-setup group and across groups. In both figures, we see how QoS in the ZUS+ achieves both (1) a good range of variation in the relative performance of the processes; and (2) smooth variations in relative performance across different QoS setups. These are fundamental traits for malleability and the main building block in our study.

## 6.2 QoS for Improved Platform Setup

In this section, we explore over 30,000 different QoS setups that (i) already factor in the outcome of our analysis (see Table 3), and (ii) provide guaranteed service to SWP1, also to SWP2 if there are enough resources left by SWP1 to achieve it, while SWP3 receives best-effort service. The values for the rest of the QoS parameters are explored to adapt to the timing constraints of the different tasks, as summarized in Table 5. The difference between guarantees and real-time requirements is better explained with an example. For some scenarios RPU1, which receives guaranteed service, might have a loose deadline so it requires achieving reduced, yet guaranteed, relative performance (e.g. 20%); while in others RPU1 has a tight deadline requiring high relative performance (e.g. 80%).

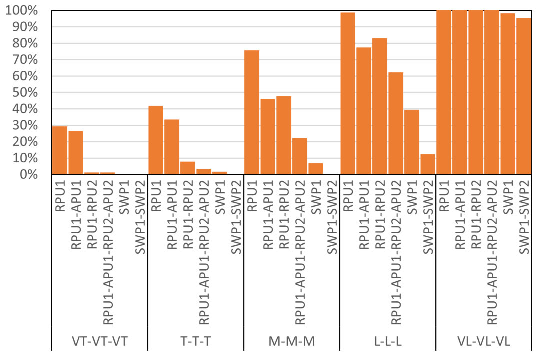


**Figure 11** Ratio of accepted QoS setups with uniform thresholds for Workload 1.

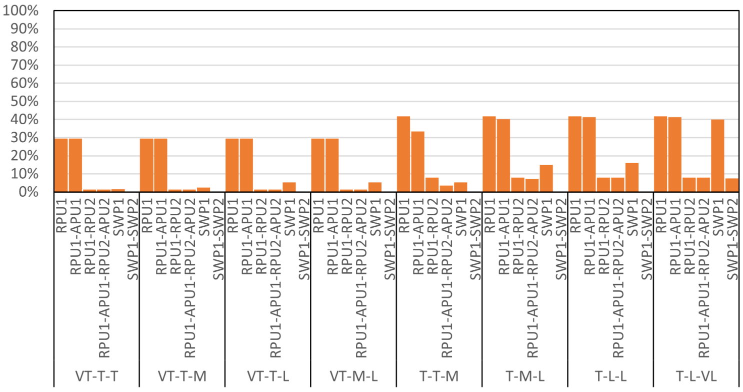


**Figure 12** Ratio of accepted QoS setups with heterogeneous thresholds for Workload 1.

We set different thresholds for the maximum slowdown admissible for the processes in a SWP with respect to the performance obtained when the SWP executes in isolation. We explored 5 minimum relative performance scenarios: `VeryTight or VT` (80%) `Tight or T` (60%), `Moderate or M` (40%), `Loose or L` (20%), and `VeryLoose or VL` (1%). Note that `VL` allows an almost unbounded performance degradation. These thresholds can be set homogeneously for all processes in a SWP or heterogeneously, e.g. (`VT`, `T`, `L`) meaning that all involved APUs, RPUs, and PLs can sustain different maximum performance degradation.

**Workload 1.** Figure 11 summarizes the ratio of accepted QoS setups for WRKLD1, when uniform thresholds are applied across computing elements, in the set of experiments. An accepted QoS setup meets the performance thresholds considered for the specific scenario. The cutoff criteria are applied to different subsets of computing elements, corresponding to the scenario where performance guarantees are extended from cores (e.g. APU1 only, referred to as APU1) to SWP1 (that includes RPU1/APU1/PL1) and SWP1-SWP2 that sets the `VT/T/M/L/VL` in all processes (RPU1/APU1/PL1 and RPU2/APU2/PL2).

Even under the tightest constraints, `VeryTight` (set of bars `VT-VT-VT`), around 30% of the QoS setups meet the constraints for RPU1. If constraints are also to be met for APU1 (RPU1-APU1), still 26% of the QoS setups are accepted. If RPU2 constraints are considered (RPU1-RPU2), 1.3% of the QoS setups are successful, and 1.1% if APU1 and APU2 constraints also need being met (RPU1-APU1-RPU2-APU2). When considering the PLs, only 1 setup (0.003%) meets all SWP1 constraints, and none SWP1 and SWP2 constraints simultaneously. If we relax the constraints (from `VT` to `T`, `M`, `L` and `VL`), the fraction

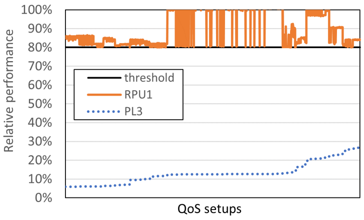


**Figure 13** $RPU1 > 80\%$.

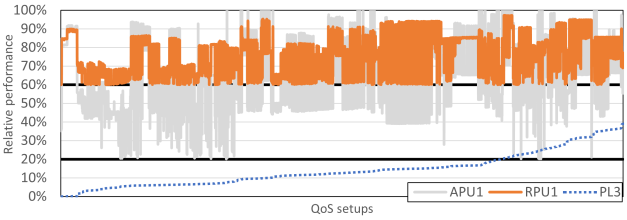


**Figure 14** $RPU1 > 60\%$ and $APU1 > 20\%$.

of successful QoS setups increases in all cases except when SWP1 and SWP2 constraints need to be met simultaneously, which is only doable with loose (`L`) or very loose (`VL`) constraints for 12% and 95% of the QoS setups respectively. This occurs because ATGs in the PL are highly bandwidth demanding and hence, under those QoS setups where one ATG gets higher priority than another computing unit systematically, the latter can experience starvation.

Figure 12 analyzes heterogeneous scenarios where RPU constraints are `VT/T`, and APU and PL constraints are relaxed. For instance, the second group of columns, (`VT-T-M`) imposes `VT` constraints on RPUs, `T` on APUs and `M` on PLs. If we compare each group w.r.t. their homogeneous counterpart with identical RPU constraints (e.g. four leftmost groups of bars `VT - T/M - T/M/L` in Figure 12 and `VT-VT-VT` in Figure 11), the fraction of successful QoS setups increases. This shows that we can exploit QoS to accommodate the timing requirements of the different processes. Similar conclusions are achieved comparing the four right-most groups of bars in Figure 12 and `T-T-T` in Figure 11, with the number of successful QoS setups increasing as the time constraints on some computing elements relax.

The presence of several accepted QoS setups in every configuration offers the possibility of satisfying different timing requirements. This, in turn, enables the system developer to apply and optimize any relevant metric to the set of valid QoS setups. As an illustrative example, Figure 13 shows how it is possible to meet the stringent performance constraints of a critical SWP while still maximizing the throughput of best effort functionalities. Specifically, Figure 13 considers QoS setups selected to preserve 80% of the reference performance for the critical software mapped to RPU1 in WRKLD1, and orders them according to the performance guaranteed for the best effort functions deployed to SWP3 (i.e., PL3). We see how the performance exhibited by PL3 under a conservative setting for RPU1 still covers a wide range of values, while the relative performance of RPU1 is always above the threshold (80%) represented with a black horizontal line. Even for a larger sets of constraints, a non-negligible number of QoS setups is able to meet them, offering optimization options. For instance, in Figure 14 we set the constraint that RPU1 relative performance must be above 60% and APU1 above 20%. In this scenario, PL3 shows a range of variation of around 40 percentage points. Overall, we see how smartly deploying hardware QoS support allows the system designer to optimize different metrics, while fulfilling timing requirements.

**Workload 2.** Figures 15 and 16 show results for WRKLD2. We observe the same trends as those for WRKLD1. The most significant difference is that WRKLD1 includes ATGr (so intensive PL read operations), whereas WRKLD2 includes ATGw (so intensive PL write operations) for both SWP1 and SWP2. The first consequence is that RPU and APU kernels are successful in a much larger fraction of setups, as kernels running in the RPUs and APUs are more sensitive to interference in their read operations than in their write operations (e.g. writes tolerate delays in store buffers), and DDR channels for read and write operations are decoupled to a large extent. Hence, ATGr in WRKLD1 creates much higher interference than ATGw in WRKLD2 on RPU and APU kernels, and therefore, the first four columns for

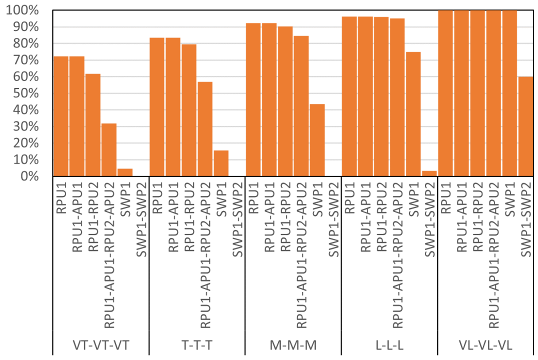


**Figure 15** Ratio of accepted QoS setups with uniform thresholds for WRKLD2.

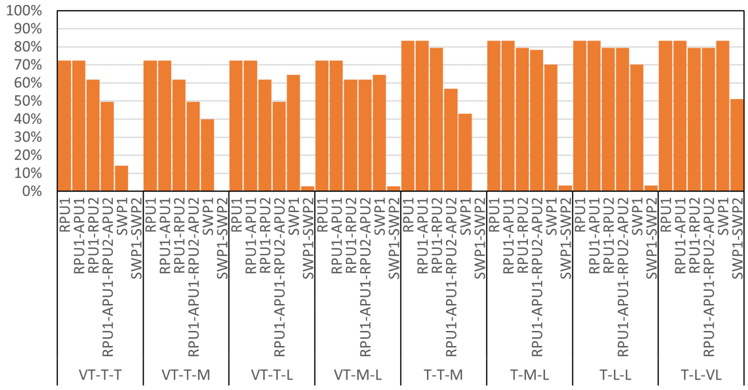


**Figure 16** Ratio of accepted QoS setups with heterogeneous thresholds for WRKLD2.

each set of constraints has a much larger fraction of successful QoS setups when compared with WRKLD1. On the other hand, when PLs are considered, the fraction of successful QoS setups for SWP1 only, or both SWP1 and SWP2, is much larger for WRKLD2. For instance, `VT-M-L` has 5.4% and 0% successful setups for SWP1 and SWP1+SWP2 respectively for WRKLD1, and 64.5% and 2.7% for WRKLD2.

## 7 Conclusions and Future Work

Hardware support for QoS is increasingly becoming a seamless technology. In a MPSoC this will be realized by a distributed QoS mechanism with QoS-enabled IP blocks (likely) coming from different providers, which calls for mechanisms to orchestrate them. In this work, we analyzed the nominal behavior of individual QoS mechanisms in the Xilinx Zynq UltraScale+ MPSoC as well as their combined behavior. We capitalize on their combined behavior including incompatible mechanisms, compatible mechanisms under specific setups, and limitations. We empirically show how to circumvent some of the limitations (e.g. using routing to allow several A53 cores to have different QoS) and provide insights on unknown features (e.g. QoS relay mechanism in second-level switches). Building on the gained knowledge, we expose a wide set of QoS setups that help providing guarantees to certain processes while allowing adapting to processes timing constraints. Indeed, we show that the QoS mechanisms in the Zynq UltraScale+ are very powerful and can successfully adapt to different constraints, offering great flexibility to the system designer to optimize the system configuration along different metrics.

From the analysis performed, it also follows that, in order to consolidate the use of QoS in critical domains, technical reference manuals should provide more focused information. In particular, IP integrators should better describe the options selected for each IP block instantiated. Also, clear examples describing how to coordinate several QoS mechanisms to achieve higher-level isolation and predictability goals will significantly reduce the effort of the software/system integrator in using hardware support for QoS.

In terms of future research directions we aim at formalizing a more generic process for orchestrating the QoS features in other MPSoCs. This includes (i) the identification of QoS domains; (ii) mapping of QoS domains; and (iii) finding compatible QoS features. We envision the definition of a set of QoS rules whose validation involves passing a set of (potentially automated) tests, assessing the validity of any QoS setup and its benefits towards achieving different isolation/predictability goals. This is in line with current practice in avionics and automotive that builds on formulating test designs to produce evidence that serves to accept or reject a hypothesis set over a specific functional or non-functional system behavior [48]. We also plan to develop more advanced search algorithms to make an efficient exploration of the QoS configuration space. Such algorithms are needed since, in the general case, with more complex workloads and different predictability constraints, the number of potential QoS setups is too large to allow an exhaustive space exploration.

## References

1 Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, November 2020. doi:10.1109/tcad.2020.3012210.

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 *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, June 2017. doi:10.1109/sies.2017.7993376.

3 ARINC. *Specification 653: Avionics Application Standard Software Interface.* Aeronautical Radio, Inc, 1996.

4 Arm. *ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual.*

5 Arm. *ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual.*

6 Arm. *ARM CoreLink QVN-400 Network Interconnect Advanced Quality of Service using Virtual Networks Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual.*

7 Arm. *ARM Cortex-A53 MPCore Processor Technical Reference Manual. Version r0p4.* URL: https://developer.arm.com/documentation/ddi0500/j/.

8 Arm. *Arm® Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A.*

9 Arm. *ARM® CoreLink™ CCI-400 Cache Coherent Interconnect. Revision: r1p3. Technical Reference Manual.*

10 Arm. *Cortex-R5 and Cortex-R5F Technical Reference Manual. Version r1p1.* URL: https://developer.arm.com/documentation/ddi0460/c/.

11 Arm. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. ARM IHI 0022E (ID033013)*, 2013.

12 Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nelis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, July 2016. doi:10.1109/ecrts.2016.14.

13 Matthias Becker, Borislav Nikolic, Dakshina Dasari, Benny Akesson, Vincent Nelis, Moris Behnam, and Thomas Nolte. Partitioning and analysis of the network-on-chip on a COTS many-core platform. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 101–112. IEEE, April 2017. doi:10.1109/rtas.2017.32.

14 Alessandro Biondi and Marco Di Natale. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 240–250. IEEE, April 2018. doi:10.1109/rtas.2018.00032.

15 Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, Chen-Yong Cher, and Mateo Valero. Software-controlled priority characterization of POWER5 processor. In *2008 International Symposium on Computer Architecture*, pages 415–426. IEEE, June 2008. doi:10.1109/isca.2008.8.

16 Jordi Cardona, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. Maximum-contention control unit (MCCU): resource access count and contention time enforcement. In *Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 710–715. IEEE, 2019. doi:10.23919/DATE.2019.8715155.

17 Jordi Cardona, Carles Hernandez, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. NoCo: ILP-based worst-case contention estimation for mesh real-time manycores. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 265–276. IEEE, December 2018. doi:10.1109/rtss.2018.00043.

18 Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.

19 Dakshina Dasari and Vincent Nelis. An analysis of the impact of bus contention on the WCET in multicores. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1450–1457. IEEE, June 2012. doi:10.1109/hpcc.2012.212.

20 Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3):272–322, May 2016. doi:10.1007/s11241-015-9229-9.

21 Dakshina Dasari, Borislav Nikolic, Vincent Nelis, and Stefan M. Petters. NoC contention analysis using a branch-and-prune algorithm. *ACM Transactions on Embedded Computing Systems*, 13(3s):113:1–113:26, March 2014. doi:10.1145/2567937.

22 Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the AURIX™ TC27x. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, June 2018. doi:10.1109/dac.2018.8465780.

23 Falk Rehm and Jörg Seitter. Software Mechanisms for Controlling QoS. In *2021 Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Virtual Conference, February 01-05, 2021*, pages 1485–1488, 2016.

24 Farzad Farshchi, Qijing Huang, and Heechul Yun. BRU: bandwidth regulation unit for real-time multicore processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375. IEEE, April 2020. doi:10.1109/RTAS48715.2020.00011.

25 Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176. IEEE, June 2017. doi:10.1109/dsn-w.2017.47.

26 Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. A sensitivity analysis of two worst-case delay computation methods for SpaceWire networks. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 47–56. IEEE, July 2012. doi:10.1109/ecrts.2012.35.

27 Freescale semicondutor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.

28 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys*, 48(2):32:1–32:36, 2015. `doi:10.1145/2830555`.

29 Mohamed Hassan and Rodolfo Pellizzoni. Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, November 2018. `doi:10.1109/tcad.2018.2857379`.

30 Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsocs. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ECRTS.2020.23`.

31 Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon. SMT QoS: Hardware Prototyping of Thread-level Performance Differentiation Mechanisms. In *HotPar 12*, Berkeley, CA, June 2012. USENIX Association.

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

33 Javier Jalle, Jaume Abella, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. AHRB: A high-performance time-composable AMBA AHB bus. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 225–236. IEEE, 2014. `doi:10.1109/rtas.2014.6926005`.

34 Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer-Verlag, 2001. `doi:10.1007/3-540-45318-0`.

35 Sunggu Lee. Real-time wormhole channels. *Journal Of Parallel And Distributed Computing*, 63(3):299–311, March 2003. `doi:10.1016/S0743-7315(02)00055-2`.

36 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):05–1–05:48, 2016. `doi:10.4230/LITES-v003-i001-a005`.

37 Kristiyan Manev, Anuj Vaishnav, and Dirk Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 179–187. IEEE, 2019. `doi:10.1109/ICFPT47387.2019.00029`.

38 Sparsh Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys*, 50(2):27:1–27:39, 2017. `doi:10.1145/3062394`.

39 Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008. `doi:10.1109/mm.2008.43`.

40 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 *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014. `doi:10.1109/ecrts.2014.20`.

41 Diniz Nuno and Jose Rufino. ARINC 653 in Space. In *DASIA - Data Systems in Aerospace*, ESA Special Publication, 2005.

42 nVIDIA. *Technical Reference Manual. Xavier Series SoC. DP-09253-002. Version 1.1*, 2018.

43 Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio C. Buttazzo. A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:24, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2019.24`.

44 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, April 2011. `doi:10.1109/rtas.2011.33`.

45 Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *2008 Real-Time Systems Symposium*, pages 221–231. IEEE, November 2008. doi:10.1109/rtss.2008.42.

46 Jon Pérez-Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys*, 53(4):79:1–79:38, 2020. doi:10.1145/3398665.

47 Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 44–53. IEEE Computer Society, 2009. doi:10.1109/nocs.2009.5071444.

48 David Radack, Harold Jr, and Paul Parkinson. Civil certification of multi-core processing systems in commercial avionics. In *2019 27th Safety-critical Systems Symposium*, February 2019.

49 Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*, 62(3):452–467, March 2013. doi:10.1109/tc.2011.240.

50 Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs. *ACM Trans. on Embedded Computer Systems*, 18(5s):51:1–51:22, 2019. doi:10.1145/3358183.

51 Shahin Roozkhosh and Renato Mancuso. The potential of programmable logic in the middle: Cache bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 296–309. IEEE, April 2020. doi:10.1109/rtas48715.2020.00006.

52 Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 759–764, 2010.

53 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. On How to Identify Cache Coherence: Case of the NXP QorIQ T4240. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2020.13.

54 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-WarP: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357. IEEE, December 2020. doi:10.1109/rtss49844.2020.00039.

55 Synopsis. *DesignWare Enhanced Universal DDR Memory Controller.*

56 Hamid Tabani, Roger Pujol, Jaume Abella, and Francisco J. Cazorla. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 144–145. IEEE, May 2020. doi:10.1109/isorc49007.2020.00030.

57 Sebastian Tobuschat and Rolf Ernst. Real-time communication analysis for networks-on-chip with backpressure. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 590–595. IEEE, March 2017. doi:10.23919/date.2017.7927055.

58 XILINX. Rockwell Collins Uses Zynq UltraScale+ RFSoC Devices in Revolutionizing How Arrays are Produced and Fielded: Powered by Xilinx, 2018. URL: https://www.xilinx.com/video/corporate/rockwell-collins-rfsoc-revolutionizing-how-arrays-are-produced.html.

59 XILINX. *Zynq UltraScale+ Device. Technical Reference Manual. UG1085 (v2.1)*, 2019.

60 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, April 2013. doi:10.1109/rtas.2013.6531079.

# Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications

Golsana Ghaemi ✉
Boston University, MA, USA

Dharmesh Tarapore ✉
Boston University, MA, USA

Renato Mancuso ✉
Boston University, MA, USA

## Abstract

There exists a divide between the ever-increasing demand for high-performance embedded systems and the availability of practical methodologies to understand the interplay of complex data-intensive applications with hardware memory resources. On the one hand, traditional static analysis approaches are seldomly applicable to latest-generation multi-core platforms due to a lack of accurate micro-architectural models. On the other hand, measurement-based methods only provide coarse-grained information about the end-to-end execution of a given real-time application.

In this paper, we describe a novel methodology, namely Black-Box Profiling (BBProf), to gather fine-grained insights on the usage of cache resources in applications of realistic complexity. The goal of our technique is to extract the relative importance of individual memory pages towards the overall temporal behavior of a target application. Importantly, BBProf does not require the semantics of the target application to be known – i.e., applications are treated as black-boxes – and it does not rely on any platform-specific hardware support. We provide an open-source full-system implementation and showcase how BBProf can be used to perform profile-driven cache management.

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

**Keywords and phrases** Cache Profiling, WSS Estimation, Cache Interference, Real-time, Multicore, Contention-induced Instruction Stall, C2IS, Coloring, Cache Management, Cacheability

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.4

**Supplementary Material**
*Software (Kernel Sources)*: `https://github.com/rntmancuso/linux-xlnx-prof`
archived at `swh:1:dir:995dd657183233e05f30f4d5755cca46e01dd7c5`
*Software (BU Black-box Profiler)*: `https://github.com/rntmancuso/black-box-profiler` [11]
archived at `swh:1:dir:2cc5a9264901e43157967138ac50a2700feb963c`

**Funding** *Renato Mancuso*: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. 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

The evolution of multi-core architectures and the ever-widening gap between the performance of processor and memory has rendered the adoption of system-level management strategies for shared memory resources a must. Indeed, inter-core interference is a fundamental challenge for the practical adoption of multi-core systems in safety-critical real-time applications, as extensively surveyed in [25]. In a nutshell, the problem of inter-core interference arises due to priority- and criticality-agnostic arbitration for the allocation of and access to shared memory components of application workload deployed in parallel on multiple cores. Important achievements have been accomplished by the research community in the design of practical memory management techniques to mitigate inter-core interference.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 4; pp. 4:1–4:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Unfortunately, however, the most widely used techniques rely on the enforcement of strict resource partitioning – e.g., shared cache space coloring [22], sustainable memory bandwidth partitioning [39, 37]. Often times, the rigidity of strict resource partitioning results in what is known as the *one-out-of-m* multi-core problem [19]. That is, the performance loss resulting from enacting strict partitioning outweighs its benefits. We argue that at the core of the problem is a fundamental lack of methodologies to analyze exactly *how* realistic, data-intensive applications interact with and benefit from the complex hierarchy of memory resources in modern high-performance embedded systems.

The goal of this paper is to provide one such methodology that goes under the name of *Black-Box Profiling*, or BBProf for short. Specifically, we propose a profiling strategy that can be used to accurately understand how an application's temporal behavior is affected by the presence/absence in the cache of individual memory pages. This sets our work apart from other profiling strategies that compute only end-to-end metrics such as the total cache hit/miss rate, number of bus accesses, resulting runtime when adopting a given resource partitioning scheme, and so on. The BBProf methodology is designed to operate without requiring a micro-architectural model, which is often unavailable (or just too complex) for high-performance systems. The proposed BBProf adopts a measurement-based approach that does not rely on any platform-specific hardware support, and can be ported to virtually any platform.

With this paper, we make the following contributions. First, we propose a novel profiling methodology that requires no special hardware support to produce insights about the relative importance of each memory page towards the overall timing of a target application. Second, we describe how said methodology can be applied to profile realistic, pre-compiled black-box applications without requiring any source-level or compile-time modifications. Third, we propose a proof-of-concept, open-source, full-system implementation and show its capability of profiling real-world vision applications. Fourth, we demonstrate that profile-driven shared cache management is enabled by our BBProf methodology and highlight its benefit in two scenarios: (1) to enact flexible interference mitigation with absolute guarantees that are comparable to strict partitioning; and (2) as an efficient solution to the previously undocumented problem of Contention-Induced Instruction Stall (C2IS).

## 2 Related Work

Research interest for workload-aware cache management has been spurred a large body of works targeting real-time systems and general-purpose systems alike. A number of works have proposed techniques to estimate the working-set size (WSS) of applications for the purpose of performing informed cache management. One such work is [6], where the WSS of a periodic application is estimated by computing the average per-activation number of cache misses. This information, albeit coarse, is proven useful to avoid concurrently scheduling applications with incompatible WSS. In a spirit quite similar to our BBProf, the work in [40] proposes a technique to detect *hot* memory pages and to dynamically perform re-coloring to improve average performance. Hot pages detection is performed by periodically scanning the *accessed-bit* in all the page-table entries that belong to the target application. This methodology, however, only provides an indirect estimation of the importance of each page that depends on the frequency of sampling. It also relies on the presence of the accessed-bit, which is an Intel-specific hardware feature. The work in [32] uses a similar approach that relies on PowerPC-specific sampled-address data registers (SDAR).

Several works [18, 16, 4] propose scheduling models where the balance between loss of performance due to smaller cache partitions and performance improvements thanks to reduced cache interference is studied. Generally, these model assume that certain intrinsic properties – e.g. their characteristic miss rates – of the applications under analysis are known. In this case, the BBProf methodology proposed hereby could be used to determine key behavioral parameters required to instantiate such and similar analytical frameworks. More recently, a seminal piece of work has proposed an approach to jointly profile an application's sensitivity to cache size and resulting increase/decrease in the requirement for main memory bandwidth [37]. In many ways, the information collected through the sensitivity study represent an experimentally driven profile. Yet, the workload characterization is quite coarse grained and cannot be directly used, for instance, to determine which specific pages of an application need to be shielded from interference.

BBProf shares many similarities, at least in terms of the end goal, with a number of well-established performance analysis toolkits. The survey in [3] provides a good overview of popular toolkits such as Oprofile [7], Dprof [29], Zoom [31], DynamoRIO [5], Valgrind [27], and Pin [23]. The latter three employ dynamic binary instrumentation (DBI), i.e. the ability to translate and instrument on the fly a target binary. DBI-based tools require extensive platform-specific porting. Translation layers for multiple platforms are already provided in Valgrind and DynamoRIO. DBI heavily impacts the timing of an application, so profiling of memory pages has to be performed by instrumenting all the memory references and then conducting a frequency analysis. To the best of our knowledge, the only work that uses one of these tools – the Lackey sub-tool in Valgrind – in this manner is [24]. In [24], a list of hot memory pages to be locked in cache is constructed via meomory tracing, but due to extreme performance degradation incurred, the evaluation is limited to small benchmarks. Lastly, DBI frameworks meant for general-purpose systems seldomly work out of the box on embedded systems due to the complex tree of library dependencies that they rely on, as also reported in [21]. Oprofile, Dprof, and Zoom rely on hardware performance counters to collect information. Oprofile records a variety of statistics such as the mix of hit/miss for L1/L2 caches. It relies on runtime sampling and provides a configurable trade-off between accuracy and overhead. Zoom and Dprof operate on similar principles but the development of Zoom has been discontinued in 2015, while Dprof relies on AMD-specific debug registers. Similarly, the profiling approach proposed in the recently published CacheFlow toolkit [34] relies on the hardware-specific ability, available in a subset of `Aarch64` CPUs, to snapshot the full content of CPU caches.

Since BBProf follows a measurement-based approach, it shares some similarities with the vast literature on measurement-based WCET estimation tools. For instance, the work in [30] aims at producing more accurate WCET estimates by designing synthetic benchmarks that stress different hardware resources in the target system. The purpose of BBProf is not to construct WCET estimates, but rather to extract the importance of each page for the timing of an application. This information can then be used to perform more fine-grained cache management. WCET analysis should be performed after a given management strategy has been applied, and it thus represents an orthogonal goal.

In light of the discussion above, what sets the proposed BBProf methodology apart is its unique capability of extracting fine-grained statistics on the contribution of each memory page to the overall runtime of an application under analysis. It does so without leveraging any hardware-specific support, by requiring no source- or compiler-level manipulation, and by operating directly on the black-box binary of the target application. Moreover, we demonstrate that the profile acquired through our BBProf can be used to enact advanced cache management techniques beyond strict task-level or core-level cache partitioning.

## 3 Background

In this section, we summarize the inner workings of the system components utilized by our tool for unfamiliar readers. We first present a brief overview of multi-level set-associative caches. Next, we review the notion of cache coloring, before concluding with a conspectus on memory representation and management in modern computing architectures.

**Multi-Level Set-Associative Caches.** Modern computing architectures implement several levels of caching. The L1 cache resides closest to the CPU and is private to a specific core. A cache miss in L1 triggers a lookup in the level below (L2, in this instance). Some architectures restrict the L2 cache to specific cores, making them private similar to the L1. A miss in the L2 cache may trigger a lookup in the level below (L3 and subsequently, L4) if it exists or failing that, a memory lookup. We constrain our discussion here to a normative ARM-based cache, with private L1 caches and a globally shared, last-level L2 cache.

At all levels, caches adhere to a set-associative modality where a set-associative cache with associativity $W$ consists of $W$ identically-structured *ways*. Blocks of consecutive bytes are stored in *lines* referred to as *cache blocks*. The constant $L_S$ denotes the number of bytes in a cache line, with most line sizes being 32 or 64 bytes. Memory addresses in the cache are divided into three groups of bits: the *offset, index*, and *tag* bits that affect the specifics of a cache lookup. Shared cache levels are physically indexed and physically tagged (PIPT), meaning all addresses used for cache lookups must be physical addresses.

**Memory Abstractions in Operating Systems.** Most modern operating systems employ a combination of hardware and software features to effectively encapsulate physical addresses into virtual addresses. Virtual addressing allows each process an exclusive view of the system's memory, alleviating problems such as memory fragmentation or the limited availability of physical memory. The OS maps virtual and physical addresses using *page tables*. When a process references a virtual address, the Memory Management Unit (MMU) performs a *page table walk* to locate the entry (PTE) – if any – that points to the corresponding physical memory page. If the walk is successful, the accessed virtual address is resolved into a physical address and the result of the translation is stored in the Translation Lookaside Buffer (TLB). If the address is not found, a page fault is triggered by the MMU and handled by the OS. If the access is legitimate, a new physical memory page is allocated and mapped to the process (*demand paging*); if it falls outside any valid range of virtual addresses, a segmentation fault (SIGSEGV) signal is delivered to the offending application.

Linux defines and manages the layout of legitimate contiguous regions of virtual memory by representing them as *virtual memory areas* or VMAs. VMAs consist of a range of start and end addresses, allowing for fine-grained control of virtual memory regions on a per-VMA basis. They have been a part of the Linux kernel since version 2.6 [8].

**Cache Coloring.** A major source of interference in multicore systems is LLC contention. One of the solutions to this problem is cache coloring, a purely software-based partitioning technique. With cache coloring, memory pages are assigned "colors" based on the cache sets they map to, which is determined by the value of the index bits. It is possible to allocate virtually-contiguous memory pages to physically discontiguous pages that have the same color. By doing this on a per-application or per-core basis, one can achieve strict cache partitioning, which is a well-known mitigation strategy for cache interference [12]. In multicore embedded SoCs that support two-stage address translations, the OS entirely manages the translation of the first layer address (user virtual address) into the intermediate physical address (IPA). The second stage of translation, however, is controlled by the hypervisor [28, 9] which maps IPAs to physical addresses. Hypervisor-level coloring is advantageous to transparently color entire guest OS's, as demonstrated in [26, 20, 13].

## 4 Design

In this section, we describe the main principles that comprise the design of the proposed BBProf. We describe the operational approach and functional components that allow it to carry out a fine-grained experiment-driven memory analysis of generic applications. While we advocate for the benefits of the proposed BBProf as a methodology for memory analysis, we have also carried out a proof-of-concept open-source implementation [11]. As we show in Section 7, the information extracted by our BBProf toolkit opens new avenues to perform fine-tuned management of shared memory resources.

In a nutshell, the main goal of the proposed BBProf toolkit can be formulated as follows. To consider a target application's memory footprint decomposed into its smallest manageable entities – individual memory pages. And with that, to produce a ranking that captures and quantifies how crucial is each page for the temporal behavior of the application. In other words, BBProf allows extracting the relative importance of memory pages towards the overall temporal behavior of a target application. Importantly, our BBProf should be able to handle applications of realistic complexity, while requiring minimum knowledge and understanding of the application itself – i.e., by largely treating the application as a black-box.

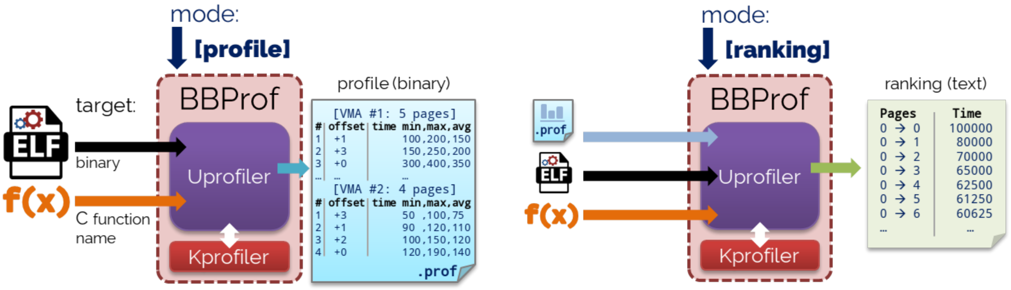
### 4.1 Core Principles

The core principles that have driven the design of the BBProf methodology can be summarized as follows.

**Model-free Operation.** Modern high-performance embedded systems are soaring in complexity. Additionally, manufacturers are often wary of providing exhaustive platform implementation details, as many of them constitute corporate intellectual property. Even if a formal micro-architectural model can be constructed, the high degree of complexity – in both software and hardware layers – can result in a state-space explosion even with simple workloads. It follows that, unfortunately, traditional static analysis methods might not be easily applicable to the considered class of embedded systems. In light of this, we aim to design a methodology that can be used in an arbitrarily complex system without the need for a micro-architectural model.

**Platform Independence.** A key design-time constraint we impose is for our BBProf methodology to be feasible regardless of the specific target platform. In other words, our BBProf should not rely on hardware support that exists only in a fraction of existing and future platforms. Instead, it should leverage widely available hardware features that are exposed by embedded and general-purpose platforms alike, and that are unlikely to be phased out in future generations.

**Usable for Realistic, Unknown Workload.** There exists a fundamental lack of practically viable toolkits that are industry-ready and capable of carrying out the memory analysis of complex applications in complex embedded platforms. The proposed BBProf aims that bridging such a gap with a solution that can be immediately adopted to better characterize



```
   [VMA #1: 5 pages]
#| offset| time min,max,avg
1|  +1   |      100,200,150
2|  +3   |      150,250,200
3|  +0   |      300,400,350
…|  …    |      …
   [VMA #2: 4 pages]
#| offset| time min,max,avg
1|  +3   |      50 ,100,75
2|  +1   |      90 ,120,110
3|  +2   |      100,150,120
4|  +0   |      120,190,140
                    .prof
```

```
Pages     |  Time
0 → 0     | 100000
0 → 1     |  80000
0 → 2     |  70000
0 → 3     |  65000
0 → 4     |  62500
0 → 5     |  61250
0 → 6     |  60625
…         |   …
```

**(a)** Profiling mode workflow.

**(b)** Ranking mode workflow.

**Figure 1** High-level workflow of BBProf in two of the main modes of operation.

the behavior of realistic applications. This implies that not only a minimal understanding of the target application should be required to perform profiling; but also that BBProf should be capable of handling widely used system-level features such as dynamically linked libraries and dynamic virtual memory allocation.

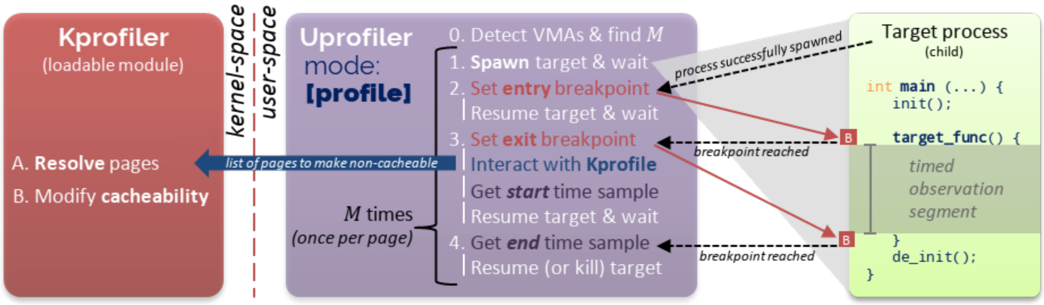
**Linear-time Profiling.** To be practically useful, we impose our BBProf methodology to be able to operate in linear time with respect to the memory footprint of the application under analysis. Because our strategy is centered around a runtime measurement-based approach, we deem as viable an analysis strategy with a linear time complexity that is impacted by (1) the runtime of the core logic of the application under analysis; and (2) the size of the memory footprint of the target application.

## 4.2 High-level BBProf Workflow

The proposed BBProf methodology pivots around the idea that it is possible to manipulate the memory allocation policy on a per-memory page basis. Thus, for a target application, it is possible to understand the importance of individual pages towards application timing by changing the allocation policy one page at a time. Albeit this idea is generic, the specific set of memory allocation policies depends on the type of analysis to be conducted. For the remainder of this paper we direct our focus to shared CPU cache analysis, which is a primary target of this work. Therefore, cacheability is the memory policy of choice to isolate the impact of a single memory page on the timing of an application.

Figure 1 provides a high-level overview of the logical workflow of BBProf in its two main modes of operation. In the *profile* mode described in greater detail in Section 4.3 and depicted in Figure 1a, the required inputs to BBProf are (1) the path to the binary of the ELF executable to be profiled; and (2) the name of the C function whose timing needs to be profiled. This function corresponds to the *observation segment* defined below. The full list of optional operational parameters are described in [11]. The output produced in this mode is a binary file[1] encoding the relative importance recorded for each page of each considered VMA. BBProf allows performing multiple profiling runs and will aggregate the result of all the runs into the same file keeping track of max, min, and average statistics on a per-page basis.

[1] The binary profile can be translated into human-readable format using the `-t` parameter as described in [11].



```
int main (...) {
  init();

  target_func() {

  }
  de_init();
}
```

Figure 2 Logical interplay between modules of BBProf in *profile* mode.

BBProf includes a number of other analysis modes described in Section 4.4. These modes require a profile file previously obtained on the target application. For instance, Figure 1b depicts the high-level workflow of the *ranking* mode which produces a human-readable output describing the runtime of the target function as an increasing number of most important pages are made cacheable.

We base our analysis on the presence of a single aforementioned observation segment, which represents a segment of logic whose temporal behavior is of interest. Although the observation segment can be extended to cover the entire application's logic, in practice this is often not the case. Realistic applications are typically characterized into three main phases: (1) an initialization phase where parameters and inputs are parsed and pre-processed; (2) the main computational payload, which might be executed multiple times in a periodic fashion; and (3) a teardown phase where any acquired resource is released. The observation segment corresponds to the main computational payload of the target application. For the sake of simplicity, we assume that such a phase is encapsulated into a single function called the `target_func`, and hence that the target application has a structure similar to what depicted in the right-hand side of Figure 2. Any initialization and de-initialization logic is excluded from the analysis.

## 4.3 Profiling Strategy

When operating in profiling mode, the adopted strategy is visualized in Figure 2 and described in the following. (1) Perform a first run of the target application to identify its virtual memory layout; (2) re-execute the target application as many times as the number of memory pages $M$ that comprise its memory footprint; (3) at each re-execution and before the invocation of the `target_func`, switch memory allocation policy for all the pages except the one under analysis; and (4) collect the impact of the selected policy over the execution time of the `target_func`. It is crucial that the profiling of an application is conducted in isolation, i.e., with the lowest possible amount of noise in the target system.

For instance, consider an application whose memory footprint is comprised of 4 pages and assume that its runtime when all the pages are marked as non-cacheable is some time $T_{nc}$. BBProf first detects the footprint of the application. Next, it performs 4 iterations. In the first iteration, only the first page is marked as cacheable, while all the others are marked as non-cacheable. Then, it measures the runtime of the `target_func` which will be of the form $(T_{nc} - x_1)$, with $x_1$ being the performance gain that arises from having the first page in cache. We then repeat the same steps for the remaining three pages to extract the terms $x_2, x_3$, and $x_4$ in the same way.

To accomplish the strategy outlined above, our methodology relies on the definition of two components, as also depicted in Figure 1: a *user-space driver* and a *kernel-space driver*, which we refer to as UProfiler and KProfiler, respectively. Intuitively, the UProfiler is responsible for launching and collecting data about the temporal behavior of the target application, while the KProfiler is used to enforce the selected memory allocation policy. The main key design principles for the two components are reviewed in the following.

### 4.3.1 User-Space Driver (UProfiler)

The design of the UProfiler component shares a number of similarities with a typical debugger. Indeed, it operates by taking in two pieces of information – which are the only ones strictly required to launch profiling. These are (1) the location of the executable binary (and any parameters it requires) of the target application; and (2) the name of the target function that corresponds to the observation segment.

First, UProfiler parses the provided binary executable to translate the name of the function into the address that corresponds to the first instruction of the target function – i.e., the beginning of the observation segment. With this information at hand, UProfiler can launch the target application and set a breakpoint, called the *entry breakpoint* right at the beginning of its computational payload (Figure 2, step 1). As soon as the entry breakpoint is reached, UProfiler pauses the target application and performs a sequence of preparatory actions, called the *entry sequence*. The actions performed in the entry sequence depend on the type of analysis being carried out.

As part of the entry sequence, UProfiler always detects the end of the observation segment. This is done by inspecting the return address of the target function. With this information, an *exit breakpoint* is installed by UProfiler (Figure 2, step 2). Before resuming the execution of the target application, UProfiler removes the entry breakpoint and snapshots the current `start` timestamp (Figure 2, step 3). In a similar way, as soon as the exit breakpoint is reached, UProfiler immediately snapshots the current `end` timestamp (Figure 2, step 4); removes the exit breakpoint, and performs a variable sequence of actions – the *exit sequence*.

During the very first run of the target application (iteration 0), UProfiler detects its layout and the number of memory pages $M$ that comprise its footprint. This information is collected during the entry sequence and double-checked during the exit sequence. Additional implementation-specific details about this step are provided in Section 5.

In the generic profiling iteration $i$, the entry sequence is used by UProfiler to prepare a descriptor that determines the memory policy to be applied to each of the pages subject to profiling. Given the current focus on cache analysis, the descriptor prepared at profiling iteration $i$ instructs the KProfiler to turn all the considered pages non-cacheable except for the $i$-th page. In the exit sequence, the difference between `start` and `end` timestamp is recorded and associated to page $i$.

Here, the use of timestamps represents the preferred metric for two main reasons. First, it allows UProfiler to be a valid methodology regardless of the target platform, since time sampling primitives are commonplace in (modern) hardware platforms. Second, it allows UProfiler to directly correlate the impact of the selected memory policy on the timing of the observation segment. Nonetheless, UProfiler can be easily extended to capture additional platform-specific performance metrics such as number of cache references, hits, misses, number of retired instructions, instructions-per-cycles, and so on.

### 4.3.2 Kernel-side Driver (KProfiler)

The KProfiler encapsulates all the logic that requires elevated kernel-level privileges to manipulate the properties of the memory pages mapped to the target application.

Following the proposed design, the KProfiler defines a communication interface exposed to the UProfiler (Figure 2, step 3). As needed – usually during the entry sequence – the interface is used to pass a descriptor with the list of changes to be applied to the target memory pages. Because absolute memory addresses change from run to run, UProfiler and KProfiler use relative addressing to uniquely identify memory pages across runs. Pages are grouped by the memory policy modification to be carried out over them.

It is responsibility of the KProfiler module to leverage appropriate kernel-level APIs to apply the requested memory policy modifications for the target pages. So far we have only discussed the most basic operation mode of the proposed BBProf. In this case, the descriptor passed by the UProfiler always follows the same structure. Only one page is selected to be kept cacheable, while all the others are requested to be made uncacheable.

## 4.4 Additional Operational Modes

So far we have described the design of UProfiler and KProfiler with respect to the main operational mode, which is page-level cache profiling. Our current design includes two additional modes that are briefly described in the following.

**Page Ranking Analysis.** Once per-page statistics have been extracted, it is possible to globally rank all the memory pages that comprise an application's footprint. Intuitively, those pages that led to the best time improvements will be ranked as *more important* towards the temporal behavior of our target. The page ranking analysis allows to understand the *cumulative* benefit of selecting the top-ranked $k$ pages to be cacheable, where $0 \leq k \leq M$. Notably, the case $k = M$ corresponds to the default case where all the memory pages are considered cacheable. Expectedly, as we increase $k$, the observed runtime of the observed segment will generally decrease. Importantly, however, if a threshold of $k^* < M$ is found where the resulting runtime already approaches the case $k = M$, then $k^*$ corresponds to the working-set size (WSS) of the target application.

**Page Migration Analysis.** A final useful operation provided in our design is the possibility of changing the physical location of a group of pages based on the information collected during profiling and ranking. For instance, consider a platform that includes a block of scratchpad memory. First profiling and ranking is performed to identify the pages that comprise the working-set of the target application. Next, our BBProf toolkit can be used to test what-if scenarios where all or a part of this group of pages is migrated to scratchpad memory. We will demonstrate two concrete use-cases where page migration can be used to efficiently mitigate inter-core cache interference.

## 5 Implementation

We hereby review the main details concerning a proof-of-concept Linux implementation of the proposed BBProf toolkit.

## 5.1 UProfiler Implementation

As we mentioned in Section 4, the UProfiler component is designed to act akin to a debugger. For this purpose, it leverages the `ptrace` family of system calls to manipulate the flow of a child process. Indeed, launch a new run of the target application, UProfiler performs the following sequence: (1) a fork system call to spawn a new child process, (2) a ptrace(PTRACE_TRACEME) in the spawned child allowing the parent to trace the child's execution, (3) an exec system call to execute the target application under tracing.

The ptrace system call represents a standard Linux interface. Albeit it is Linux-specific, it is possible to achieve a similar behavior even in a bare-metal system or RTOS by relying on basic debugging features. Indeed, the only features used by UProfiler are (1) the ability to set/remove breakpoints, and (2) the ability to read the content of CPU registers. These capabilities are available even in simple microcontrollers.

**Breakpoint Handling.** To set a breakpoint in an architecture-independent way via the ptrace interface, one can replace (PTRACE_POKETEXT) the instruction at the desired breakpoint address with any illegal opcode. This way, when the execution of the tracee reaches the modified instruction, the process is paused by a SIGILL POSIX signal and a SIGCHLD signal is delivered to the parent process – i.e., to our UProfiler. Before setting the breakpoint, UProfiler records the value of the instruction being replaced (PTRACE_PEEKTEXT) so that it can be restored once the breakpoint is reached. As soon as the breakpoint is hit, UProfiler records the value of the tracee's program-counter (PC) register. To allow the tracee to resume from the breakpoint, UProfiler (1) restores the original instruction at the breakpoint address and (2) rewinds the PC of the tracee to the recorded address. Accessing the tracee's CPU registers can be done via a combination of PTRACE_GETREGS/PTRACE_SETREGS operations[2].

As discussed in Section 4, UProfiler only sets two breakpoints. The entry breakpoint is set upon launching the target application and at the first instruction of the target function. The exit breakpoint is installed at the address to which the target function is set to return. To find the address of the entry breakpoint, UProfiler accepts as a command-line parameter the name of the target function whose body corresponds to the observation segment. It then uses the LibELF[3] library to translate the provided function name into the corresponding instruction address by performing a lookup in the target ELF's symbols table (SHT_SYMTAB). The address of the exit breakpoint is only known once the tracee hits the entry breakpoints. In ARM32 and ARM64, it is enough to read the content of the link register (LR) to retrieve the return address of the target function.

**Layout Detection and Enforcement.** In a generic POSIX-compliant application, there is a number of system calls that can dynamically modify the memory layout of an application. Most notably, sbrk is internally used by the libc to implement functions that perform dynamic memory (de)allocation, such as malloc and free. Calling the sbrk can affect the size of the heap virtual memory area (VMA). Similarly, the mmap and unmap system calls can cause the addition, deletion, or modification of VMAs in the tracee's layout. Importantly, the libc uses mmap instead of performing a heap extension when applications allocate large buffers. For the final output of our BBProf to be valid, it is crucial that no memory layout changes occur during the execution of the observation segment. This is not a concern with applications written for embedded/safety-critical systems where memory is always statically allocated. Nonetheless, UProfiler includes logic to enforce a deterministic memory layout even on applications that use dynamic memory allocation primitives.

[2] Note: this is true for many platforms, including x86, x86_64 and ARM32. Equivalent operations can be carried out in ARM64 through PTRACE_GETREGSET and PTRACE_SETREGSET.

[3] LibELF is part of the elfutils open-source project which is a toolkit to read, create and modify Executable and Linkable Format (ELF) binaries.

To achieve that, when the tracee is spawned for the first time, UProfiler runs the tracee a first time and records the peak amount (VmPeak) of data that was used during the target function. Once the maximum amount of memory required by the observation segment is known, all the subsequent runs of the target application are performed by setting two environmental variables that modify the behavior of the libc memory allocation routines. These are (1) the MALLOC_TOP_PAD_ and (2) the MALLOC_MMAP_MAX_ variables. The former allows setting an initial size for the heap and is set to the peak memory size detected by UProfiler in the first run. The latter is set to 0 to disable the use of mmap to handle dynamic memory allocations.

All the subsequent runs of the target application can be used to perform profiling. In the first of such runs, UProfiler further detects the actual memory layout that results from setting the aforementioned environmental variables. It does so by querying the /proc/PID/maps interface as soon as the entry breakpoint is reached. Additional launch parameters are accepted by UProfiler to include/exclude certain types of VMAs in the profiling. For instance, in order to make profiling faster, one might want to exclude VMAs that belong to shared libraries and that are not used during the observation segment.

**Single-page Profiling.** Once UProfiler has computed the number of pages $M$ in the target VMAs, the single-page profiling phase can be initiated. Of course, the $M$ pages can be distributed across multiple VMAs (e.g. text, heap, stack). Moreover, their absolute address will change from run to run due to address space layout randomization (ASLR). To operate even with ASLR in place, UProfiler uses a run-independent relative encoding to express the coordinate of memory pages. Specifically, we use two indices to identify each page: (1) the index $v$ of the VMA that contains the page; and (2) the offset $o$ of the page from the beginning of the VMA.

To profile a generic page $i \in \{1, \ldots, M\}$ with coordinates $\langle v, o \rangle$, the UProfiler prepares a descriptor to instruct the KProfiler module to modify the cacheability of the pages in the target VMAs. In profiling mode, this descriptor contains the list of all the VMAs under analysis. For each of them, a list of pages whose cacheability attributes need to be modified is included, with an opcode field that determines how the cacheability attributes should be altered. In this case, the cacheability of page $i$ is unchanged, but that of all the other pages is the target VMAs is set to become non-cacheable. The descriptor prepared as mentioned above is then passed to KProfiler to apply the necessary changes once the entry breakpoint is reached. The target application is resumed only once all the pending changes are effective. Note that any timestamp acquisition is performed after the cacheability changes have been applied, so that the overhead required to switch the cacheability attributes is excluded from the time measurements.

**Time Measurements.** Albeit extensible, the current use of the BBProf toolkit is to analyze the relative importance of individual memory pages toward the overall temporal behavior of the observation segment. The most direct and platform-independent way to extract this information is by acquiring timing samples of the target function as we vary which page is allowed to be allocated in cache. In order to be as precise as possible, UProfiler directly reads CPU cycle counters instead of relying on system primitives.

Time measurements are acquired right before resuming the application from the entry breakpoint and right after it reaches the exit breakpoint. Moreover, since timestamps can be affected by random system noise, UProfiler allows specifying an arbitrary number of samples to be collected for the same profiled page. System noise originates from workload on other cores, interrupt handlers, non-deterministic hardware behavior, and inaccuracy of time sampling instructions. Various mitigations strategies can be adopted to reduce the magnitude of system noise, such as turning off other cores and disabling peripherals. The only mitigation strategy used by BBProf is running UProfiler and the target process with the `SCHED_FIFO` Linux policy and with a high real-time priority. As we evaluate in Section 7.2, the observed degree of noise was negligible and did not impact the validity of our profiles. The final profile stores, for each page, the maximum, minimum, and average runtime of the observed segment across all the acquired samples. Note that with this infrastructure in place, it is straightforward to extend UProfiler to collect additional metrics such as hardware counters for micro-architectural events – e.g. cache references, misses, hits, bus accesses, to name a few. This can be done in a platform-agnostic fashion by leveraging the `perf` infrastructure [10].

**Page Ranking and Migration.** The implementation of the other two modes of operation is similar to what has been discussed above, hence much of the details are omitted. To perform page ranking and migration, it is assumed that a profile has been previously acquired for the target application. The pages in the profile are then arranged in a sorted set in descending order of their impact on the timing of the target application. Examples of the output produced by a ranking experiment are provided in Figure 8.

In the ranking phase, UProfiler performs $M$ runs where in run $k$, the top $k$ pages in the sorted set are requested to be kept cacheable by the KProfiler, while all the remaining pages in the set are turned non-cacheable. The timing of the $M$ runs is collected and stored for later analysis.

In a similar way, a page migration experiment requires a pre-acquired profile. The $M$ pages in the target VMAs are sorted according to the same criterion described above. In this case, however, a single run is performed where the UProfiler instructs the KProfiler module to migrate the top $k$ pages in the sorted set to a new location in physical memory. The value of $k$ represents a parameter supplied by the user. The destination of the migration is determined by the KProfiler, as we discuss below. The support to conduct page migration directly from the profiler allows quick testing of what-if scenarios for the allocation of important pages. As part of our future work, we plan to directly modify the way applications are launched to take advantage of profiling information without the need to go through the profiler.

## 5.2 KProfiler Implementation

The KProfiler component is implemented as a Linux kernel module. Our current implementation targets Linux 5.4. At startup, a communication channel with the UProfiler is created in the form of a file in the `proc` pseudo-file system. Whenever the UProfiler needs to trigger a kernel-side operation, the `write` system call is used to pass the content of the aforementioned operation descriptor. The descriptor also contains the PID of the tracee that will be targeted for the current operation. A combination of `find_get_pid` and `get_pid_task` kernel APIs is used to retrieve the descriptor of the tracee's process given the provided PID. Moreover, the descriptor contains redundant information about the structure of the memory layout of the tracee as detected by UProfiler. This is used to perform a sanity-check in the KProfiler and ensure that the desired operations are performed on the right VMAs and pages.

**Cacheability Modification.** For the profiling and ranking phases in which only the cacheability of the target page(s) is changed, no changes to the source code of the Linux kernel are required.

For each VMA in the passed descriptor, the KProfiler retrieves the corresponding vm_area_struct descriptor by scanning the kernel-maintained linked list of tracee's VMAs. It then ensures that any page that will be affected by the current operation is present in physical memory. This is done by *faulting-in* the target pages that can be achieved via the kernel API revget_user_pages_remote and with flags FOLL_POPULATE, FOLL_TOUCH and FOLL_MLOCK. Next, the kernel API apply_to_page_range is used to invoke a custom function for each page on which a change in cacheability attributes needs to be carried out. Such a function already invokes our custom routine with a pointer to the Page Table Entry (PTE) that needs to be manipulated to change the cacheability attributes of the page.

Given a page that is set to be made non-cacheable, the following steps are performed. First, a new PTE is prepared to mirror the same exact value of the existing PTE, but where the page attributes have been switched to encode for normal, non-cacheable memory. Next, we clean and invalidate data and instruction caches to make sure that any dirty line is written back to main memory. Then, we install the newly created PTE to replace the previous entry. Finally, we invalidate any TLB entry (if any) for the current page on all the online CPUs.

**Page Migration.** Being able to support page migration requires some changes to the kernel sources[4]. A total of around 200 lines have been modified to implement the required changes. Specifically, we have generalized the existing support for the migration of physical memory pages across NUMA nodes used to implement the move_pages system call. We have introduced a new exported kernel API with the following prototype:

```
int move_pages_to_pvtpool(struct mm_struct *mm, unsigned long nr_pages,
                          unsigned long * vaddrs, new_page_t get_new_page,
                          unsigned long private);
```

Here mm is the virtual address space descriptor of the process targeted for page migration, nr_pages is the number of pages to be migrated, vaddrs is an array of nr_pages virtual addresses of pages to be migrated, get_new_page is a function pointer used by the internal routines to allocate destination pages, and private is a parameter to be passed to the allocation function.

At load time, the KProfiler module internally maps an area of memory reserved at boot for page migration. The reservation is performed via a modified Device Tree Blob (DTB). Here we use the reserved-memory attribute [5] to exclude a given range of physical addresses from the default Linux allocator – the Buddy System. We do not mark this region with the no-map attribute to allow the kernel to initialize the necessary page descriptors to correctly map kernel virtual addresses and physical addresses in the reserved region.

If a valid reservation is found by the KProfiler at load time, the module uses a combination of memremap and gen_pool_create kernel APIs to instantiate a new general-purpose memory allocator over the reserved memory region [6]. The former produces a valid kernel virtual address that can be used to access the reserved memory region, while the latter enables the allocation of new pages from the region.

With our custom allocator in place, whenever UProfiler requests the migration of a set of pages, a set of initial steps similar to those required to change the cacheability attributes is performed. But instead of manipulating the cacheability attribute of the exiting pages, a

[4] The modified kernel sources are available at https://github.com/rntmancuso/linux-xlnx-prof.

[5] See https://www.kernel.org/doc/Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt.

[6] See https://www.kernel.org/doc/html/v5.4/core-api/genalloc.html.

list of pages to be migrated is compiled and the newly introduced `move_pages_to_pvtpool` API is invoked. When doing so, a wrapper to a `gen_pool_alloc` call is passed as the `get_new_page` function pointer to allow internal book-keeping.

We describe in Section 7.4 how profile-driven page migration can be used to enact advanced techniques to manage inter-core interference in the shared cache. Nonetheless, the implications of profile-driven page migration are deeper than what presented in Section 7.4. Indeed, this support allows defining a distinct memory pool for each heterogeneous memory component available in the system, e.g. scratchpad memory, in-FPGA block RAM, non-volatile memory, reduced-latency DRAM blocks (RL-DRAM) [14], to name a few. By leveraging profiling information, one can then decide which pages need to be mapped to the various memory resources.

## 6 System Instantiation

In this section, we review the full-system setup that was carried out to evaluate the potential of the proposed BBProf approach and proof-of-concept implementation.We have deployed the implemented UProfiler and KProfiler modules on an `ARM64` platform that we also use for all our experiments. Specifically, we use a Xilinx-ZCU102 development platform featuring a Zynq UltraScale+ XCZU9EG MPSoC [36] with a quad-core ARM Cortex-A53 [2] 64-bit CPU operating at 1.5 GHz and implementing the ARMv8-A [15] architecture profile. The L1 cache consists of a split cache with a 32 KB 2-way instruction (I) cache plus a 32 KB 4-way data cache. The L2, which is also the last-level cache (LLC) is unified and 1 MB in size; it has associativity 16, and it is shared among all the A53 cores. The cache line size is 64 bytes for both L1 and L2.

Profiling and ranking analysis can be carried out directly under Linux. Conversely, to evaluate the ability to enact advanced memory management via profile-driven page migration, we additionally deploy a thin partitioning hypervisor, namely Jailhouse [1]. Jailhouse is used to perform cache coloring [38, 18, 26, 20] in a way that remains transparent to the Linux environment where we conduct our experiments. Our goal is to conduct a series of experiments centered around the problem of shared cache management. To achieve this, we have reproduced the setup described in [20] on the ZCU102 system, where dynamic re-coloring of the Linux environment is available. We use coloring in two ways. First, in a traditional way to statically restrict the applications running in the Linux environment to only a subset of the available colors – we vary this amount from two to 15, with 16 being the maximum value and corresponding to no partitioning. In this case, Linux is restricted to use only one CPU. Moreover, when strict coloring is used, interfering workload (Interf) consists of bare-metal memory-intensive synthetic applications deployed on all the other cores as stand-alone virtual machines (VM).

We then use Jailhouse and page coloring to illustrate a new technique enabled by the profiler to mitigate the problem of shared cache interference. The setup, illustrated in Figure 3, essentially defines two contiguous ranges of intermediate physical addresses (IPA). The first corresponds to all the memory that Linux uses for legacy memory allocations through the Buddy System and is mapped by Jailhouse to $12/16 = 3/4$ of the available colors. The second IPA range is mapped to pages with the remaining $4/16 = 1/4$ of the available colors. The latter is then used by the KProfiler to instantiate a privately managed allocation pool. It follows that pages can be allocated in the pool only through explicit profiler-driven page migration. We refer to this setup with the PVT+SH short-hand notation. Note also that this setup provides page-level granularity over memory allocated in the private cache pool. This sets this work apart from the large literature on colored page allocators proposed in the past that assign colors at the process or core granularity [17, 19, 18, 22].

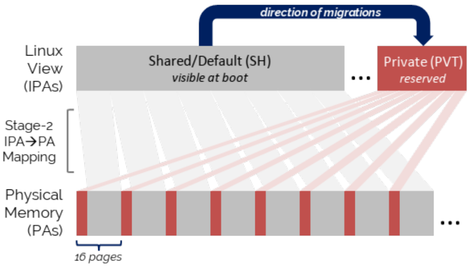


**Figure 3** Overview of PVT+SH setup.

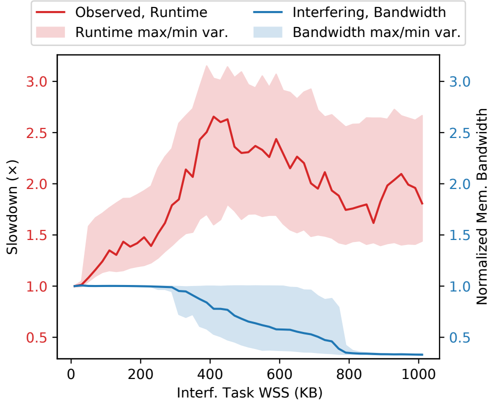


**Figure 4** Interference as a function of WSS.

In terms of workload, apart from the aforementioned INTERF workload, an equivalent synthetic memory-intensive application, namely `bandwidth` from the IsolBench suite[7], is used to generate cache contention when no other VMs are active in the system and Linux is used in SMP mode on all the cores. For the purposes of building confidence in the ability of the profiler to characterize the importance of memory pages, we use the STAIRCASE synthetic benchmark described more in detail in Section 7.2. For our observed realistic workload, we used the San Diego Vision Benchmark (SD-VBS) suite [35]. While we conducted all our experiments on all the benchmarks, due to space constraints we only include a subset of the results that capture the more interesting cases. We also limit our discussion to the input sizes SQCIF, QCIF, CIF, and VGA. We exclude the FULLHD sizes as the runtime of the benchmarks on the target platform is excessively high. As we mentioned in Section 5, the observed system noise was quite negligible which resulted in the timing of the profiled applications to be remarkably deterministic. Thus, five independent runs were sufficient to acquire each profile. For production systems with worse signal-to-noise ratios, we expect that a much larger number of runs might be needed to construct meaningful profiles.

## 7 Evaluation

In this section, we describe the evaluation that we have carried out on the system setup described in the previous section. We focus our attention on four main aspects. First, in Section 7.1 we evaluate the amount of shared cache contention that can be suffered by applications in this platform and understand the ability of strict cache coloring to mitigate such interference. Next, we show in Section 7.2 that our proof-of-concept BBProf implementation is capable of extracting useful profiling information for the considered synthetic and real-world applications. Third, we discuss how profile-driven migration can be used efficiently to solve the problem of contention-induced instruction stall in Section 7.3. Finally, we evaluate in Section 7.4 how profile-driven page migration can be used to controllably mitigate shared cache contention in real-world applications.

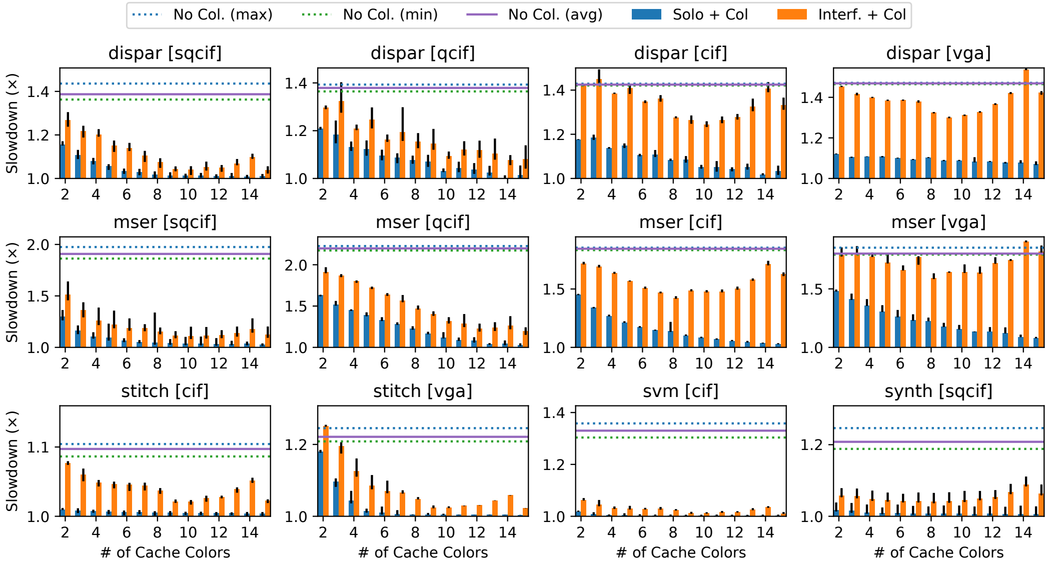


**Figure 5** Performance of SD-VBS benchmarks under strict partitioning with (orange) and without (blue) cache contention.

## 7.1 Interference and Mitigation via Strict Partitioning

In the experiments presented in this section, we focus on cache contention. Generating cache contention for an application under analysis is done by deploying a set of interfering synthetic memory-intensive applications on all the other cores. In order to set the WSS of the interfering workload with the goal of maximizing contention, we have conducted the experiment depicted in Figure 4. In this experiment, the application under analysis is MSER from the SD-VBS suite with input size SQCIF. Three interfering applications deployed on the remaining cores continuously perform cache-allocate store operations over a buffer of increasing size ($x$-axis). We plot on the left $y$-axis (red) the runtime normalized to the case in which MSER runs in isolation (*solo* case) in the system. We display the memory bandwidth observed by the interfering workload on the right $y$-axis (blue). A clear trend emerges that highlights how the cache interference is maximized (both in average and maximum terms) when each interfering application accesses a buffer of around 420 KB, i.e. access in a total of about 1.23 MB.

In light of the results highlighted above, we have set our interfering tasks to have a WSS of 420 KB. With this in mind, we want to understand how well strict coloring is able to mitigate cache interference. We have conducted a study where all the strict coloring configurations described in Section 6 are explored for all of our SD-VBS benchmarks and considered input sizes. The most interesting nine cases are presented in Figure 5. In all the sub-plots, the vertical bars represent the slowdown of the application under analysis when no cache partitioning is performed. The blue bars (resp., orange) report the runtime of the application under analysis in the solo case (resp., under interference). It emerges that partitioning leads to significant improvements in certain circumstances, especially for workload with L2-sensitive footprint such as DISPARITY and MSER with input sizes QCIF

[7] See `https://github.com/CSL-KU/IsolBench/blob/master/bench/bandwidth.c`.

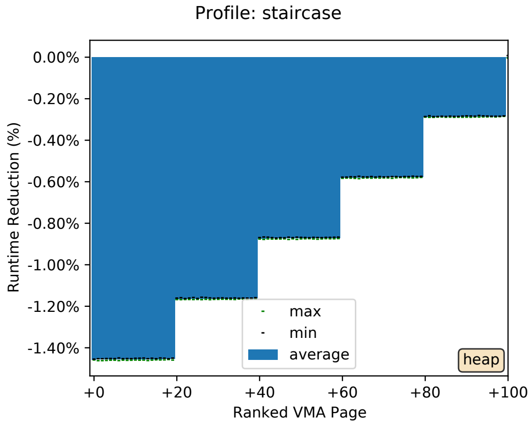


**Figure 6** Profile of STAIRCASE benchmark.

and SQCIF, and for STITCH with input size CIF. However, the ability to mitigate cache contention with coloring alone is limited in some cases. This is due to contention over memory bandwidth which exacerbates as larger partitions are given to large-footprint applications – see DISPARITY and MSER with input sizes CIF and VGA. Indeed, the stress over the main memory subsystem placed by the interfering workload increases as it is confined to a smaller cache partition. Traditionally, bandwidth throttling techniques are used to solve this problem, such as MemGuard [39, 33].

But an important takeaway from this study is that strict partitioning is just too rigid to (1) be able to efficiently mitigate cache contention for a wide variety of tasks deployed on the same core. And (2) that over-throttling of the interfering workload might be required to compensate for the lack of flexibility in coloring-based cache partitioning. Conversely, as shown in the following, the proposed BBProf toolkit can be used to strike a balance between strict partitioning and unregulated interference.

## 7.2 Profiling of Staircase and SD-VBS benchmarks

The first step toward profile-driven cache management is to use the proposed BBProf toolkit to acquire the page-level profile about the applications to be managed. As a first step to build confidence on the correctness of BBProf, we have designed the STAIRCASE benchmark[8] to exhibit a well-recognizable behavior in terms of memory accesses that can serve as the *ground truth* on the extracted profile. Specifically, the benchmark allocates a buffer of 100 heap memory pages. It then performs a total of 1000 iterations reading over the buffer. In the first 200 iterations, the buffer is read entirely; in the next 200 iterations, the first 20 pages are skipped; after 200 additional iterations, the first 40 pages are skipped and so on. The result is that the second group of 20 pages is accessed 2× more than those at the beginning of the buffer. The third 20-pages group 3× more, and so on. Thus if we were to plot the importance of each page from beginning to end, the resulting plot would resemble a *staircase*, hence the name. Figure 6 provides a visualization of the extracted profile focused on the heap VMA. In the figure, the $x$-axis represents the index of the page under profiling. The blue bars from the top of the plot visualize by how much (in percentage) the runtime of the benchmark is reduced when each page is kept cacheable while all the others are not. A taller bar signifies a page with relatively higher importance for the temporal behavior of the

[8] The code of the STAIRCASE benchmark is available in the project repository [11].

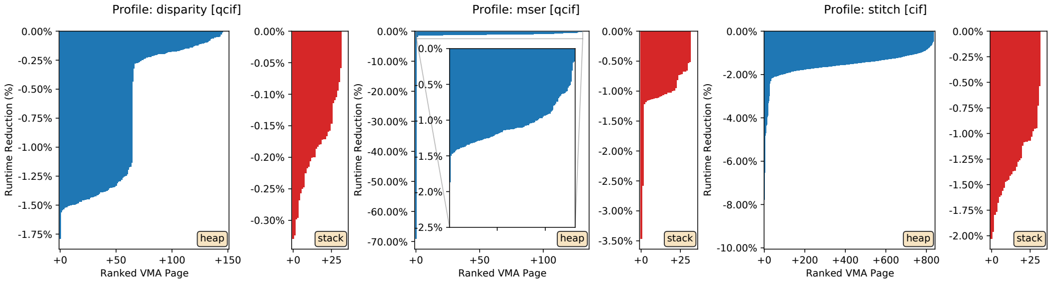


**Figure 7** Profile of Disparity (left), Mser (center), and Stitch (right) – heap, stack pages only.

application under analysis. For all the bars, the normalization baseline is always taken as the application's runtime when none of the pages in the target VMAs is made cacheable. The pages are sorted based on their importance rather than their offset in the VMA. Because of the by-importance sorting, the most-accessed pages appear to the left-hand side of the plot, with the recognizable staircase characterization having been reconstructed by BBProf. One can also note that the gap between min and max in each profile sample is quite small, thus leading to the conclusion that the overall measurement noise is negligible.

Next, we have acquired a profile for all the benchmarks in the SD-VBS suite, one for each of the considered input sizes. Due to space constraints we only visualize the three most representative profiles, namely those for Disparity, Mser with input size QCif, and for SVM with input size Cif. These are displayed in Figure 7, where we limit the plots only to the `heap` and `stack` VMAs. The style of the sub-plots in Figure 7 is identical to that of Figure 6, with the only difference that the bars of stack pages are color-coded in red and that we have omitted max/min error bars to avoid over-plotting. From the figure it emerges that in all the cases there exists a small group (1-3 pages) of heap pages that has a large impact on the runtime of the application. From left to right, these alone cause a reduction of around 1.8%, 69%, and 7.9% when kept cacheable. Moreover, the temporal behavior of Mser and Stitch is more heavily impacted by `stack` pages; the Disparity benchmark has a core set of around 65 `heap` pages that comprise its working-set. Taken individually, the presence in cache of each of these pages alone contributes to a runtime reduction between 1.25% and 1.5%.

To further understand the relationship between important pages and overall application runtime, we conduct a ranking analysis (see Section 4.4) given the profiles obtained at the previous step. In Figure 8 we depict the result of the ranking analysis conducted on Disparity, Mser, Localization, and Stitch. In each subplot, the $x$-axis reports the number of pages, sorted in order of importance, that are made cacheable. The $y$-axis reports the resulting normalized runtime of the application under analysis. The normalization baseline is the runtime when only the most important page is made cacheable. A stark contrast emerged in the behavior of the considered applications. Specifically, Disparity features a block of pages with comparable importance that produces a constant slope in the runtime reduction as more pages are made cacheable. It is also possible to appreciate how the WSS size increases as the input size goes from SqCif to Vga. Conversely, the WSS of Mser is concentrated in a very small set of pages for the SqCif and QCif case, and increases rapidly for input sizes Cif and Vga. Next, Localization is characterized by *quantized* temporal improvements unlocked only when a certain threshold of pages is allocated in cache. Finally, Stitch appears to be relatively insensitive to caching as long as a core set of about 10 pages is allocated.

Once the profile has been acquired, it is important to understand if the set of memory

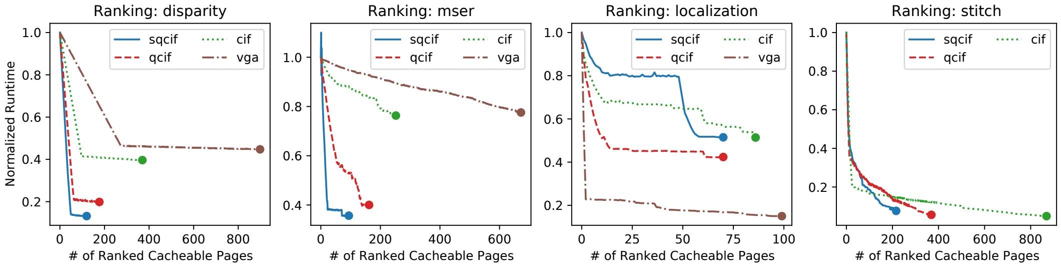


**Figure 8** From left to right, ranking analysis of Disparity, Mser, Localization, and Stitch.

pages deemed *important* remains the same as when the content of the input images changes while their size remains the same. In the general case, this might not be true while for some applications the profile might transcend the specific data input. We hereby conduct a sample evaluation to understand in which category the considered benchmarks fall. Note that this is not meant to represent an exhaustive evaluation. For this experiment, we consider the profiles acquired on the default ("def") input images provided with the SD-VBS suite. In terms of benchmarks, we limit ourselves to Disparity, Mser, Tracking, and Stitch. Compared to Figure 8, we have replaced Localization with Tracking because the latter uses images as input while the former takes as input a text file with an unknown format. The selected input size is VGA for Disparity, Mser, and Tracking and CIF for Stitch because the latter runs for too long over the VGA input size. For each benchmark, we have produced four additional input images. The first two called "nor1" and "nor2" are meaningful (*normal*) scenes, while the last two, namely "deg1" and "deg2" are scenes that correspond to corner (*degenerative*) cases. Specifically, "deg1" corresponds to random noise while "deg2" to a solid-color frame. Due to space contraints, we refer the reader to the project repository [11] for the full list of images used in this experiment.

Figure 9 provides the same type of analysis used to construct Figure 8. The key difference here is that for each of the considered benchmarks we construct the displayed ranking curves using the profile originally acquired with the "def" input images. To more clearly appreciate the difference in absolute runtimes as we vary the images supplied in input, the runtimes are not normalized and are instead expressed in CPU cycles. Among the four considered benchmarks, the runtime of Mser is the most heavily affected by the content of the input data. Nonetheless, the general trend in terms of runtime reduction as an increasing number of ranked pages is made cacheable is consistent across experiments. In the Disparity case, all the curves remain quite consistent. This suggests that the benchmark remains quite insensitive to the input image and that the profile acquired with the default input captures well the relative importance of individual memory pages regardless of the supplied input images. The Tracking case is quite similar to the Disparity case, with the trend of the curve remaining consistent across experiments. Conversely, Stitch shows visible variations in the relative importance of memory pages, especially when comparing between the "deg1" and "deg2" cases. In this case, the profile obtained with the "def" input images does not generalize well. We can conclude that what captured by BBProf remains mostly accurate for three out of the four benchmarks considered in this experiment. The fourth case (Stitch) displays important dependencies between input images and memory usage, in which case the profile constructed by BBProf does not generalize.

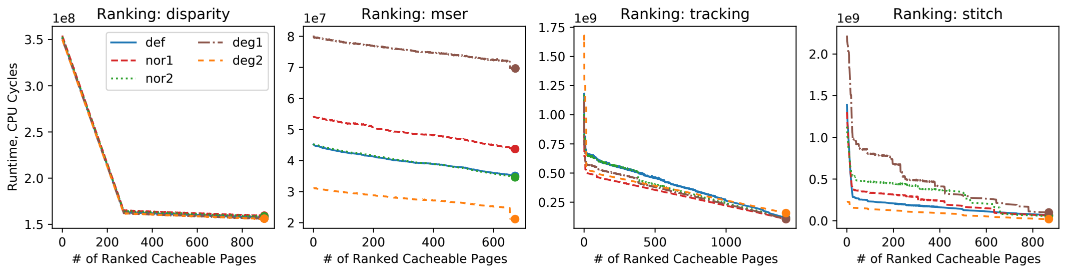


**Figure 9** From left to right, ranking analysis of DISPARITY, MSER, TRACKING, and STITCH with profiles acquired under "def" and varying input images.

## 7.3 Mitigation of Contention-induced Instruction Stall

We hereby want to bring to the attention of the community a previously understudied problem, namely the problem of *contention-induced instruction stall*, or C2IS, for short. We also demonstrate that profile-driven page migration represents an effective strategy to mitigate the problem.

In a nutshell, C2IS can occur in platforms with small L1 caches and shared, unified L2/LLC caches. The problem manifests itself when a process operates in a periodic fashion over a large block of instructions (e.g. a long function) that spans more pages than the size of the L1 instruction cache. For instance, in the target ZCU102 platform, the size of the L1 cache can hold up to eight pages. When such a threshold is crossed, instruction pages spill over L1 and are allocated in L2. But when the L2 is shared, these instruction pages are subject to be evicted by data fetched by any interfering workload. Unlike with missed over data items, an L1 and L2 miss during an instruction fetch cannot be hidden by the micro-architecture, which causes an immediate pipeline stall. The resulting impact on the runtime of the application under analysis can be dramatic.

We observed this effect in the wild and created a synthetic benchmark, namely C2ISBM, to isolate and study the C2IS problem. Our C2ISBM is a process that invokes a long function that spans through 65 `text` pages – i.e., it performs around 64,000 nops. Using as a baseline its solo performance, the runtime increases by a factor of 6.5× when INTERF workload is activated on all the other cores. We extract a profile of the C2ISBM benchmark, where the instruction pages are identified as important. We then configure our system in the PVT+SH mode (see Section 6), and progressively select the instruction pages to be migrated to the PVT pool. Recall that in the PVT+SH configuration, the PVT pool is exclusively allocated to 1/4 of the L2 cache. Gradually migrating the profiled instruction pages to the private pool allows us to gradually de-conflict these pages and to create an equivalent L2 instruction cache with a size that is proportional to the number of migrated pages. The resulting impact on the runtime of the C2ISBM process is plotted in Figure 10. A sharp improvement in runtime can be observed until around 43 pages are migrated. After that, the benchmark becomes unaffected by the interfering workload as around 51 (43 + 8 in the I-cache) of the 65 instruction pages are deterministically present in the cache. It can be noted that a slight runtime increase is visible when more than 64 pages are migrated because the private pool can hold up to 1/4 of the L2 cache size, i.e. 64 pages.

In the presented use-case, being able to identify those pages that are crucial for the application's performance and selectively migrate them to a reserved portion of the cache, space is an efficient solution to the C2IS problem. By contrast, strict coloring would force all the pages of the application to share the same color, which would require the allocation of a much larger cache partition to achieve the same degree of interference mitigation.

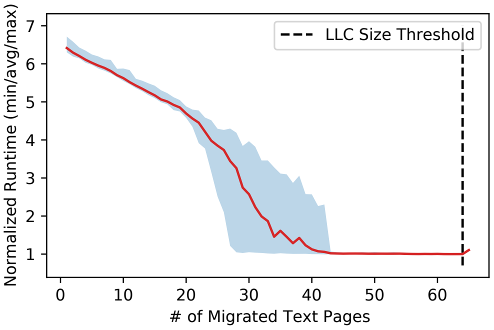


**Figure 10** Inteference mitigation via migration of instruction pages.

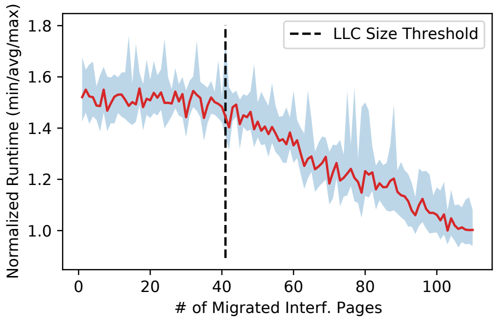


**Figure 11** Interference mitigation via migration of data pages.

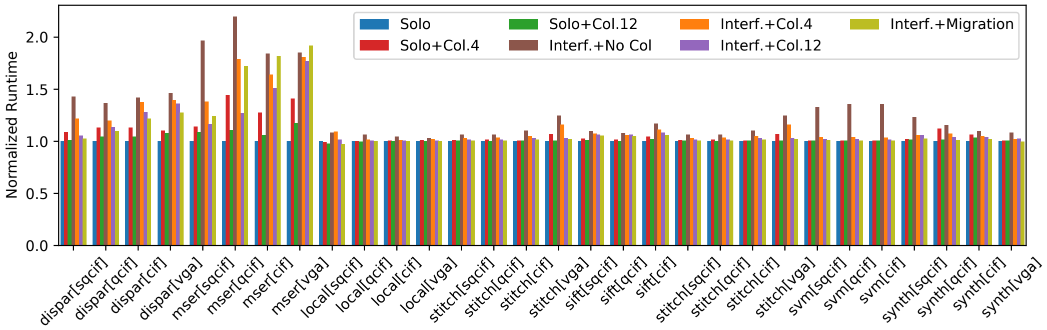


**Figure 12** Mitigation of cache interference with profile-driven migration of interfering data pages.

## 7.4 Controllable Mitigation of Cache Interference

In the last set of experiments, we use our BBProf toolkit and PVT+SH setup to demonstrate that (1) profile-driven interference mitigation is effective for real-world applications, and (2) that, albeit more flexible, its effectiveness is comparable to strict partitioning. For this experiment, we leverage the fact that we can profile the interfering workload and progressively migrate to the private pool the pages that are responsible for the generated cache contention, while we keep the pages of the application under analysis in their original location. Doing this allows cache-sensitive applications to benefit from 12/16 of the LLC space. First, we study the temporal behavior of the Mser benchmark with input size SqCif in Figure 11. On the $x$-axis we track the number of pages migrated to the private pool for each of the three Interf benchmarks – hence the total size of migrated pages is three times this value. The timing behavior of Mser starts to improve after 123 pages from the Interf benchmarks are migrated away. That is because each Interf process accesses a total of 315 pages (420 KB each, see Section 7.1), meaning that only 192 pages are left to migrate, which is exactly 12/16 of the total LLC size.

Lastly, Figure 12 summarizes the behavior of the most interesting benchmarks when a full migration of interfering pages is performed – see last bar of each cluster ("Interf.+Migration"). The resulting runtime is compared against a number of notable cases: (1) the "Solo" case where no Interf is deployed and no cache partitioning is performed. This is also the normalization baseline for all the other cases; (2) and (3) the solo runtime where only four ("Solo+Col.4") or 12 ("Solo+Col.12") cache colors are assigned to the application under analysis; (4) the "Interf.+No Col" case where INTERF is deployed on all the other cores and no partitioning is enforced; (5) and (6) the cases "Interf.+Col.4" and "Interf.+Col.12" that correspond to (2) and (3) but with INTERF active on all the other cores. Profile-driven migration has comparable performance to the case where 12 page colors are dedicated to the application under analysis. In a few cases (see MSER with input sizes SQCIF and QCIF) migration does worse. The reason is likely interference over shared Linux meta-data (e.g. page tables, kernel code and data structures). This kind of contention does not occur with strict partitioning because the INTERF workload operates in a different, fully colored VM.

## 8 Known Limitations

The proposed method and current implementation present a number of limitations. First (i), BBProf is not designed to handle multithreaded applications, or applications comprised by multiple processes with complex data sharing, synchronization and dependencies. Second (ii), for applications that that exhibit strong dependencies between inputs and memory usage, the profile produced by BBProf on a given input might not generalize well to the entire input space. Third (iii), the only piece of information used by BBProf to construct profiles is timing. While this is a deliberate choice that allows BBProf to better generalize on many COTS platforms, we envision that being able to integrate additional metrics (e.g. L1/L2 cache/miss count, consumed main memory bandwidth, energy consumption) might be useful to characterize page importance along additional dimensions beyond timing. In our current implementation, we only provide sample code to integrate calls to Perf [10] APIs during the entry/exit protocols, but more comprehensive handling of the additional metrics that can be collected is required. Fourth (iv), our current implementation relies on a number of Linux-specific features, such as PTRACE and the `proc` filesystem. Thus, while porting to other non-Linux OS's or even bare-metal environments is possible, some heavy re-engineering is required. We expect that PTRACE might need to be replaced with direct interaction with platform-specific debug registers, while memory layout information currently collected via `proc` interfaces might need to be exported at compile-time. Next (v), BBProf does not rely on any hardware features that are not widely available. Nonetheless, a few architecture-dependent features are leveraged, requiring some porting effort when moving to different architectures. These are (1) cacheability manipulation, (2) sampling of CPU clock cycles, and (3) cache maintenance operations. Lastly (vi), the time required to carry out profiling is strictly dependent on the WSS of the target application and on the runtime of the observation segment. Thus, BBProf might become impractically slow at profiling large-footprint and/or long-running applications. Operating on groups of adjacent pages instead of individual pages might mitigate this problem, but the trade-off between loss in granularity and speed-up needs to be investigated.

## 9 Concluding Remarks

In this work, we introduced BBProf, a methodology and toolkit to extract the importance of individual memory pages towards the runtime of a target application. The proposed BBProf does not rely by design on any hardware-specific feature, and thus it can be implemented on any platform where (1) it is possible to change cacheability attributes at a single-page granularity; and (2) it is possible to acquire time samples. Additionally, BBProf can operate on the unmodified, pre-compiled binaries of complex applications, and includes strategies to cope with the use of dynamic memory allocation primitives. We have performed and described an open-source full system implementation and setup on a state-of-the-art high-performance embedded platform. With this setup, we have shown three main aspects. First, that BBProf is capable of extracting the profile of real-world complex vision applications. Second, that the extracted page-level profiles can be used to enact fine-grained shared cache management. Third, that a previously undocumented variant of inter-core interference, namely contention-induced instruction stall can arise in multi-core embedded platforms; in which case profile-driven selective page migration represents an efficient mitigation strategy.

As part of our future work, we intend to relax some of the limitations described above. For instance, we aim at expanding the capabilities of BBProf to capture additional per-page properties. Moreover, we plan to develop strategies to use profiling information for OS-driven mapping of pages to heterogeneous memory resources – e.g., scratchpad memory, FPGA BRAM. Finally, we plan to further improve the level of detail of the collected information by identifying how each page impacts the runtime of multiple code sub-segments.

## References

1 Siemens AG. Jailhouse, 2014. URL: `https://github.com/siemens/jailhouse`.

2 ARM Holdings. Cortex-A53 MPCore technical reference manual (r0p4), 2018. URL: `https://developer.arm.com/documentation/ddi0500/j/`.

3 I. Ashraf, M. Taouil, and K. Bertels. Memory profiling for intra-application data-communication quantification: A survey. In *2015 10th International Design Test Symposium (IDT)*, pages 32–37, 2015. `doi:10.1109/IDT.2015.7396732`.

4 F. Bouquillon, C. Ballabriga, G. Lipari, and S. Niar. A wcet-aware cache coloring technique for reducing interference in real-time systems. *CoRR*, abs/1903.09310, 2019. `arXiv:1903.09310`.

5 D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA, 2003. IEEE Computer Society.

6 J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 194–204, 2009. `doi:10.1109/ECRTS.2009.13`.

7 W. Cohen. Multiple Architecture Characterization of the Build Process with OProfile, 2003. URL: `http://oprofile.sourceforge.net`.

8 J. Corbet, J. Edge, and R. Sobol. Kernel Development. Linux Weekly News – `https://lwn.net/Articles/74295/`, 2004. [Online; accessed 7-May-2019].

9 C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2541940.2541946`.

10 The Linux Foundation. perf: Linux profiling with performance counters. URL: `https://perf.wiki.kernel.org/index.php/Main_Page`.

11 R. Mancuso G. Ghaemi, D. Tarapore. BU Black-box Profiler. `https://github.com/rntmancuso/black-box-profiler`, 2021.

12 G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), 2015. `doi:10.1145/2830555`.

13 G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31th Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:25, Stuttgart, Germany, July 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2019.27`.

14 M. Hassan. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 495–505, 2018. doi:10.1109/RTSS.2018.00062.

15 ARM Holdings. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile (version G.a), 2011.

16 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89, 2013. doi:10.1109/ECRTS.2013.19.

17 H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016. doi:10.1145/2968478.2968480.

18 H. Kim and R. (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1), 2017. doi:10.1145/3092946.

19 N. Kim, B. C. Ward, M. Chisholm, C. 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, 2016. doi:10.1109/RTAS.2016.7461323.

20 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019. doi:10.1109/RTAS.2019.00009.

21 Y. Kwon, X. Zhang, and D. Xu. Pietrace: Platform independent executable trace. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 48–58, 2013. doi:10.1109/ASE.2013.6693065.

22 J. Liedtke, H. Haertig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, page 213, USA, 1997. IEEE Computer Society.

23 C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005. doi:10.1145/1064978.1065034.

24 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

25 S. Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 50(2), 2017. doi:10.1145/3062394.

26 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018. doi:10.1109/ICIT.2018.8352429.

27 N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. doi:10.1145/1273442.1250746.

28 A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, 2015. doi:10.1109/PDP.2015.108.

29 A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 335–348, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1755913.1755947.

30 P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F.J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4), 2012. doi:10.1145/2086696.2086713.

31 RotateRight. Zoom Performance Analysis Tool. URL: http://www.rotateright.com/.

32 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 *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, 2008. doi:10.1109/MICRO.2008.4771796.

33 P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, Los Alamitos, CA, USA, December 2020. IEEE Computer Society. doi:10.1109/RTSS49844.2020.00039.

34 D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso. Observing the invisible: Live cache inspection for high-performance embedded systems. *IEEE Transactions on Computers*, pages 1–1, 2021. doi:10.1109/TC.2021.3060650.

35 S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, October 2009. doi:10.1109/IISWC.2009.5306794.

36 Xilinx, Inc. Zynq ultrascale+ mpsoc data sheet: Overview (v1.8), 2019. URL: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

37 M. Xu, R. Gifford, and L.T. Xuan Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3316781.3317840.

38 Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, 2014. doi:10.1145/2628071.2628104.

39 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. doi:10.1109/RTAS.2013.6531079.

40 X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 89–102, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1519065.1519076.

# nDimNoC: Real-Time D-dimensional NoC

**Yilian Ribot González**
CISTER Research Centre, ISEP, Polytechnic Institute of Porto, Portugal

**Geoffrey Nelissen**
Eindhoven University of Technology, The Netherlands

**Eduardo Tovar**
CISTER Research Centre, ISEP, Polytechnic Institute of Porto, Portugal

## Abstract

The growing demand of powerful embedded systems to perform advanced functionalities led to a large increase in the number of computation nodes integrated in Systems-on-chip (SoC). In this context, network-on-chips (NoCs) emerged as a new standard communication infrastructure for multi-processor SoCs (MPSoCs). In this work, we present nDimNoC, a new D-dimensional NoC that provides real-time guarantees for systems implemented upon MPSoCs. Specifically, (1) we propose a new router architecture and a new deflection-based routing policy that use the properties of circulant topologies to ensure bounded worst-case communication delays, and (2) we develop a generic worst-case communication time (WCCT) analysis for packets transmitted over nDimNoC. In our experiments, we show that the WCCT of packets decreases when we increase the dimensionality of the NoC using nDimNoC's topolgy and routing policy. By implementing nDimNoC in Verilog and synthesizing it for an FPGA platform, we show that a 3D-nDimNoC requires ≈5-times less silicon than routers that use virtual channels (VC). We computed the maximum operating frequency of a 3D-nDimNoC with Xilinx Vivado. Increasing the number dimensions in the NoC improves WCCT at the cost of a more complex routing logic that may result in a reduced operating clock frequency.

**2012 ACM Subject Classification** Computer systems organization → Real-time systems; Networks → Network on chip

**Keywords and phrases** Real-Time Embedded Systems, Systems-on-Chips, Network-on-Chips, Worst-Case Communication Time

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.5

**Funding** This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDP/UIDB 04234/2020); also by FCT and the ESF (European Social Fund) through the Regional Operational Programme (ROP) Norte 2020, under PhD grant 2020.06898.BD.

## 1 Introduction

These days, SoCs include more and more heterogeneous processing elements that execute dedicated functions in parallel. Traditional shared communication buses, which used to connect all the computation nodes together, are a major performance bottleneck of modern SoCs. Therefore, NoCs emerged as a new standard communication infrastructure for SoC as they present a scalable and versatile solution for systems with a high level of parallelism [2, 15].

The literature on NoCs is extensive. However, real-time systems add new constraints on the NoC infrastructures. In addition to ensure that messages arrive at their destination in a correct fashion, real-time NoCs must guarantee that packet transmissions respect strong timing constraints [16]. Over the years, there have been several attempts to design real-time NoCs by considering different approaches. A large body of solutions consider a mesh topology and rely on wormhole switching with VCs. That strategy leads to powerful NoC infrastructures with bounded WCCT but they rely extensively on buffers and virtual channels to provide timing guarantees. This makes them expensive to implement in terms of silicon footprint, and increases their power consumption.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 5; pp. 5:1–5:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These last years, buffer-less NoCs have gain popularity as an alternative to VC-based NoCs. Buffer-less NoCs are compact; their implementation cost and power consumption are lower than traditional approaches. Therefore, they are more suitable to (embedded) systems with area and/or power consumption constraints. In [39] and [31] two novel buffer-less deflection-based real-time NoCs called HopliteRT and HopliteRT* were proposed. They ensure predictable timing behaviors, accommodates dynamic workload and have an extremely low hardware consumption footprint. Noticeably, HopliteRT* uses the characteristics of a circulant topology to ensure bounded worst-case communication delays.

NoCs are an attractive and promising alternative for the traditional shared-buses. Yet, most of the existing literature for real-time systems focuses on 2-Dimensional NoCs (2D-NoCs), i.e., where routers are connected according to a mesh or torus topology for example. However, in a non-real-time setting, Romanov [32] shows that circulant topologies possess better characteristics over traditional mesh and torus topologies. Circulant topologies are a type of n-dimensional topologies for networks. Thus, in this work, we explore the design of n-dimensional NoCs architectures compatible with real-time systems requirements.

This line of research is also motivated by the recent evolution in the integrated circuit (IC) industry. Indeed, three-dimensional integrated circuits (3D-ICs) seem to be the future of ICs [19, 5, 20, 33, 29]. 3D-ICs achieve higher performance, while reducing average interconnection length; provide higher packing density thanks to the added third dimension; reduce power consumption; and enhance computation bandwidth. Hence, there is currently a drive towards creating new powerful NoCs solutions that meet the requirements of future large-scale MPSoCs by combining the advantages of 3D integration and NoC architecture.

**Contribution.** We propose nDimNoC, a new D-dimensional NoC that provides real-time guarantees for systems implemented upon MPSoCs, reduces average network communication latency and provides greater flexibility compared to more traditional 2D NoCs. The main contributions of our work are: (1) to design a new buffer-less router architecture that allows synthesizing D-dimensional NoCs; (2) to propose a new deflection-based routing policy that uses the characteristics of D-dimensional circulant topologies to ensure bounded worst-case communication delays; (3) to develop a generic WCCT analysis for packets transmitted over nDimNoC; (4) to implement a 3D version of nDimNoC in Verilog (a hardware description language) that can be instantiated on a real FPGA platform; and (5) to assess our new design against related works in terms of computed WCCT bounds and hardware requirements.

## 2 Related work

Most 2D-NoC solutions rely on wormhole switching with virtual channels (VCs) (e.g., CONNECT [27], IDAMC [37]). In [34], Shi et al. propose an analysis of the worst-case network latency for a new real-time fixed priority preemptive wormhole NoC in which each priority level is assigned its own VC. Several variations of that approach were proposed over the years [36, 7, 37, 6, 30], for instance, handling the case where several flows share the same priority [21], changing the routing policy to EDF [25] or supporting communication flows with different criticality levels [3, 18]. The complexity of those designs and their routing policies led to complex WCCT analyses inspired by both the classic real-time system theory [41, 42, 17, 26] and Network Calculus [10, 11].

In [24], a new type of NoC called SBT-NoC was proposed. In this work, Nikolic et al. introduced a global arbitration protocol inspired by the CAN protocol. Theoretical results are promising but this NoC solution has not been implemented in a real platform yet.

Recently, Wasly et al. in [39] proposed a new buffer-less NoC for real-time systems. Their NoC is called HopliteRT. The design of HopliteRT ensures that the WCCT of packets is upper-bounded. HopliteRT* is an evolution of HopliteRT proposed in [31]. It introduces a notion of quality of service in the routing policy and uses a circulant topology in order to improve the packets' WCCT in comparison to HopliteRT.

In [32], various routing strategies, i.e., table routing, Clockwise routing and Adaptive routing, were studied for two-dimensional ring circulant networks. The author shows that several characteristics of NoCs are improved in comparison to mesh and torus topologies when circulant graphs are used as a topological basis.

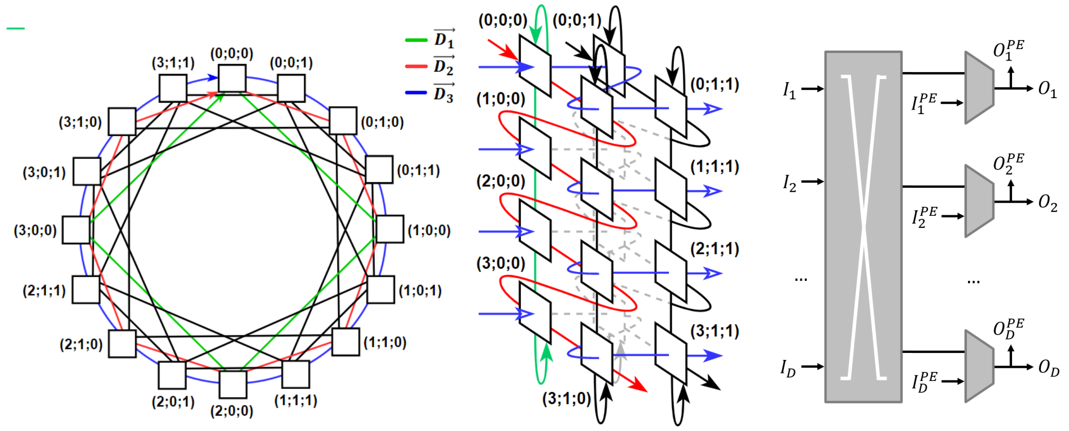
From a 3D-NoC perspective, Park et al. [28] proposed a Multi-layered on-chip Interconnect Router Architecture (MIRA). Their approach assumes 3D processor designs (i.e., processor cores partitioned into multiple layers), and is therefore inadequate for existing highly optimized 2D processor designs. In [9], Ghidini et al. presented a 3D-NoC mesh architecture called Lasio relying on wormhole switching with FIFO queues. In order to minimize packet communication latency and NoC area, Tiny 3D mesh NoC was later proposed in [22]. Tiny NoC reduces the number of routers and links in the network by connecting multiple programming elements to the same router. This solution minimizes the total NoC area as compared to Lasio NoC, however, average packets latency improves only when there are few flows and/or under a low packet injection rate. In [4], a 3D NoC architecture based on De-Bruijn graph was proposed. Tree-based interconnect architectures have been also considered in some works [13, 14, 12]. However, they are very complex to implement due to their irregular and complex network topologies. In [8], a NoC/Bus-based hybrid 3D architecture was proposed, but the approach suffers from low throughput due to inefficient hybridization between the NoC and bus media.

To the best of our knowledge, none of the 3D-NoC solutions developed so far targets real-time systems. Therefore, they do not provide guaranteed upper-bounds on the packets WCCT, and do not come with a WCCT analysis. In this work, we develop a new real-time D-dimensional NoC (with $D \geq 2$) and its associated timing analysis.

## 3 System model

In this paper, we assume a system composed of $N$ programming elements $\{\pi_1, ..., \pi_N\}$. Each programming element $\pi_q$ is connected to a different router $R_q$ of a D-dimensional NoC. The coordinates of a router $R_q$ in the D-dimensional network are noted $(r_1^q, r_2^q, ..., r_D^q)$. Each programming element $\pi_q$ injects a set of $n^q$ communication flows $F^q = \{f_1^q, f_2^q, ..., f_n^q\}$ into the network. A communication flow $f_j$ is defined by the parameters $\{(s_1^j, s_2^j, ..., s_D^j), (d_1^j, d_2^j, ..., d_D^j), C_j, T_j\}$. A communication flow $f_j$ generates a potentially infinite number of packets that are injected at coordinates $(s_1^j, s_2^j, ..., s_D^j)$ of the NoC and must reach the programming element at coordinates $(d_1^j, d_2^j, ..., d_D^j)$. $f_j$ respects a minimum inter-arrival time $T_j$ between the generation of every two packets. Each packet sent by flow $f_j$ is divided in $C_j$ flits that are sequentially injected in the network. Each flit has a size $S_{flit}$ (in bits). We assume that all the routing information is encoded in each flit of the packet, i.e., there is no distinction between header, body or tail flits. The routing information is the coordinates of the destination programming element of the associated flow.

In the rest of this paper, we use the notations $R_{orig}(f_j)$ and $R_{dest}(f_j)$ to refer to the origin and destination router of flow $f_j$, respectively. That is, $R_{orig}(f_j)$ has coordinates $(s_1^j, s_2^j, ..., s_D^j)$ and $R_{dest}(f_j)$ has coordinates $(d_1^j, d_2^j, ..., d_D^j)$.



**(a)** Circulant topology $C(16; 1, 2, 4)$.

**(b)** Equiv. 4x2x2 grid-based 3D-network.

**(c)** nDimNoc router architecture.

**Figure 1** nDimNoc's topology and router architecture.

## 4 nDimNoC architecture

In this section, we present nDimNoC. More specifically, we describe: (1) the network topology, (2) the router architecture, and (3) the routing policy. We later provide the timing analysis for nDimNoC in Section 5.

### 4.1 NoC topology

Consider a network composed of $N$ routers $R_0$ to $R_{N-1}$. In nDimNoC, the routers are connected together according to a ring circulant topology $C(N; g_1, g_2, ..., g_D)$ where $g_1 = 1$, $N$ is the total number of routers, $D$ indicates the dimensionality of the network, and $g_1, g_2, ..., g_D$ are the *generatrices* of the network. We assume that the generatrices follow the following properties: $1 = g_1 < g_2 < ... < g_D < N$, and that their values are harmonic (i.e., for any pair of generatrices $g_i$ and $g_j$ such that $i < j$, $g_i$ is a divider of $g_j$). Under the circulant topology $C(N; 1, g_2, ..., g_D)$, all routers have $D$ inputs $I_1, I_2, ..., I_D$ and $D$ outputs $O_1, O_2, ..., O_D$ for inter-routers communications. All routers are connected by a single unidirectional ring using one of their inputs and one of their outputs (see blue line in Figure 1a). Then, each router is also connected to the routers that are $g_2, g_3, ..., g_D$ hops away on the ring (see red, green and black lines in Figure 1a). Formally stated, for each router $R_q$ (with $0 \leq q < N$), its $u^{th}$ output port $O_u$ ($1 \leq u \leq N$) is connected to the $u^{th}$ input port $I_u$ of the router $R_{(q+g_u) \bmod N}$.

A circulant network $C(N; 1, g_2, ..., g_D)$ may also be represented as a $S_1$x$S_2$x...x$S_D$ grid-based D-dimensional network, where $S_1, S_2, ...S_D$ correspond to the number of routers on the dimension $\overrightarrow{D_1}$,$\overrightarrow{D_2}$,...,$\overrightarrow{D_D}$, respectively. The size of the network on each dimension can be computed as follows $S_1 = \frac{N}{g_D}$, $S_2 = \frac{g_D}{g_{D-1}}$, $S_3 = \frac{g_{D-1}}{g_{D-2}}$, ..., $S_D = \frac{g_2}{g_1}$. The coordinates $(r_1^q, r_2^q, ..., r_D^q)$ of a router $R_q$ defines the position of the router $R_q$ in the grid representation.

As an example, Fig. 1a shows the circulant network $C(16; 1, 2, 4)$. In Fig. 1b, we provide the equivalent representation as a 4x2x2 grid-based 3-Dimensional network of the circulant network shown in Fig. 1a. The red, green, and blue links in Fig. 1a correspond to the red, green, and blue links in Fig. 1b, respectively.

In the rest of this paper, we often reason about the position $pos^q$ of a router $R_q$ on the main unidirection ring of the circulant topology. That position can be inferred from the coordinates of the router in the grid topology as follows

$$pos^q = \sum_{k=1}^{D} r_k^q \times g_{D-k+1} \tag{1}$$

To simplify some of our further discussions, we define the helping function $\text{dist}(R_q, R_m)$ as the distance between routers $R_q$ and $R_m$ on the main ring, i.e.,

$$\text{dist}(R_q, R_m) = (pos^m - pos^q + N) \bmod N \tag{2}$$

Note that the following properties hold for circulant topologies.

▶ **Property 1.** *Let $R_l$ be a router at which flit $p$ is located. After one hop on dimension $\overrightarrow{D_u}$ of the network, flit $p$ reaches a router $R_m$ located $g_{D-u+1}$ steps further on the main ring of the network, i.e.,* $\text{dist}(R_l, R_m) = g_{D-u+1}$.

Finally, we define $\text{ring}_u(R_q)$ as the set of routers that are on the same ring of dimension $\overrightarrow{D_u}$ as $R_q$. That is,

$$\text{ring}_u(R_q) = \{R_l \mid \forall b \in [u+1, D],\, r_b^l = r_b^q\} \tag{3}$$

As an example, let $R_0$ be the router at coordinates $(0;0;0)$ in Figure 1a, then all the routers connected by the green links are in $\text{ring}_1(R_0)$, and all the routers connected by red links are in $\text{ring}_2(R_0)$.

## 4.2 Router architecture

In order to reduce implementation cost in terms of hardware resources utilization and network analysis complexity, nDimNoC does not use VCs and does not rely on extensive buffer.

As we discuss in the previous section, nDimNoC routers have $D$ inputs (i.e., $I_1$, $I_2$, ..., $I_D$) and $D$ output ports (i.e., $O_1$, $O_2$, ..., $O_D$) connected to neighboring routers to allow for inter-routers communication (see Fig. 1c). In addition, all routers also have $D$ input ports (i.e., $I_1^{PE}$, $I_2^{PE}$, ..., $I_D^{PE}$) that may be used by the programming element to inject packets into the network. Therefore, in total, each router has $2 \times D$ input ports and $D$ output ports. A programming element can inject packets on any of the $D$ dimensions $\overrightarrow{D_1}$,$\overrightarrow{D_2}$,...,$\overrightarrow{D_D}$ of the network by using the input ports $I_1^{PE}, I_2^{PE}, ...I_D^{PE}$, respectively. Therefore, several packets may be injected to different dimensions simultaneously. Thus, the waiting times suffered by the packets inside the programming elements decreases. Indeed, in solutions that support a single input port to inject packets into the network, all packets compete for the same input port. In nDimNoc, however, a packet that is waiting to be injected into the network only conflicts with the subset of packets that must be injected to the same input port $I_u^{PE}$.

▶ **Property 2.** *In this paper, we assume that a flit of a flow $f_j$ with origin and destination coordinates $(s_1^j, s_2^j, ..., s_D^j)$ and $(d_1^j, d_2^j, ..., d_D^j)$ is injected in the network using port $I_u^{PE}$* ***if and only if*** *$s_u^j \neq d_u^j$ and $\forall x \mid u < x \leq D, s_x^j = d_x^j$.*

From Property 2, we get that all the packets of a given flow will be injected using the same input port.

The ports $O_1$, $O_2$,..., $O_D$ of a router are connected to the ports $I_1$, $I_2$,..., $I_D$ of its neighboring routers, but also serve as inputs to the programming elements. That is, the programming element connected to a router can reads packets from all the output ports

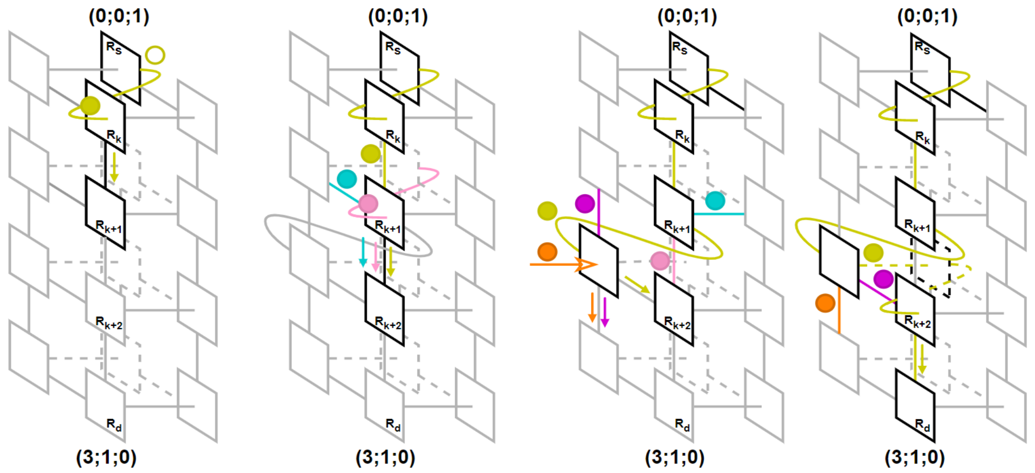
**Table 1** Generic routing policy table of nDimNoC with D dimensions.

| Rule | Flows requests | Conflicting requests | Routing decisions | Explanation |
|---|---|---|---|---|
| 1 | $I_D \rightarrow O_D$ | None | $I_D \rightarrow O_D$ | No contention over $O_D$. |
| 2 | $I_D \rightarrow O_1$ | Any | $I_D \rightarrow O_1$ | $I_D \rightarrow O_1$ always wins. |
| 3 | $I_u \rightarrow O_u$ | None | $I_u \rightarrow O_u$ | No contention over $O_u$. |
| 4 | | $I_{u-1}$ deflected to $O_u$ | $I_u \rightarrow O_{u+1}$ | Flows coming from the $I_{u-1}$ and $I_u$ ports conflict over $O_u$. $I_{u-1} \rightarrow O_u$ always wins over $I_u \rightarrow O_u$. The flow coming from the $I_u$ port is deflected to the $O_{u+1}$ port. |
| 5 | $I_u \rightarrow O_1$ | None **or** $I_{v<u} \rightarrow O_1$ | $I_u \rightarrow O_1$ | No flow entering by a port on a higher dimension than $I_u$ requests $O_1$. $I_u \rightarrow O_1$ wins. |
| 6 | | $I_{v>u}$ | $I_u \rightarrow O_{u+1}$ | A flow entering by a port on a higher dimension than $I_u$ wins $O_1$. The flow coming from the $I_u$ port is deflected to the $O_{u+1}$ port. |
| 7 | $I_u^{PE} \rightarrow O_u$ | None | $I_u^{PE} \rightarrow O_u$ | There is no flow coming from another port that requests $O_u$. The flow on $I_u^{PE}$ is injected in the network via $O_u$. |
| 8 | | $I_v \rightarrow O_u$ and/or $I_{u-1}$ deflected to $O_u$ | None | The flow waiting on the $I_u^{PE}$ port conflicts over the $O_u$ port with flows coming from neighboring routers. Since flows from $I_u^{PE}$ have the lowest priority, the flow waiting on the $I_u^{PE}$ port is not injected in the network. |

$O_1$, $O_2$,..., $O_D$. We show this property (i.e., that the programming element has read-access to all output ports of the router) using the notations $O_1^{PE}$, $O_2^{PE}$, ..., $O_D^{PE}$ in Fig. 1c. We assume that a programming element can read packets from several different output ports simultaneously. This may be done by considering that each programming element has a FIFO queue connected to each port $O_u^{PE}$ (with $1 \leq u \leq D$). We assume that those FIFO queues are large enough to prevent back pressure in the network. Although this design solution may lead to increased router programming logic complexity, it avoids the extra cost of implementing expensive exit multiplexers.

We consider that each programming element has also a FIFO queue connected to each port $I_u^{PE}$. These queues store flits that are pending to be injected in the network. Note that, the FIFO queues connected to each $O_u^{PE}$ and $I_u^{PE}$ port could be implemented in software or hardware. Their specific implementation is irrelevant to the matter discussed in this work.

We assume that no traffic injection regulator exists at the programming elements. Therefore, they can inject flits into the network as fast as possible. Nonetheless, we assume that each flow $f_j$ can have a maximum of one packet in the FIFO queue pending to be injected in the network at any moment in time. That is, only after a packet is injected, a new packet from the same flow $f_j$ can be stored in the FIFO queue. The implicit assumption is that the minimum inter-arrival time $T_j$ between the generation of every two packets of $f_j$ is larger or equal than the worst-case packet injection time $wcit_j$ of that flow, i.e., $\forall f_j, T_j \geq wcit_j$. Note that, the restriction is only related to the content of the FIFO queues at the injection ports



**(a)** Packet request to use $I_1$.

**(b)** Situation after routing arbitration and new packet requests.

**(c)** Situation after routing arbitration and new packet requests.

**(d)** Situation after routing arbitration.

**Figure 2** nDimNoc's routing policy example.

and does not limit the number of in-flight packets in the network. That is to say, several packets from the same flow $f_j$ can be traveling around the network at the same time. Also note that this assumption is less constraining than those made in many works on real-time NoCs that assume periods larger than the worst-case communication time (of which the injection time is just one component).

## 4.3 Routing policy

Consider a flit that must travel from router (0;0;0) to router (2;0;0) in the example network of Figure 1a. It will reach its destination faster if it travels on the green link than if it hops through the red or blue links. Since the green, red and blue links correspond to dimensions $\overrightarrow{D_1}$,$\overrightarrow{D_2}$, and $\overrightarrow{D_3}$, respectively, it is equivalent to say that it is faster for the flit to travel on a dimension of lower order. nDimNoC's routing policy simply builds upon that property. Additionally, it uses the idea of deflection routing [1] to avoid the cost of packet buffering. The approach is as follows.

Consider a flit of a flow $f_j$ that has been injected at the origin router $(s_1^j, s_2^j, ..., s_D^j)$ and with destination $(d_1^j, d_2^j, ..., d_D^j)$. As mentioned in Section 4.2, the programming element injects that flit on port $I_u^{PE}$ such that $s_u^j \neq d_u^j$ and $\forall x \mid u < x \leq D,\ s_x^j = d_x^j$.

If the flit was transmitted in isolation (i.e., without any interfering flow), it would travel along the dimension $\overrightarrow{D_u}$ of the network by entering in each router by input port $I_u$ and requesting output port $O_u$. Then, when it reaches the first router $R_k$ with the same coordinates $r_2^k$, $r_3^k$,...,$r_D^k$ as its destination (i.e., $r_b^k = d_b^j, \forall b \in [2, D]$), it would request the output port $O_1^k$ and travel along dimension $\overrightarrow{D_1}$ until reaching its destination. It results that flits entering by input port $I_u$ (such that $2 \leq u \leq D$) may only request the output port $O_u$ or $O_1$. Flits entering by the input port $I_1$ may only request the output port $O_1$.

If there is interfering traffic, nDimNoC's routing policy allows flits to be "deflected" to make place for "higher priority" traffic. Two such scenarios may happen:

1. If multiple flits entering by different input ports request the output port $O_1$ at the same time, nDimNoC always gives the highest priority to the flit that entered by the input port with highest dimension (i.e., $I_D$ wins over $I_{D-1}$, which wins over $I_{D-2}$, etc.). Consider

two flits entering by ports $I_u$ and $I_v$ such that $u < v$ and that request output port $O_1$. Then, the flit entering by $I_v$ exists through $O_1$, and the flit entering by $I_u$ exists through $O_{u+1}$. We say that the flit that entered by $I_u$ is deflected to dimension $\overrightarrow{D_{u+1}}$.

2. A flit entering by port $I_u$ that was deflected to the output port $O_{u+1}$ may now conflict for port $O_{u+1}$ with a flit coming from $I_{u+1}$ and that requests $O_{u+1}$ at the same time. Under this contention scenario, the flit coming from $I_u$ and that was deflected towards $O_{u+1}$ wins the right to use $O_{u+1}$ and the flit coming from $I_{u+1}$ is deflected towards the output port $O_{u+2}$.

Note that deflections redirect deflected flits on longer paths towards their destination. However, the topology presented in Section 4.1 ensures that it still progresses towards its destination router. Therefore, nDimNoC's routing policy is deadlock-free and livelock-free. Furthermore, after each deflection, a flit's priority to request output port $O_1$ in a future router increases (since flits traveling on higher dimensions have higher priorities). Therefore, its probability to be able to later travel on a shorter route increases too.

Finally, flits injected by the programming element (i.e., flits entering by any port $I_u^{PE}$), always have the lowest priority and must wait for the respective port $O_u$ to be free. Table 1 summarizes the routing policy of a D-dimensional nDimNoC.

**Example.** Consider a 4x2x2 3-dimensional nDimNoC (i.e., $D = 3$) (see Figure 2a-2d). Each 3D-nDimNoC router has six input ports ($I_1$, $I_2$, $I_3$, $I_1^{PE}$, $I_2^{PE}$, and $I_3^{PE}$) and three output ports ($O_1$, $O_2$, and $O_3$). Consider also a flit of a flow $f_j$ (yellow flit in Figure 2a) with origin and destination coordinates $(0; 0; 1)$ and $(3; 1; 0)$, respectively. Since $s_3^j \neq d_3^j$, the flit is injected via input port $I_3^{PE}$ (see Figure 2a). The flit then travels along the dimension $\overrightarrow{D_3}$ until it reaches router $R_k$ with the same coordinates $r_2^k$, $r_3^k$,...,$r_D^k$ as its destination, i.e., $r_2^k = d_2^j = 1$ and $r_3^k = d_3^j = 0$ (see Figure 2a). In $R_k$, the flit enters by input port $I_3$ and requests output port $O_1$ to travel along the dimension $\overrightarrow{D_1}$ until its destination (see Figure 2b). According to rule 2 of nDimNoC's routing policy (see Table 1), it has the highest priority to use $O_1$ and therefore enters the router $(1; 1; 0)$ (next router to $R_k$ on dimension $\overrightarrow{D_1}$) by its port $I_1$, and requests port $O_1$ (see Figure 2b). If a flit enters by the input $I_2$ (blue flit in Figure 2b) and/or $I_3$ port (pink flit in Figure 2b) and request $O_1$ at the same time as the yellow flit, then the yellow flit is deflected to the output port $O_2$ (see Figure 2c and rule 6 in Table 1). Thus, it must now travel along dimension $\overrightarrow{D_2}$ until it reaches the same router as it would have if it could have used the $O_1$ port instead. Note that the yellow flit may still suffer additional deflections to dimension $\overrightarrow{D_3}$ in any router it reaches while traveling along dimension $\overrightarrow{D_2}$ as it is the case on Figure 2c where both a flit entering by the $I_1$ port (violet flit) and a flit entering by the $I_3$ port (orange flit) request the $O_1$ port. Then, the request $I_3 \rightarrow O_1$ wins over the other requests and the flits entering by the $I_1$ and $I_2$ ports are deflected to the $O_2$ and $O_3$ ports, respectively (see Fig. 2d and rule 4 in Table 1).

## 5 Bound on the worst-case communication time

In Section 4, we presented nDimNoC's design. In this section, we present an analysis for the worst-case communication time (WCCT) between two processing elements connected with nDimNoC. The WCCT of a packet is defined as the sum of the maximum amount of time *wcit* during which the last flit of the packet must wait in the programming element before to be injected into the network, and the maximum amount of time *wctt* taken by any flit of the packet to traverse the network and reach its destination. We refer to those as the worst-case injection time ($wcit_j$) and the worst-case traversal time ($wctt_j$) of flow $f_j$, respectively. Then, the WCCT of a packet of a flow $f_j$ is defined as:

$$wcct_j = wcit_j + wctt_j, \tag{4}$$

## 5.1 Worst-case and best-case traversal time

In this section, we compute bounds on the worst- and best-case traversal time of a flit $p$ (abbreviated WCTT and BCTT, respectively). A bound on the WCIT is later derived in Section 5.2.

As discussed in the previous section, a flit $p$ of flow $f_j$ that travels through nDimNoC can be deflected in any router on its path to its destination, but there is only a limited set of routers in which it can *actively* request to change the dimension it travels along. Those routers are (i) the origin router of the flit with coordinates $(s^j_1, s^j_2, ..., s^j_D)$, and (ii) every router $R_k$ on the path of $p$ such that its coordinates respect $r^k_b = d^j_b$, $\forall b \in [2, D]$. We formally denote this set of routers by $\mathcal{R}$ where

$$\mathcal{R} = \{R_k \mid \forall l \in [1, D],\ r^k_l = s^j_l \ \vee\ \forall b \in [2, D], r^k_b = d^j_b\}. \tag{5}$$

Note that the destination router of flow $f_j$ is obviously in $\mathcal{R}$ since that router has the coordinates $(d^j_1, d^j_2, ..., d^j_D)$.

As will be shown later in this section, the routing decisions in the routers in $\mathcal{R}$ are the only ones that must be analyzed to get a bound on the BCTT and WCTT of a flit $p$.

We use a directed acyclic graph (DAG) $G$ to compute the WCTT and BCTT of a flit of a flow $f_j$ that traverses an D-dimensional nDimNoC. A DAG $G = (V, E)$ is formed by a set of vertices $V$ and a set of edges $E$. Each edge $e \in E$ connects two vertices $u$ and $v$ in $E$. We note $e = (u, v)$. Each edge is assigned a weight $w(u, v)$.

The DAG compactly represents all the routes that the flit $p$ may potentially follow (from its origin to its destination) when it traverses nDimNoC. Each vertex $v$ in the DAG $G$ represents one input port of a router in $\mathcal{R}$. Let $R(u)$ and $I(u)$ be the router and the input port of the router represented by vertex $u$ in the graph, then we note $u = (R(u), I(u))$. Each edge $e = (u, v) \in E$ connecting vertices $u$ and $v$ represents a possible path taken by the flit from input port $I(u)$ of router $R(u)$ to the input port $I(v)$ of another router $R(v)$ on its path. The weight of the edge $e = (u, v)$ is the maximum number of hops from $I(u)$ to $I(v)$ according to that path. Additionally, we label each edge $e = (u, v)$ with the specific output port taken by the flit in router $R(u)$.

**Example.** Figure 3 shows the DAG of the example of Section 4.3 (Figure 2). It shows the potential paths that a flit of a flow $f_j$ may follow from the origin router $(0; 0; 1)$ to the destination router $(3; 1; 0)$ when it traverses a 4x2x2 3-dimensional nDimNoC. The source vertex $v_0$ at level 0 of the graph represents the input port $I^{PE}_3$ of the origin router $R_s$ at which the flit is injected by the programming element. Since the flit $p$ is injected by $I^{PE}_3$, it can only exist by output port $O_3$ of $R_s$. $R_k$ is the first router $p$ reaches after leaving $R_s$ where $p$ may request output port $O_1$. Flit $p$ may only enter $R_k$ by the input port $I_3$. Vertex $v_1$ on Level 1 represents input port $I_3$ of $R_k$. The weight $w_1$ is the number of hops the flit $p$ does from the $O_3$ port of $R_s$ to the $I_3$ port of $R_k$. In router $R_k$, the flit enters by the $I_3$ port and requests the $O_1$ port to travel along dimension $\overrightarrow{D_1}$. According to rule 2 of nDimNoC's routing policy (see Table 1), the routing decision for that request is always $I_3 \rightarrow O_1$ (because any flit entering by the $I_3$ port has the highest priority to use the $O_1$ port in a 3-dimensional

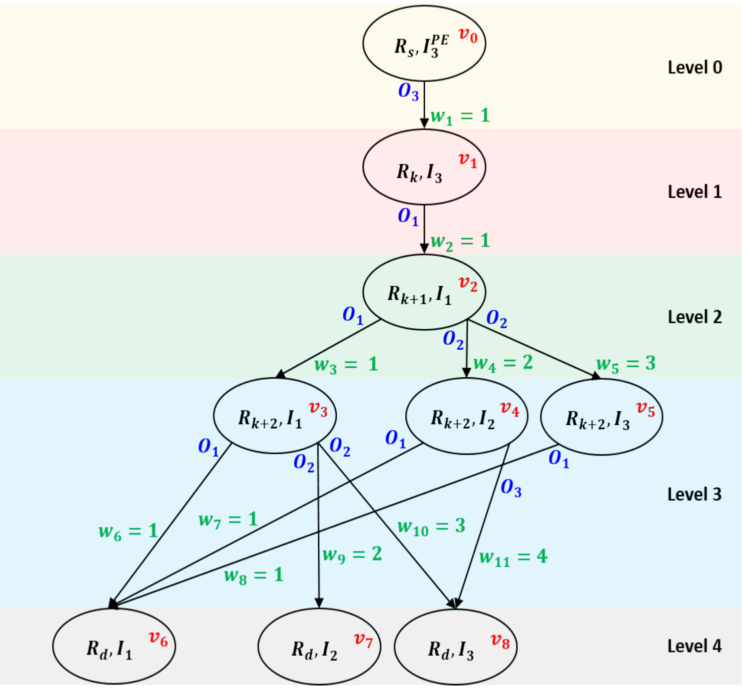


**Figure 3** DAG of the potential trajectories that a flit may take from the origin (0; 0; 1) to the destination (3; 1; 0) when it traverses a 4x2x2 3-D nDimNoC.

nDimNoC). Therefore, $p$ reaches router $R_{k+1}$ in one hop, and certainly enters $R_{k+1}$ by port $I_1$. Since $R_{k+1}$ is also in $\mathcal{R}$, it is represented by vertex $v_2$. According to Table 1, two different routing decisions may be taken in $R_{k+1}$: (1) $p$ is routed to the $O_1$ port if there is no conflict over $O_1$ (see rule 5 in Table 1); or (2) $p$ is deflected to the $O_2$ port if there is one or more flows coming from other ports that request the $O_1$ port at the same time as $p$ (see rule 6 in Table 1). If situation (1) happens (i.e., the flit under analysis is routed to the $O_1$ port), it enters the router $R_{k+2}$ using port $I_1$. We represent the $I_1$ port of $R_{k+2}$ as vertex $v_3$ in Level 3. If situation (2) occurs (i.e., the flit is deflected to the $O_2$ port), it may enter router $R_{k+2}$ from: (1) the $I_2$ port if it suffer no further deflection to reaching $R_{k+2}$ (see rule 3 in Table 1) or (2) the $I_3$ port if it suffers more deflections on its path to $R_{k+2}$ (see rule 4 in Table 1). We represent the $I_2$ and $I_3$ ports of $R_{k+2}$ as vertices $v_4$ and $v_5$ in the level 3 of the graph, respectively. Considering the potential routing decisions to which the flit $p$ may be subjected after it enters $R_{k+2}$ by the ports $I_1$, $I_2$ or $I_3$, $p$ may reach its destination router $R_d$ by input ports $I_1$, $I_2$ or $I_3$ (vertices $v_6$, $v_7$, and $v_8$ on Level 4) in a maximum number of hops represented by the weights of the edges connecting the vertices of level 3 to those of level 4. Note that, the flit will always be received by the programming element regardless of the routing decision taken in the destination router.

After building the graph $G$ as exemplified above, the WCTT of flit $p$ is the longest weighted path in graph $G$, and its BCTT is the shortest weighted path in $G$. For the example of Figure 3, the WCTT is thus equal to 8 and corresponds to the case where the flit $p$ follows the path represented by vertices $v_0$, $v_1$, $v_2$, $v_4$ and $v_8$. Similarly, taking the shortest weighted path, we get that the BCTT of $p$ is 4 in that example. Note that the WCTT may not always be obtained when the flit experiences its maximum number of deflection, hence the need for building the full graph $G$.

Following the reasoning above, the graph $G$ can systematically be built using Algorithm 1. Algorithm 1 uses Lemmas 1 to 5 to compute the set of input and output ports to which the flit $p$ may be routed in each router in $\mathcal{R}$, and to compute the weight of each edge. We now present and prove those lemmas.

In the following, we denote by $R_{cur}$ and $R_{next}$ any two routers in $\mathcal{R}$ such that $R_{next}$ is the first router in $\mathcal{R}$ reached by $p$ after leaving $R_{cur}$.

▶ **Lemma 1.** *A flit $p$ of a flow $f_j$ that enters router $R_{cur}$ by port $I_u^{PE}$ will be routed to the output port $O_u$.*

**Proof.** By rule 7 of nDimNoC's routing policy (Table 1). ◀

**Algorithm 1** Building the DAG of the potential trajectories.

```
Input: flow $f_j$;
Output: $V$, $E$;
$V \leftarrow \emptyset$; $E \leftarrow \emptyset$;
Build set $\mathcal{R}$ according to Equation (5);
$R_{cur} \leftarrow$ source router of $f_j$;
$I_u \leftarrow$ input port by which $f_j$ is injected in its source router according to Property 2;
Create vertex $v_{cur} = (R_{cur}, I_u)$;
$V \leftarrow V \cup \{v_{cur}\}$;
$\Gamma^I \leftarrow \{I_u\}$;
$\Gamma^I_{new} \leftarrow \emptyset$;
while $R_{cur}$ is not the destination router of $f_j$ do
    $R_{next} \leftarrow$ first router in $\mathcal{R}$ reached by any flit of $f_j$ after it leaves $R_{cur}$;
    foreach $I_{cur} \in \Gamma^I$ do
        $v_{cur} \leftarrow$ get vertex $(R_{cur}, I_{cur})$ in $V$;
        $\Gamma^O \leftarrow$ Set of output ports to which the flit may be routed if it enters $R_{cur}$ by the
          input port $I_{cur}$ ;                                   // use Lemmas 1 and 2
        foreach $O_{cur} \in \Gamma^O$ do
            $\Gamma^I_{next} \leftarrow$ Set of input ports by which the flit may enter $R_{next}$ if it exits $R_{cur}$ by
              the output port $O_{cur}$ ;                          // use Lemmas 3 and 4
            foreach $I_{next} \in \Gamma^I_{next}$ do
                if $I_{next} \notin \Gamma^I_{new}$ then
                    $\Gamma^I_{new} \leftarrow \Gamma^I_{new} \cup \{I_{next}\}$ ;
                    Create vertex $v_{next} = (R_{next}, I_{next})$;
                    $V \leftarrow V \cup \{v_{next}\}$;
                else
                    $v_{next} \leftarrow$ get vertex $(R_{next}, I_{next})$ in $V$;
                end
                Create edge $e = (v_{cur}, v_{next})$ with weight $h^{R_{cur} \to R_{next}}_{O_{cur} \to I_{next}}$ ;   // use Lemma 5
                $E \leftarrow E \cup \{e\}$;
            end
        end
    end
    $\Gamma^I \leftarrow \Gamma^I_{new}$ ;
    $\Gamma^I_{new} \leftarrow \emptyset$ ;
    $R_{cur} \leftarrow R_{next}$;
end
```

▶ **Lemma 2.** *A flit $p$ that enters router $R_{cur}$ by port $I_u$ may be routed to any of the output ports belonging to the set $\Gamma^O_u$, such that*

$$\Gamma^O_u = \begin{cases} \{O_1\} & \text{when } u = D \\ \{O_1\} \cup \{O_{u+1}\} & \text{when } u \neq D \end{cases} \tag{6}$$

**Proof.** According to rule 2 in Table 1, a flit entering by the $I_D$ port has the highest priority to use the $O_1$ port and will never be deflected to any other output port. This proves the first case of Equation (6). If the flit enters the router by an input port $I_u$ such that $u < D$ and requests output port $O_1$, two scenarios may happen according to Table 1: (1) it wins port $O_1$ (see rule 5 in Table 1), or (2) it is deflected to port $O_{u+1}$ (see rule 6 in Table 1). This proves the second case of Equation (6). ◀

▶ **Lemma 3.** *Let $O_u$ be the output port by which flit $p$ exits the router $R_{cur}$. If $R_{next}$ is only one hop further on dimension $\overrightarrow{D_u}$, then flit $p$ enters $R_{next}$ by its port $I_u$.*

**Proof.** Since, according to the network topology defined in Section 4.1, the output port $O_u$ of $R_{cur}$ is connected to the input port $I_u$ of $R_{next}$, and because flit $p$ exits $R_{cur}$ by port $O_u$, the lemma follows. ◀

▶ **Lemma 4.** *Let $O_u$ be the output port by which flit $p$ exits the router $R_{cur}$. If $R_{next}$ is more than one hop away from $R_{cur}$ on dimension $\overrightarrow{D_u}$, then the flit $p$ will enter $R_{next}$ by one of the input ports belonging to the set $\Gamma^I_u$, such that,*

$$\Gamma^I_u = \{I_v \mid u \leq v \leq D\} \tag{7}$$

**Proof.** If $R_{next}$ is more than one hop away from $R_{cur}$ on dimension $\overrightarrow{D_u}$, flit $p$ must hop through at least one other router between $R_{cur}$ and $R_{next}$. First, we note that by definition, $R_{next}$ is the first router after $R_{cur}$ on flit $p$'s path where $p$ may request output port $O_1$. Thus, according to nDimNoC's routing policy (Section 4.3), $p$ may only continue to travel along the same dimension (see rule 3 in Table 1) or be deflected to a higher order dimension while traveling between $R_{cur}$ and $R_{next}$ (see rule 4 in Table 1).

If no deflection happens in the routers located between $R_{cur}$ and $R_{next}$, flit $p$ will enter $R_{next}$ by input port $I_u$. However, according to rule 4 of nDimNoC's routing policy (Table 1), if $u < D$, the flit $p$ may also be deflected to dimension $\overrightarrow{D_{u+1}}$ in one of those intermediate routers. If no further deflection happen until reaching $R_{next}$, $p$ will then enter $R_{next}$ by the input port $I_{u+1}$. Yet, if $u + 1 < D$, Table 1 states that the flit $p$ may still be deflected to dimension $\overrightarrow{D_{u+2}}$ while traveling along dimension $\overrightarrow{D_{u+1}}$. Repeating this reasoning, we get that flit $p$ may enter $R_{next}$ by any input port $I_v$ such that $u \leq v \leq D$. ◀

▶ **Lemma 5.** *The maximum number of hops from the output port $O_u$ of $R_{cur}$ to the input port $I_v$ of $R_{next}$ (with $u \leq v$) is upper bounded by*

$$h^{R_{cur} \to R_{next}}_{O_u \to I_v} = (v - u) + \frac{(pos^{next} - pos^{cur'} + N) \bmod N}{g_{D-v+1}} \tag{8}$$

*where*

$$pos^{cur'} = (pos^{cur} + \sum_{k=u}^{v-1} g_{D-k+1}) \tag{9}$$

*and $pos^{cur}$ and $pos^{next}$ are computed with Equation* (1)*.*

**Proof.** According to nDimNoC's routing policy and following the same explanation as in the proof of Lemma 4, a flit that exits $R_{cur}$ by port $O_u$ and enters $R_{next}$ by port $I_v$ must have been deflected exactly $(v-u)$ times.

According to Property 1, flit $p$ bypasses $g_{D-k+1}$ routers on the main ring of the network with every hop it does on dimension $\overrightarrow{D_k}$. Because, by definition of our circulant topology, we have $g_{D-k+1} > g_{D-k}$ for all $k$, the flit $p$ will make its maximum number of hops when it suffers its $(v-u)$ deflections as early as possible and thus travels as long as possible along the highest order dimension, i.e., along $\overrightarrow{D_v}$.

In such scenario, the flit does exactly one hop on each dimension $\overrightarrow{D_u}$, $\overrightarrow{D_{u+1}}$, $\overrightarrow{D_{u+2}}$, ..., $\overrightarrow{D_{v-1}}$ and bypasses $\sum_{k=u}^{v-1} g_{D-k+1}$ routers on the network's main ring. Thus, after the $(v-u)$ initial hops, the flit reaches router $R_{cur'}$ situated $\sum_{k=u}^{v-1} g_{D-k+1}$ steps further than $R_{cur}$ on the main ring. That is, the position of $R_{cur'}$ on the main ring is given by Equation (9).

Since the network contain $N$ routers on its main ring, the router $R'_{cur}$ and $R_{next}$ are still $(pos^{next} - pos^{cur'} + N) \bmod N$ steps away from each other on that ring. However, since the flit $p$ only travels along dimension $\overrightarrow{D_v}$ after it reached $R'_{cur}$, it bypasses $g_{D-v+1}$ routers of the main ring at each hop. Thus, it needs $\frac{(pos^{next}-pos^{cur'}+N) \bmod N}{g_{D-v+1}}$ hops from port $O_v$ of $R_{cur'}$ to port $I_v$ of $R_{next}$. Therefore, in total, flit $p$ does $(v-u)$ hops to reach $R'_{cur}$ and $\frac{(pos^{next}-pos^{cur'}+N) \bmod N}{g_{D-v+1}}$ additional hops to reach $R_{next}$, hence proving Equation (8). ◀

▶ **Corollary 6.** *If $u = v$, then $h^{R_{cur}\to R_{next}}_{O_u \to I_v}$ is an exact bound on the number of hops between the output port $O_u$ of $R_{cur}$ and the input port $I_v$ of $R_{next}$.*

**Proof.** According to nDimNoC's routing policy, any deflection of a flit $p$ between $R_{cur}$ and $R_{next}$ would result in $p$ entering $R_{next}$ by an input port $I_v$ such that $v > u$. Therefore, if $u = v$, flit $p$ must not have suffered any deflection and must have travel along dimension $\overrightarrow{D_u}$ only. Because $(pos^{next} - pos^{cur} + N) \bmod N$ is the distance between $R_{cur}$ and $R_{next}$ on the main ring of the network, and because for every hop on dimension $\overrightarrow{D_u}$, packet $p$ bypasses $g_{D-u+1}$ routers on the main ring (by Property 1), we have that $p$ reaches $R_{next}$ in exactly $\frac{(pos^{next}-pos^{cur}+N) \bmod N}{g_{D-v+1}}$ hops. Note that this last equation is equal to $h^{R_{cur}\to R_{next}}_{O_u \to I_v}$ when $v = u$, which proves the claim. ◀

We now prove that the WCTT and BCTT of a flit of flow $f_j$ are bounded by the longest and the shortest weighted path of the graph $G$ returned by Algorithm 1, respectively. To do so, we first prove that the graph $G$ built using Algorithm 1 contains all routes that may be taken by packets of flow $f_j$ between its origin and destination.

▶ **Lemma 7.** *The DAG built using Algorithm 1 contains one edge for each possible path between any two routers in $\mathcal{R}$ that may be successively traversed by any flit of flow $f_j$.*

**Proof.** Algorithm 1 iterates over all routers in $\mathcal{R}$ that are on the path of $f_j$ from its origin to its destination router (Lines 3, 9, 10 and 31). For each pair of routers $R_{cur}$, $R_{next}$, the algorithm computes the set $\Gamma^I$ of all input ports by which $f_j$ may enter $R_{cur}$. For each such input, it uses Lemmas 1 and 2 to compute the set $\Gamma^O$ of all output ports by which $f_j$ may exit $R_{cur}$ (Line 13). For each output port $O_{cur} \in \Gamma^O$, it then uses Lemmas 3 and 4 to compute the set $\Gamma^I_{next}$ of all input ports by which $f_j$ may enter $R_{next}$ (Line 15). It finally creates an edge for every path between $O_{cur}$ and the input ports in $\Gamma^I_{next}$ (Line 24). Since Lemmas 1 to 4 were all proven correct, we have that Algorithm 1 creates an edge for every possible path between any two routers $R_{cur}$ and $R_{next}$ in $\mathcal{R}$, i.e., one edge for any combination of output and input port of $R_{cur}$ and $R_{next}$ that may be successively traversed by a packet of $f_j$. ◀

Lemma 7 has the following corollary as direct consequence.

▶ **Corollary 8.** *The DAG built using Algorithm 1 contains all possible paths taken by flow $f_j$ from its origin to its destination router.*

▶ **Theorem 9.** *The longest weighted path of the DAG built with Algorithm 1 is an upper bound on the WCTT of any flit of flow $f_j$.*

**Proof.** By Lemma 7 and Corollary 8, the DAG built with Algorithm 1 contains all possible routes from the origin to the destination of $f_j$ encoded as a different path in the graph. Furthermore, by Lemma 5 and Line 24 of Algorithm 1, the weight of every edge in the graph is an upper bound on the number of hops on the longest path between the output and input ports of the two routers represented by the vertices connected by that edge. Thus, the longest weighted path in the graph is an upper bound on the number of hops between all routers on the path of $f_j$ from its origin to its destination. This proves the Theorem. ◀

▶ **Theorem 10.** *The shortest weighted path of the DAG built with Algorithm 1 is the BCTT of any flit of flow $f_j$.*

**Proof.** According to nDimNoC's routing policy and its discussion in Section 4.3, a flit $p$ of flow $f_j$ performs its minimum number of hops between its origin and destination router when it does not suffer any deflection.

Since the DAG built with Algorithm 1 contains all possible routes from the origin to the destination of $f_j$ encoded as a different path in the graph (by Lemma 7 and Corollary 8), it also contains the path where the flit of $f_j$ does not suffer any deflection. Furthermore, by Corollary 6 and Line 24 of Algorithm 1, the weight of every edge corresponding to a path where $p$ does not suffer deflection is equal to the exact number of hops performed by $p$ on that path. Therefore, the shortest weighted path in the graph is an exact bound on the BCTT of $f_j$ from its origin to its destination. This proves the Theorem. ◀

## 5.2 Worst-case injection time

In the previous section, we explained how to compute bounds on the BCTT and WCTT of any flit of a packet injected by a flow $f_j$. In this section, we derive a bound on the worst-case injection time WCIT of any packet of $f_j$ (see Theorem 12).

First, we recall a bound on the maximum number of packets that may be injected in the network by any flow $f_j$. This bound was already proven in [31].

▶ **Lemma 11.** *In any time interval of length $\Delta t$, the flow $f_j$ can inject in the network at most $\lambda_j(\Delta t) = \min\left\{\Delta t,\ \left\lceil \frac{\Delta t + wcit_j}{T_j} \right\rceil C_j\right\}$ flits.*

**Proof.** The proof is similar to that of the maximum workload that can be executed by a task with minimum inter-arrival time $T_j$ and release jitter $wcit_j$. The complete proof is provided in Lemma 14 of [31]. ◀

Then, we prove an upper bound on the WCIT of any packet of $f_j$ using Theorem 12. To prove that theorem, we use the following notation: flow $f_j$ is injected in router $R_{inj}$ via input port $I_{inj}^{PE}$ (i.e., $R_{inj}$ is the origin router of $f_j$). We define $\mathcal{F}_{inj}$ as the set of all flows (including $f_j$) injected in the same input port $I_{inj}^{PE}$ of the same router $R_{inj}$ as $f_j$. Note that this set of flows is a property of the system and thus we assume that it is given as an input to the analysis. We also define $\Gamma_{inj}^{conf}$ as the set of all flows originating from other routers than $R_{inj}$ and that may conflict with the injection of flow $f_j$ in router $R_{inj}$. The content of $\Gamma_{inj}^{conf}$ is computed using Lemmas 13 and 17 proven later in this section.

▶ **Theorem 12.** *The WCIT $wcit_j$ of any packet of flow $f_j$ is given by the smallest positive solution to the recursive equation*

$$wcit_j \geq \left( \sum_{\forall f_i \in \mathcal{F}_{inj}} C_i \right) - 2 + \sum_{\forall f_l \in \Gamma_{inj}^{conf}} \lambda_l(wcit_j + 1 + J_l) \tag{10}$$

*where $J_l = wctt'_l - bctt'_l$ is the difference between the worst-case and the best-case traversal time of flow $f_l$ until router $R_{inj}$ (computed with Theorems 9 and 10).*

**Proof.** Let $p$ be the last flit of any packet of flow $f_j$. According to nDimNoC's routing policy, the flit $p$ will be injected in the network as soon as : 1) all flits ahead of $p$ in the FIFO queue of the input port by which it is injected in the network have been injected, and 2) there is one clock cycle during which no packet entering $R_{inj}$ from other input ports conflicts for the same output port as $p$.

Let $n_{flits}$ be the maximum number of flits ahead of $p$ in the FIFO queue, and let $W_u(\Delta t)$ be the maximum number of flits entering into $R_{inj}$ by another input port than $p$ and requesting the same output port as $p$ in a time interval of length $\Delta t$. Then, conditions 1) and 2) are met as soon as

$$\Delta t + 1 \geq n_{flits} + W_u(\Delta t + 1) \tag{11}$$

for $\Delta t \geq 0$. The solution to Equation (11) is thus equal to the WCIT $wcit_j$ of flow $f_j$.

To solve the above equation, we first derive a bound on $n_{flits}$, and then derive a bound on $W_u(\Delta t + 1)$.

**Bound on $\boldsymbol{n_{flits}}$:** Section 4.2 explains that each flow in $\mathcal{F}_{inj}$ may have at most one packet in the FIFO queue of the input port by which they are injected in the network. Therefore, in the worst-case scenario, there is one packet of each other flow injected via the same port $I_{inj}^{PE}$ as $f_j$ ahead of $p$ in the FIFO queue. Since $p$ is the last flit of $f_j$'s packet, there must also be $(C_j - 1)$ other flits of $f_j$ ahead of $p$ in the FIFO queue. That is,

$$n_{flits} \leq \left( \sum_{\forall f_i \in \mathcal{F}_{inj}} C_i \right) - 1 \tag{12}$$

**Bound on $\boldsymbol{W_u(\Delta t + 1)}$:** Let $f_l$ be a flow entering the router $R_{inj}$ by another input port than $p$ but requesting the same output port as $p$ (i.e., $f_l \in \Gamma_{inj}^C$). In the best and worst case scenarios, a flit from $f_l$ takes $bctt'_l$ and $wctt'_l$ clock cycles to reach $R_{inj}$, respectively. Therefore, the first flit of $f_l$ that may conflict with the injection of $p$ must have been injected no earlier than $wctt'_l$ clock cycles before the beginning of the period during which $p$ is interfered with. Conversely, the last flit of $f_l$ that may conflict with the injection of $p$ must have been injected no later than $bctt'_l$ clock cycles before the end of the interference with $p$. Therefore, the length of the interval during which $f_l$ may inject flits that conflict with the injection of $p$ is

$$\Delta t + 1 + wctt'_l - bctt'_l = \Delta t + 1 + J_l. \tag{13}$$

Lemma 11 states that a flow $f_l$ may inject at most $\lambda_l(\Delta t + 1 + J_l)$ flits in that time interval. Therefore, the total number of flits from all conflicting flows $f_l \in \Gamma_{inj}^C$ is upper bounded as

$$W_u(\Delta t + 1) \leq \sum_{\forall f_l \in \Gamma_{inj}^C} \lambda_l(\Delta t + 1 + J_l) \tag{14}$$

Injecting Equations (12) and (14) in Equation (11), we prove the lemma. ◀

Theorem 12 requires to know the set of all flows $\Gamma_{inj}^{C}$ that may conflict with the injection of $f_j$ in router $R_{inj}$. The content of that set depends on the specific output port requested by the packets of $f_j$. Lemmas 13 and 17 below provide a means to compute the content of $\Gamma_{inj}^{C}$ when $f_j$ request output port $O_1$ or $O_u$ (with $u \neq 1$), respectively.

▶ **Lemma 13.** *The set of flows coming from other routers than $R_{inj}$ and that may be routed to output port $O_1$ of $R_{inj}$ is given by*

$$\Gamma_{inj}^{1} = \{f_l \mid \forall b \in [2, D],\ d_b^l = r_b^{inj} \wedge \mathrm{dist}(R_{orig}(f_l), R_{inj}) \leq \mathrm{dist}(R_{orig}(f_l), R_{dest}(f_l))\} \quad (15)$$

**Proof.** According to nDimNoC's routing policy, a flow $f_l$ may request the output port $O_1$ of $R_{inj}$ only if (i) $f_l$ may hop through router $R_{inj}$, and (ii) $\forall b > 1,\ d_b^l = r_b^{inj}$. Condition (i) requires that $R_{inj}$ is located between the origin and destination router of $f_l$, i.e., $\mathrm{dist}(R_{orig}(f_l), R_{inj}) \leq \mathrm{dist}(R_{orig}(f_l), R_{dest}(f_l))$. Combining both (i) and (ii) proves the lemma. ◀

To compute the set $\Gamma_{inj}^{C}$ for the case where $f_j$ requests output port $O_u$ (with $u \neq 1$), we must first prove some intermediate results using Lemmas 14 to 16.

To prove those lemmas, we define $\mathcal{F}_{inj}^{u}$ as the set of all flows that may enter router $R_{inj}$ by input port $I_u$. That set can easily be built by checking all paths that may be taken by each flow in the network according to Table 1. All those that have at least one path in which they enter $R_{inj}$ by input port $I_u$ are then added to the set $\mathcal{F}_{inj}^{u}$.

▶ **Lemma 14.** *The set of flows that may enter $R_{inj}$ by input port $I_u$ and request output port $O_u$ is given by $\Gamma_{inj}^{u \to u} = \mathcal{F}_{inj}^{u} \setminus \{f_l \mid \forall b \in [2, D], r_b^{inj} = d_b^l\}$*

**Proof.** According to Table 1, a flow entering by input port $I_u$ and requesting output port $O_1$ cannot be routed to output port $O_u$ (only to $O_1$ or $O_{u+1}$) (see rules 2, 5, and 6 in Table 1). Therefore, the set of flows entering by $I_u$ that may be routed to $O_u$ is the set of flows that enters $R_{inj}$ by $I_u$ (i.e., $\mathcal{F}_{inj}^{u}$) minus those that request $O_1$, i.e., all the flows $f_l$ that have a destination such that $\forall b \in [2, D], r_b^{inj} = d_b^l$ (see routing policy explained in Section 4.3). This proves the lemma. ◀

▶ **Lemma 15.** *Let $\mathrm{def}_q$ be a boolean equal to true if a deflection may happen in router $R_q$, and equal to false otherwise. Then, we have*

$$\mathrm{def}_q = \begin{cases} \textbf{\textit{true}} & \textit{if } \exists u, v \textit{ with } u \neq v \mid \exists f_l \in \mathcal{F}_q^u, \exists f_m \in \mathcal{F}_q^v \textit{ s.t. } l \neq m \wedge \forall b \in [2, D],\ d_b^l = d_b^m = r_b^q \\ \textbf{\textit{false}} & \textit{otherwise.} \end{cases} \quad (16)$$

**Proof.** According to Table 1, a deflection may happen in router $R_q$ only if at least two different flows compete to access the output port $O_1$. For that situation to happen, there must exist at least two different flows $f_l$ and $f_m$ entering by two different input ports (i.e., $\exists u, v$ with $u \neq v \mid \exists f_l \in \mathcal{F}_q^u, \exists f_m \in \mathcal{F}_q^v$ s.t. $l \neq m$) that both request output port $O_1$. According to the routing policy explained in Section 4.3, this happens only if $\forall b \in [2, D],\ d_b^l = d_b^m = r_b^q$. This proves the lemma. ◀

▶ **Lemma 16.** *The set of flows that may enter $R_{inj}$ by input port $I_{u-1}$ and be routed to output port $O_u$ is given by*

$$\Gamma_{inj}^{u-1 \to u} = \begin{cases} \mathcal{F}_{inj}^{u-1} & \textit{if } \mathrm{def}_{inj} = \textbf{\textit{true}} \\ \emptyset & \textit{if } \mathrm{def}_{inj} = \textbf{\textit{false}} \end{cases} \quad (17)$$

**Table 2** Resources utilization of different NoCs in Kirtex-7 FPGAs.

| NoC | LUTs | % Resource utilization of the platform |
|---|---|---|
| 8x8 ProNoC | 100000 | 20%-150% |
| 8x8 IDAMC | 83000 | 18%-127% |
| 8x8 CONNECT | 96000 | 20%-147% |
| 8x8 HopliteRT* | 5632 | 1.1%-8.5% |
| 4x4x4 3D-nDimNoC | 18560 | 3.9%-28% |

**Proof.** According to Table 1, all flows that enter router $R_{inj}$ by input port $I_{u-1}$ (i.e., those in $\mathcal{F}_{inj}^{u-1}$) may be deflected to output port $O_u$ (see rules 4 and 6 in Table 1). Thus, the set of flows entering by $I_{u-1}$ and routed to $O_u$ is given by all flows in $\mathcal{F}_{inj}^{u-1}$ if a deflection may happen in $R_{inj}$, i.e., if $\text{def}_{inj} = \textbf{true}$. If no deflection may happen in $R_{inj}$ (i.e., $\text{def}_{inj} = \textbf{false}$), then Table 1 states that none of the flows entering by $I_{u-1}$ may be routed to $O_u$. This proves both cases of Equation (17). ◀

▶ **Lemma 17.** *The set of flows coming from other routers than $R_{inj}$ and that may be routed to output port $O_u$ of $R_{inj}$ (with $u \neq 1$) is given by*

$$\Gamma_{inj}^{u} = \Gamma_{inj}^{u \to u} \cup \Gamma_{inj}^{u-1 \to u} \tag{18}$$

**Proof.** According to Table 1, only flows that enter a router by its input ports $I_u$ or $I_{u-1}$ can be routed to output port $O_u$. Since, according to Lemmas 14 and 16, $\Gamma_{inj}^{u \to u}$ and $\Gamma_{inj}^{u-1 \to u}$ contain all the flows entering $R_{inj}$ by input ports $I_u$ and $I_{u-1}$, respectively, that may be routed to output port $O_u$, their union contains all flows that may come from other routers than $R_{inj}$ and may be routed to output port $O_u$ of $R_{inj}$. ◀

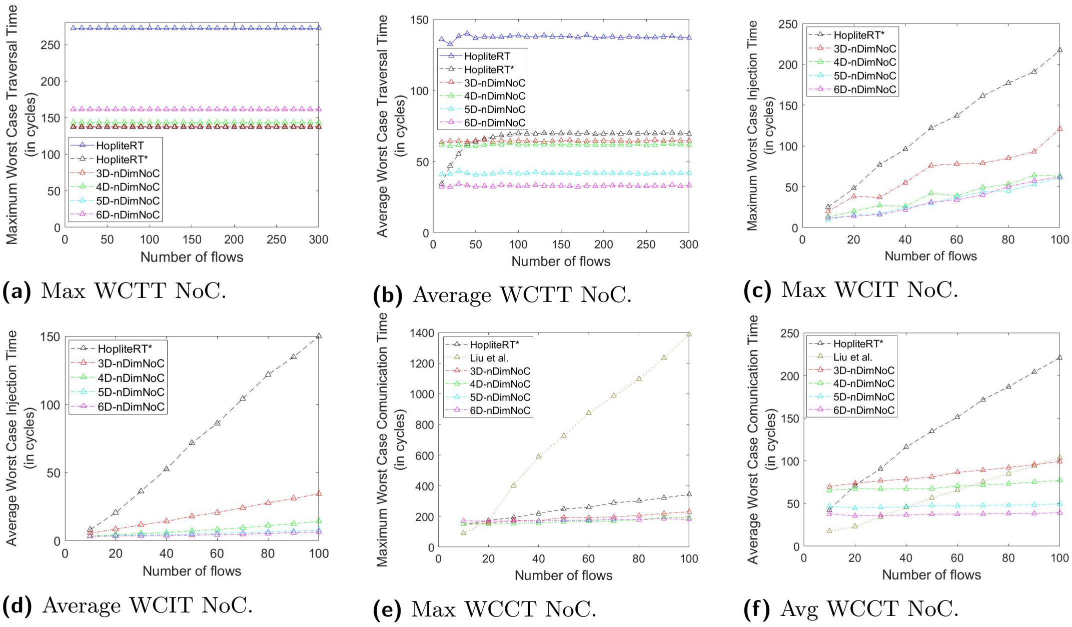
The content of $\Gamma_{inj}^{C}$ is thus equal to $\Gamma_{inj}^{1}$ if $f_j$ requests output port $O_1$ (Lemma 13), and to $\Gamma_{inj}^{u}$ if it requests any other output port (Lemma 17). Using $\Gamma_{inj}^{C}$ in Theorem 12 we can now bound the WCIT of $f_j$.

# 6 Experimental results

## 6.1 Implementation of nDimNoC

We implemented a 3D-nDimNoC with the hardware description language Verilog. We synthesized a single router of 3D-nDimNoC for flits of 64 bits. The target platform was a Xilinx Virtex-7 485T FPGA. It required 290 LUTs and 202 Flip-Flops (FFs) in total. This corresponds to only 0.1% and 0.03% of the total number of LUTs and FFs available in the target FPGA, respectively.

We compared the hardware cost of a 3D-nDimNoC with HopliteRT* [31], as well as to some other NoCs based on virtual channels (VCs): ProNoC [23], IDAMC [37], and CONNECT [27]. The target platform was a Xilinx Kirtex-7 FPGA. Kirtex-7 is a mid-range family of FPGAs that contains approximately between 65,600 and 477,760 LUTs depending on which one you pick. Table 2 shows the synthesis results. A ProNoC router with two VCs required 1574 LUTs, a HopliteRT* router required 88 LUTs, and according to [38] and [27], an IDAMC and a CONNECT router require approximately 1300 and 1500 LUTs,



**(a)** Max WCTT NoC.

**(b)** Average WCTT NoC.

**(c)** Max WCIT NoC.

**(d)** Average WCIT NoC.

**(e)** Max WCCT NoC.

**(f)** Avg WCCT NoC.

**Figure 4** Experimental results for a random traffic pattern.

respectively. Then, as reported in Table 2, an 8x8 ProNoC, IDAMC, and CONNECT NoCs require ≈100,000, ≈83,000, and ≈96,000 LUTs, respectively, eating up a big portion (if not all in some cases) of the logic available in the FPGA. This leaves limited resources available for any computation logic. Therefore, those solutions are not really suitable for systems implemented over mid-range FPGAs. On the other hand, a 4x4x4 3D-nDimNoC requires 18,560 LUTs, i.e., between 3.9% to 28% of the Xilinx Kirtex-7 resources. It is three times more expensive than HopliteRT* (which requires 5632 LUTs) but approximately 5-times cheaper than ProNoC, IDAMC, and CONNECT NoCs in terms of LUTs utilization. We thus conclude that 3D-nDimNoC is a suitable solution for such FPGA platforms.

Finally, we connected the nDimNoC router to a Microblaze soft-core and synthesized a 4x2x2 3D-network for a Virtex-7 485T using Xilinx Vivado. We computed the maximum operating frequency of the network with Xilinx Vivado. We obtained ≈210 MHz for a 4x2x2 3D-nDimNoC against ≈275 MHz for an equivalent 4x4 HopliteRT* NoC. This degradation in terms of maximum operating frequency may be explained by the fact that (1) an nDimNoC router requires more complex logic to route packets from its input to its output ports, and (2) the additinal dimensions increase the number of wires between routers, which increases the complexity of the placement and routing during the logic synthesis.

## 6.2 Analyses results

In this section, we provide experimental results by computing the WCTT, WCIT, and WCCT of sets of communication flows that traverse NoCs of different dimensionalities.

As a starting point, we generated sets of communication flows for a 16x16 2D-NoC according to a random traffic pattern. The origin and destination coordinates of each flow were randomly generated using a uniform probability distribution. The number of flits of packets released by a communication flow was randomly chosen between 1 and 5, and their inter-arrival times were generated as in [36]. Then, we made a one-to-one mapping of the routers in the 16x16 2D-NoC to the routers of a 4x8x8 3D-nDimNoC, a 4x4x4x4 4D-nDimNoC, a 2x2x4x4x4 5D-nDimNoC, and a 2x2x2x2x4x4 6D-nDimNoC. The origin and destination of each flow were accordingly updated for each network topology.

In Figs. 4a and 4b, we show the maximum and average packets WCTT for an increasing number of flows in NoCs of different dimensionalities. The results were computed by using the analysis of HopliteRT [39, 40] and HopliteRT* [31] (assuming a 16x16 2D-NoC), and the analysis presented in Section 5.1 for the 2D, 3D, 4D, 5D and 6D-nDimNoC topology. To establish a fair comparison, we assume one priority level (i.e., all flows were assigned the highest priority) for the analysis proposed in [31]. Each point in the plot is the result of 100 repetitions (100 different random flow sets). We varied the number of generated flows from 10 to 300 by steps of 10.

In 4a, we observe that the maximum WCTT is slightly worse with nDimNoC as compared to HopliteRT*. Nonetheless, Fig 4b) shows that the average traversal time improves with nDimNoC as the dimensionality of the network increases. This can easily be explained by the fact that new routes, possibly shorter and faster, are made available between pairs of routers when a new dimension is added to the network. Moreover, the number of interfering flows, and therefore, the number of deflections that flows may suffer on each link decreases since the number of routers on each dimension decreases. Note that, the average packets WCTT is reduced by ≈40% and ≈60% with a 5D-nDimNoC and a 6D-nDimNoC, respectively, against HopliteRT*. We also show that the maximum and average worst-case traversal times are noticeably reduced with nDimNoC as compared to HopliteRT.

In Fig. 4c and 4d, we show the maximum and average WCIT of flows using the analysis of HopliteRT* and nDimNoC. We also computed the maximum and average WCIT by using the analysis of HopliteRT, but we do not show them on the graphs as they are extremely pessimistic and would render the plots unreadable by cluttering all other lines together. As shown, the packets see their WCIT drastically reduced in nDimNoC in comparison to HopliteRT*. This is expected since nDimNoC allows the programming element connected to a router to inject packets simultaneously via as many input ports as there are dimensions in the network. A router of HopliteRT*, on the other hand, can inject at most one flit per cycle in the network (on either of the router output ports). Furthermore, the number of communication flows that may interfere with the injection of a packet at a router decreases since more routes are available in the network, and thus less traffic uses each individual route.

In Fig. 4e and 4f, we show the maximum and average packets WCCT (which we recall to be equal to the sum of the WCTT and WCIT of those packets). We varied the number of generated flows from 10 to 100 by steps of 10. The results were obtained by using the analysis of nDimNoC, and the analyses proposed in [31] and [21] for HopliteRT* and a VC-based real-time NoC, respectively. The analysis presented in [21] by Liu et al. is an improved analysis of that proposed in [36, 35] by Shi and Burns. To establish a fair comparison, we assume one VC (i.e., one priority level) for the analysis presented in [21]. As shown in Figure 4e, for almost all configurations, the WCCT returned by the analysis of nDimNoC outperforms that returned by the analysis of [31] and [21]. The average WCCT is only better with the analysis by Liu et al. when the network is completely underloaded and very few flows are traversing the network (i.e., less than 30 flows). Note also that, [21] considers that each flow may only have one packet of each flow traveling across the network at the same time, whilst nDimNoC supports the transmission of several packets from the same communication flow simultaneously.

The average WCCT with nDimNoC improves when the network's dimensionality increases and is barely impacted by the number of flows. Therefore, we conclude that increasing the dimensionality of nDimNoC has a positive impact from an average performance perspective for a limited impact on the worst-case performance of the flows.

## 7 Summary and conclusion

In this paper, we presented nDimNoC, a new and flexible real-time D-dimensional NoC that uses the properties of circulant topologies to provide real-time guarantees to the flows transmitted over that NoC. We proposed a timing analysis for nDimNoC. We also did a complete implementation of 3D-nDimNoC in HDL Verilog. Experimental results show improvements in terms of network communication latency in comparison to existing 2D solutions.

## References

1 P. Baran. On distributed communications networks. *IEEE Transactions on Communications Systems*, 12(1):1–9, 1964. `doi:10.1109/TCOM.1964.1088883`.

2 L. Benini and G. De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 418–419, March 2002.

3 Alan Burns, James Harbin, and Leandro Soares Indrusiak. A wormhole NoC protocol for mixed criticality systems. In *IEEE Real-Time Systems Symposium*, pages 184–195, 2014.

4 Yiou Chen, Jianhao Hu, Xiang Ling, and Tingting Huang. A novel 3d noc architecture based on de bruijn graph. *Computers & Electrical Engineering*, 38(3):801–810, 2012.

5 Shamik Das, Andy Fan, Kuan-Neng Chen, Chuan Seng Tan, Nisha Checka, and Rafael Reif. Technology, performance, and computer-aided design of three-dimensional integrated circuits. In *Proceedings of the 2004 International Symposium on Physical Design*, ISPD '04, page 108?115, New York, NY, USA, 2004. Association for Computing Machinery. `doi:10.1145/981066.981091`.

6 Dakshina Dasari, Borislav Nikoli'c, Vincent N'elis, and Stefan M Petters. NoC contention analysis using a branch-and-prune algorithm. *ACM Transactions on Embedded Computing Systems*, 13(3s):113, 2014.

7 Jonas Diemer, Jonas Rox, Mircea Negrean, Steffen Stein, and Rolf Ernst. Real-time communication analysis for networks with two-stage arbitration. In *9th ACM International Conference on Embedded Software*. IEEE, 2011.

8 Feihui Li, C. Nicopoulos, T. Richardson, Yuan Xie, V. Narayanan, and M. Kandemir. Design and management of 3d chip multiprocessors using network-in-memory. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 130–141, 2006. `doi:10.1109/ISCA.2006.18`.

9 Yan Ghidini, Thais Webber, Edson Moreno, Fernando Grando, Rubem Fagundes, and César Marcon. Buffer depth and traffic influence on 3d nocs performance. In *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 9–15. IEEE, 2012.

10 Frédéric Giroudot and Ahlem Mifdaoui. Buffer-aware worst-case timing analysis of wormhole NoCs using network calculus. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018.

11 Frederic Giroudot and Ahlem Mifdaoui. Tightness and computation assessment of worst-case delay bounds in wormhole networks-on-chip. In *27th International Conference on Real-Time Networks and Systems*, 2019.

12 C. Grecu, P. P. Pande, A. Ivanov, and R. Saleh. A scalable communication-centric soc interconnect architecture. In *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*, pages 343–348, 2004. `doi:10.1109/ISQED.2004.1283698`.

13 R. I. Greenberg and Lee Guan. An improved analytical model for wormhole routed networks with application to butterfly fat-trees. In *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No.97TB100162)*, pages 44–48, 1997. `doi:10.1109/ICPP.1997.622554`.

14 P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pages 250–256, 2000. doi:10.1109/DATE.2000.840047.

15 Jörg Henkel, Wayne Wolf, and Srimat Chakradhar. On-chip networks: A scalable, communication-centric embedded system design paradigm. In *17th International Conference on VLSI Design*. IEEE, 2004.

16 S. Hesham, J. Rettkowski, D. Goehringer, and M. A. Abd El Ghany. Survey on real-time networks-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1500–1517, May 2017. doi:10.1109/TPDS.2016.2623619.

17 Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolić. Buffer-aware bounds to multi-point progressive blocking in priority-preemptive nocs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 219–224. IEEE, 2018.

18 Leandro Soares Indrusiak, James Harbin, and Alan Burns. Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip. In *27th Euromicro Conference on Real-Time Systems*. IEEE, 2015.

19 J. W. Joyner, P. Zarkesh-Ha, and J. D. Meindl. A stochastic global net-length distribution for a three-dimensional system-on-a-chip (3d-soc). In *Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat. No.01TH8558)*, pages 147–151, 2001. doi:10.1109/ASIC.2001.954688.

20 C. C. Liu, I. Ganusov, M. Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3d ic technology. *IEEE Design Test of Computers*, 22(6):556–564, 2005. doi:10.1109/MDT.2005.134.

21 Meng Liu, Matthias Becker, Moris Behnam, and Thomas Nolte. Tighter time analysis for real-time traffic in on-chip networks with shared priorities. In *10th IEEE/ACM International Symposium on Networks-on-Chip*, 2016.

22 César Marcon, Ramon Fernandes, Rodrigo Cataldo, Fernando Grando, Thais Webber, Ana Benso, and Letícia B Poehls. Tiny noc: A 3d mesh topology with router channel optimization for area and latency minimization. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 228–233. IEEE, 2014.

23 Alireza Monemi, Jia Tang, Maurizio Palesi, and Muhammad Nadzir Marsono. ProNoC: A low latency network-on-chip based many-core system-on-chip prototyping platform. *Microprocessors and Microsystems*, 54, September 2017. doi:10.1016/j.micpro.2017.08.007.

24 B. Nikolic, Robin Hofmann, and R. Ernst. Slot-based transmission protocol for real-time nocs - sbt-noc. In *ECRTS*, 2019.

25 B. Nikolić and S. M. Petters. Edf as an arbitration policy for wormhole-switched priority-preemptive nocs-myth or fact? In *International Conference on Embedded Software*, pages 1–10, October 2014.

26 Borislav Nikolić, Sebastian Tobuschat, Leandro Soares Indrusiak, Rolf Ernst, and Alan Burns. Real-time analysis of priority-preemptive nocs with arbitrary buffer sizes and router delays. *Real-Time Systems*, 55(1):63–105, 2019.

27 M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining conventional wisdom for designing Nocs in the context of FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, New York, NY, USA, 2012. ACM.

28 D. Park, S. Eachempati, R. Das, A. K. Mishra, Y. Xie, N. Vijaykrishnan, and C. R. Das. Mira: A multi-layered on-chip interconnect router architecture. In *2008 International Symposium on Computer Architecture*, pages 251–261, 2008. doi:10.1109/ISCA.2008.13.

29 Vasilis F Pavlidis, Ioannis Savidis, and Eby G Friedman. *Three-dimensional integrated circuit design*. Newnes, 2017.

30 Eberle A Rambo and Rolf Ernst. Worst-case communication time analysis of networks-on-chip with shared virtual channels. In *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

31 Y. Ribot González and G. Nelissen. Hoplitert*: Real-time noc for fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3650–3661, 2020. doi:10.1109/TCAD.2020.3012748.

32 Aleksandr Yu Romanov. Development of routing algorithms in networks-on-chip based on ring circulant topologies. *Heliyon*, 5(4):e01516, 2019.

33 Abbas Sheibanyrad, Frédéric Pétrot, Axel Jantsch, et al. *3D integration for NoC-based SoC Architectures*. Springer, 2011.

34 Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Second ACM/IEEE International Symposium on Networks-on-Chip*, 2008.

35 Zheng Shi and Alan Burns. Improvement of schedulability analysis with a priority share policy in on-chip networks. In *17th International Conference on Real-Time and Network Systems*, pages 75–84, 2009.

36 Zheng Shi and Alan Burns. Real-time communication analysis with a priority share policy in on-chip networks. In *21st Euromicro Conference on Real-Time Systems*, pages 3–12. IEEE, 2009.

37 S. Tobuschat, P. Axer, R. Ernst, and J. Diemer. IDAMC: A NoC for mixed criticality systems. In *IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2013. doi:10.1109/RTCSA.2013.6732214.

38 Sebestian Tobuschat. *Predictable and Runtime-Adaptable Network-On-Chip for Mixed-critical Real-time Systems*. PhD thesis, TU Braunschweig, 2019.

39 S. Wasly, R. Pellizzoni, and N. Kapre. HopliteRT: An efficient FPGA NoC for real-time applications. In *International Conference on Field Programmable Technology*, pages 64–71, December 2017.

40 Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. Worst case latency analysis for hoplite FPGA-based NoC. Technical report, University of Waterloo, 2017.

41 Qin Xiong, Zhonghai Lu, Fei Wu, and Changsheng Xie. Real-time analysis for wormhole noc: Revisited and revised. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*, pages 75–80, 2016.

42 Qin Xiong, Fei Wu, Zhonghai Lu, and Changsheng Xie. Extending real-time analysis for wormhole nocs. *IEEE Transactions on Computers*, 66(9):1532–1546, 2017.

# Light Reading: Optimizing Reader/Writer Locking for Read-Dominant Real-Time Workloads

Catherine E. Nemitz
University of North Carolina at Chapel Hill, NC, USA

Shai Caspin
University of North Carolina at Chapel Hill, NC, USA

James H. Anderson
University of North Carolina at Chapel Hill, NC, USA

Bryan C. Ward
MIT Lincoln Laboratory, Lexington, MA, USA

## Abstract

This paper is directed at reader/writer locking for read-dominant real-time workloads. It is shown that state-of-the-art real-time reader/writer locking protocols are subject to performance limitations when reads dominate, and that existing schedulability analysis fails to leverage the sparsity of writes in this case. A new reader/writer locking-protocol implementation and new inflation-free schedulability analysis are proposed to address these problems. Overhead evaluations of the new implementation show a decrease in overheads of up to 70% over previous implementations, leading to throughput for read operations increasing by up to 450%. Schedulability experiments are presented that show that the analysis results in schedulability improvements of up to 156.8% compared to the existing state-of-the-art approach.

**2012 ACM Subject Classification** Computer systems organization → Real-time system architecture; Computing methodologies → Shared memory algorithms

**Keywords and phrases** Reader/writer, real-time, synchronization, spinlock, RMR complexity

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.6

**Supplementary Material** *Software (ECRTS 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.1.3`

**Funding** Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698. 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. This work was also supported by a Dissertation Completion Fellowship from the UNC Graduate School.

**Acknowledgements** DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering. © 2021 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 6; pp. 6:1–6:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

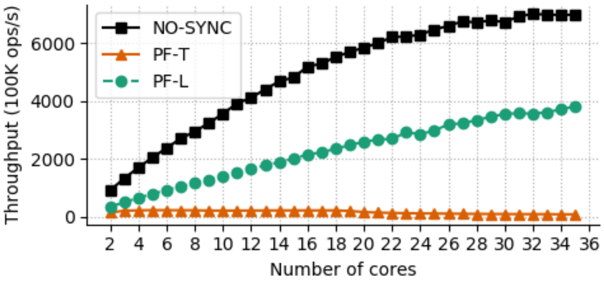


**Figure 1** Throughput for red-black tree lookups. This read-only scenario is representative of use cases where writes can occur but are so infrequent that over long intervals, only reads occur.

## 1 Introduction

In an ongoing project, our research group has been investigating real-time use cases where shared resources exist that are much more frequently read than written. The importance of such read-dominant use cases has been well documented by McKenney [24], who devised a non-blocking synchronization solution called *Read-Copy-Update (RCU)* to support resource sharing. As explained later, RCU can be problematic in real-time systems, so for our use cases, a better option is *reader/writer locks*, a now-standard synchronization solution first proposed five decades ago [16]. A reader/writer lock extends a mutex lock by distinguishing between *read accesses* (non-modifying) and *write accesses* (potentially modifying) and seeks to allow reads to execute concurrently with one another while supporting exclusive writing.

At the outset of our project, it was our belief that reader/writer locking was a solved problem for real-time systems. This belief was rooted in the existence of reader/writer locking protocols that are asymptotically optimal with respect to blocking times [12], and the apparent ease with which such blocking times can be factored into schedulability analysis [3]. In delving further, however, we found this belief to be wrong. In particular, for read-dominant use cases, we found major deficiencies with respect to both state-of-the-art real-time reader/writer locking protocols and the schedulability analysis needed to apply them.

These experiences motivated this paper, which is directed at the goal of efficiently supporting read-dominant real-time workloads. Our contributions towards this goal include a new reader/writer locking-protocol implementation and schedulability analysis. We expound on these contributions below, after first elaborating on the deficiencies noted above.

**Surprising performance limitations in existing reader/writer locks.** In the real-time literature, Brandenburg and Anderson presented a category of reader/writer locks called *phase-fair locks* over a deacade ago [12] and established the optimality of such locks under common definitions of *priority-inversion blocking (pi-blocking)*. To our knowledge, phase-fair locks stand as the state-of-the-art for spin-based (our focus) real-time reader/writer locking.

In our work on read-dominant workloads, we employed a *phase-fair ticket lock (PF-T)*, one of Brandenburg and Anderson's proposed phase-fair variants [11]. In experiments involving PF-Ts, we observed perplexing behavior, shown in Fig. 1: throughput did not scale beyond four cores for a purely read-only workload. This was surprising as the phase-fair lock logic should allow all reads to execute without ever blocking. So why then did the PF-T not scale to match the case of having no synchronization at all (NO-SYNC)?

The answer relates to the overhead of the PF-T's lock/unlock logic. In particular, every lock and unlock call updates the shared lock state. Even under a read-only workload, these updates invalidate cached lock state on other cores. Contention for this shared lock state incurs significant overhead that severely hampers throughput. Additionally, many shared-state updates require the use of an atomic instruction.

**Analytically leveraging the sparsity of writes.** In examining existing schedulability analysis for phase-fair locks [11], we found that they do not exploit the sparsity of writes in read-mostly workloads. In particular, reads are pessimistically assumed to always incur some write blocking. Thus, even if lock performance close to NO-SYNC could be achieved, such improvements would be lost *analytically* in checking schedulability. In recent years, holistic, *inflation-free* blocking analysis has been developed for mutex locks that limits over-estimates of blocking. However, such analysis has not been extended to apply to reader/writer locks.

**Contributions.** The contributions of this paper are four-fold. First, in Sec. 3, we present the spin-based *phase-fair with light reading ticket lock (PF-L)*, which is optimized for read-dominant workloads. The PF-L achieves low read overhead by eliminating shared lock state between readers (at the expense of forcing write requests to check additional state) and by eliminating atomic instructions from read lock/unlock logic. It also enables *sequences* of reads to access cached lock state exclusively if they are uninterrupted by writes.

Second, in Sec. 4, we present an experimental evaluation of the PF-L compared to other alternatives on the basis of throughput and locking overheads. Across all of our experiments, the PF-L enabled throughput increases over the state-of-the-art PF-T in the range 2–450%, and overhead reductions for reading in the range 40–73% for read-dominant workloads.

Third, in Sec. 5, we extend prior inflation-free blocking analysis proposed for mutex locks [9] to apply to reader/writer locks. This analysis involves modifying an integer linear program (ILP), as the introduction of reads requires applying numerous additional constraints.

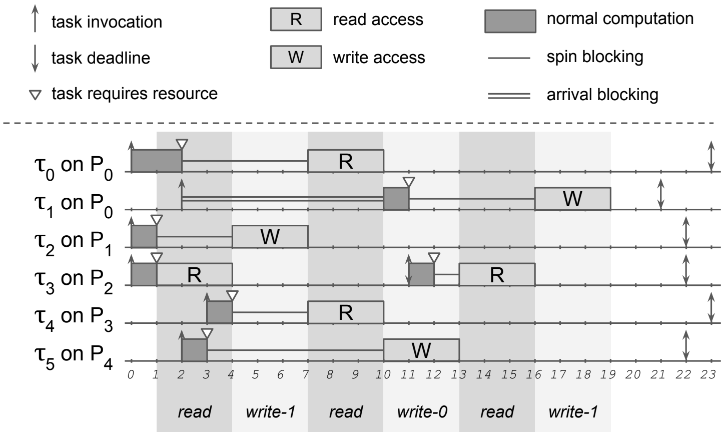
Fourth, in Sec. 6, we present the results of a schedulability study that we conducted to compare our new PF-L implementation and inflation-free reader/writer blocking analysis to prior alternatives. In this study, our new analysis improved schedulability by up to 159% compared to previous state-of-the-art methods.

## 2 Background

In this section, we present our assumed models and relevant background and related work.

**Task model.** We consider a sporadic task system $\Gamma$ comprised of $n$ constrained-deadline tasks scheduled by the Partitioned Earliest-Deadline-First (P-EDF) scheduler on a multiprocessor platform with $m$ cores. (We assume familiarity with the sporadic model and P-EDF.) An arbitrary task is denoted $\tau_i$. When conducting analysis, the partition (core) under consideration is denoted $P^*$, and an arbitrary partition is denoted $P_k$.

**Resource model.** We assume a set of $n_r$ shared resources, with an arbitrary resource denoted $\ell_q$. A job of a task can issue a request for only one resource at a time (no nesting). Each request is either a *read request* (which may execute concurrently with other reads) or a *write request* (which must be exclusive). We use $\mathcal{R}_i^r$, $\mathcal{R}_i^w$, and $\mathcal{R}_i$ to denote a read, write, or arbitrary (read or write) request, respectively, issued by a job of $\tau_i$. Once a request $\mathcal{R}_i$ for a resource $\ell_q$ has been granted, $\mathcal{R}_i$ is *satisfied*, and the issuing job *holds* $\ell_q$ until $\mathcal{R}_i$ *completes*.



| time | read queue | write queue |
|---|---|---|
| [0, 1) | - | - |
| [1, 2) | - | $\mathcal{R}_2$ |
| [2, 3) | $\mathcal{R}_0$ | $\mathcal{R}_2$ |
| [3, 4) | $\mathcal{R}_0$ | $\mathcal{R}_2$, $\mathcal{R}_5$ |
| [4, 7) | $\mathcal{R}_0$, $\mathcal{R}_4$ | $\mathcal{R}_2$, $\mathcal{R}_5$ |
| [7, 10) | - | $\mathcal{R}_5$ |
| [10, 11) | - | - |
| [11, 12) | - | $\mathcal{R}_1$ |
| [12, 13) | $\mathcal{R}_3$ | $\mathcal{R}_1$ |
| [13, 16) | - | $\mathcal{R}_1$ |
| [16, 21) | - | - |

**Figure 2** Phase-fair request satisfaction.

**Phase-fair locks.** Phase-fair (PF) reader/writer (RW) locks were proposed to improve blocking bounds in real-time systems [12]. Some prior approaches to RW locking can starve read (resp., write) requests by prioritizing writes (resp., reads) over them [16, 25]. PF locks instead employ alternating read and write phases. Assuming non-preemptive spin-based locking (as we do here), the orchestration of these phases is defined by four rules [12]:

**PF1** reader phases and writer phases alternate;
**PF2** writers are subject to FIFO ordering, but only with regard to other writers;
**PF3** at the start of each reader phase, all currently unsatisfied reads are satisfied (exactly one write request is satisfied at the start of a writer phase); and
**PF4** during a reader phase, newly issued read requests are satisfied only if there are no unsatisfied write requests pending.

▶ **Example 1.** Consider the six tasks depicted in Fig. 2. For simplicity, we assume that each task requires access to the same resource; the type of access is indicated for each request. At time $t = 1$, the first job of $\tau_3$ issues a read request, $\mathcal{R}_3^r$, which is satisfied immediately. Just after this, at $t = 1 + \epsilon$, $\tau_2$ issues a write request $\mathcal{R}_2^w$, which must wait (Rule PF1). At $t = 2$ and $t = 4$, $\tau_0$ and $\tau_4$ issue read requests $\mathcal{R}_0^r$ and $\mathcal{R}_4^r$, respectively, which also must wait (Rule PF4). At $t = 3$, $\tau_5$ issues a write request, which waits; it will not be satisfied before the write request issued by $\tau_2$ (Rule PF2). When $\mathcal{R}_3^r$ completes, $\mathcal{R}_2^w$ is satisfied (Rule PF1). When $\mathcal{R}_2^w$ completes, both $\mathcal{R}_0^r$ and $\mathcal{R}_4^r$ are satisfied (Rule PF3). Read and write phases continue to alternate, as shown. Note that the job of $\tau_1$ released at $t = 2$ does not preempt the currently executing job of $\tau_0$ on $P_0$ because the latter is executing non-preemptively.

Different implementation strategies can be applied to realize the phase-fair rules. Brandenburg and Anderson presented several implementations, including a ticket-lock-based implementation (PF-T), a more compact version for embedded systems (PF-C), and a queue-based implementation with $O(1)$ RMR time complexity (PF-Q) [12]; as discussed more fully later, RMR time complexity counts only operations that entail an interconnect traversal to access memory. An alternative $O(1)$ implementation has been given by Bhatt and Jayanti [8].

**Blocking analysis.** When checking schedulability, locking delays must be accounted for. The simplest approach involves *inflating* execution times by *per-request* worst-case blocking bounds. This approach is safe but pessimistic, as the worst case may not always occur.

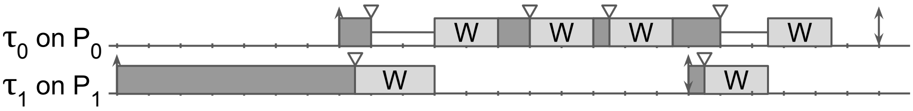


**Figure 3** Example of blocking between two tasks.

Recently, *holistic, inflation-free* analysis, which accounts for blocking over an analysis interval, has been proposed for mutexes [9, 27, 30]. Such analysis seeks to avoid over-estimating locking delays.

▶ **Example 2.** Fig. 3 depicts a schedule of two tasks, $\tau_0$ and $\tau_1$, that write a common resource. Note that each request of $\tau_0$ can potentially be blocked by one request of $\tau_1$. Under conventional inflation-based blocking analysis, the execution time of $\tau_0$ would thus be inflated by the cost of four requests. However, only two requests may be issued by $\tau_1$ during one job of $\tau_0$. Holistic analysis leverages such knowledge to more tightly bound total blocking, and an inflation-free approach accounts for this blocking without inflating task execution times.

**Related work.** As mentioned in Sec. 1, RCU, like our PF-L implementation, was designed for read-dominant workloads. However, RCU is not linearizable [19]. Furthermore, RCU efficiently executes reads by requiring writes to copy shared-object state and this copying results in a need for garbage collection. To our knowledge, no schedulability analysis exists for RCU, in part due to the non-deterministic behavior of garbage collection.

In addition to the RW locks already mentioned, FIFO-based and reader-preference locks have been developed for higher throughput [23], and reader-preference, writer-preference, and no-preference RW locks with $O(1)$ RMR time complexity (see Sec. 3) have been presented [7].

Of the PF variants by Brandenburg and Anderson mentioned earlier, the two ticket-lock-based approaches, the PF-T and PF-C, have $O(m)$ RMR time complexity, while the queue-based PF-Q has $O(1)$ RMR time complexity [12]. However, when measuring worst-case overheads, the sub-optimal PF-T outperforms the asymptotically optimal PF-Q [12] due to the lower frequency of atomic operations. Similarly, the $O(1)$ implementation of Bhatt and Jayanti [7] also includes multiple atomic instructions within the entry and exit sections of both reads and writes. We therefore focus our experiments to compare with PF-Ts.

Brandenburg and Anderson also developed suspension-based phase-fair implementations for clustered-scheduled systems [13].[1] Finally, Ward and Anderson applied phase-fair reasoning to support nested reader/writer locking [28].

## 3 The PF-L: A New Phase-Fair Lock with Light Reading

Our proposed PF-L is shown in Alg. 1. In this section, we described its motivation, data structures and how its code works, and analyze its RMR time complexity.

**PF-L motivation.** Recall from Fig. 1 that PF-T fails to scale for a read-only workload. We conjectured this was due to overheads, particularly cache-line bouncing and interconnect traffic triggered by updates to shared lock state. Therefore, we designed the PF-L to isolate, with respect to caches, lock state where possible, especially among reads on different cores.

[1] Clustered scheduling generalizes partitioned and global scheduling.

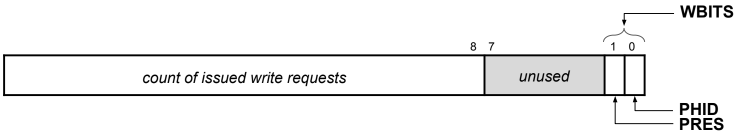


**Figure 4** Usage of *win*.

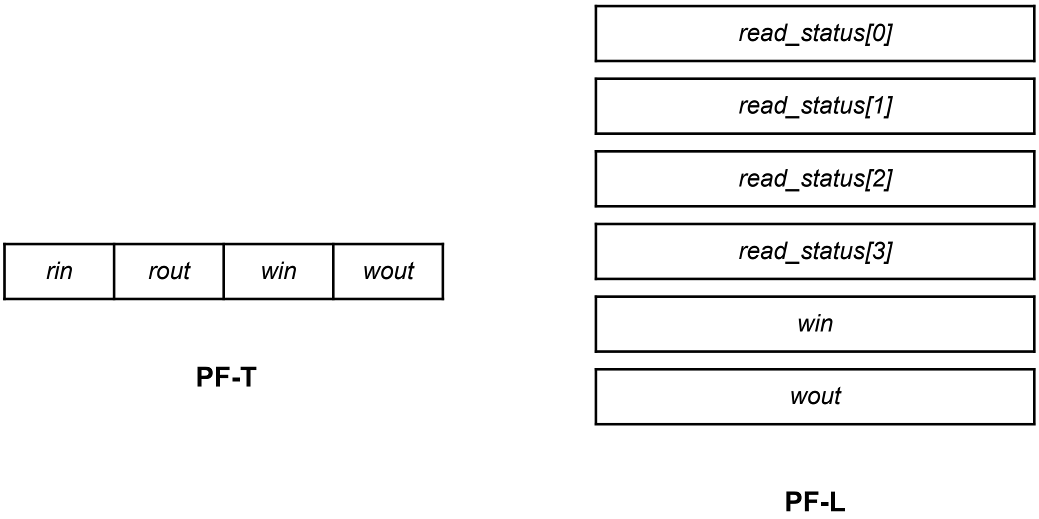


**Figure 5** Distribution of locking protocol data structures across cache lines.

**Data structures.** Like the PF-T, the PF-L uses variables to count the number of write requests "in" and "out" (*i.e.*, the number issued and completed), *win* and *wout*, respectively. In the lowest-order byte of *win*, the PF-L maintains two bits indicating if a write request is present and the current write-phase – each write request is satisfied during either a 0-phase or a 1-phase. (The alternation of these phases prevents a race condition in which read requests could otherwise fail to distinguish the end of one write phase from the waiting of the subsequent active write request.) `WBITS` refers to these two bits, with `PRES` being the writer-present bit and `PHID` the write-phase bit, as illustrated in Fig. 4. The PF-T uses two global counters for the number of read reqests issued (*rin*) and the number of read requests completed (*rout*). In contrast, the PF-L uses a per-core variable, *read_status*, to maintain the status of any read requests. This difference is illustrated for a four-core system in Fig. 5, in which all of the variables for the PF-T are stored on the same cache line and each variable for the PF-L is allocated a separate cache line. Even if *rin* and *rout* were separated in the PF-T, all read requests would require updating those same locations. Instead, in the PF-L, a read request only updates *read_status* for its processor, avoiding conflicts with other read requests. This per-core definition enables isolating the variables to reduce cache interference (described in depth below). Coordination of read and write phases is achieved by requests updating and reading these variables. As such, this lock state is essential to ensuring linearizability [19].

**Code description.** We explain the code in Alg. 1 by walking through part of the example illustrated in Fig. 2. We describe the execution of the code at several time instants.

**Algorithm 1** Phase-Fair with Light Reading (PF-L).

```
 1: type res_state: record // all aligned on different cache lines
 2:     read_status: array of unsigned integer, each initially COMPLETED    ▷ Cache aligned
 3:     win, wout: unsigned integer, initially 0

 4: constant
 5:     WINC 0x100  // writer increment value
 6:     WBITS 0x3  // writer bits in win
 7:     PRES 0x2  // writer-present bit
 8:     PHID 0x1  // write-phase bit
 9:     PRESENT 0x3  // reader present indicator
10:     COMPLETED 0x4  // reader completed indicator

11: procedure READ_LOCK($\ell$: ptr to res_state, k: core index)
12:     var w: unsigned int
13:     $\ell$ →read_status[k] := PRESENT
14:     w := $\ell$→win & WBITS
15:     $\ell$ →read_status[k] := w & PHID    ▷ To wait on write phase (w & PHID), if active
16:     await (w & PRES = 0) or ($w \neq$ ($\ell$→win & WBITS))    ▷ Satisfied

17: procedure READ_UNLOCK($\ell$: ptr to res_state, k: core index)
18:     $\ell$ →read_status[k] := COMPLETED

19: procedure WRITE_LOCK($\ell$: ptr to res_state)
20:     var wticket, read_waiting: unsigned int
21:     wticket := fetch&add($\ell$→win, WINC) and ¬WBITS    ▷ In write queue
22:     await (wticket = $\ell$→wout)    ▷ Head of write queue
23:     fetch&xor($\ell$→win, 0x3)    ▷ Marked present and new phase for reads to see
24:     read_waiting := $\ell$→win & PHID
25:     for k in core numbers do
26:         await (read_status[k] = read_waiting) or (read_status[k] = COMPLETED)

27: procedure WRITE_UNLOCK($\ell$: ptr to res_state)
28:     fetch&and($\ell$→win, 0xFFFFFF01)    ▷ Clear PRES, but keep PHID
29:     $\ell$→wout := $\ell$→wout + WINC
```

**Time $t = 1$.** When $\mathcal{R}_3^r$ is issued on $P_2$, it first marks *read_status*$[2] =$ `PRESENT` (Line 13). Then it reads the value in *win* (Line 14). This is the first request in the system, so $w = 0$. Now, *read_status*$[2] = 0$, indicating that $\mathcal{R}_3^r$ would wait for a satisfied write request in Phase 0, if there is one, but none exists, so $\mathcal{R}_3^r$ is satisfied immediately (Line 16).

Just as $\mathcal{R}_3^r$ finishes executing READ_LOCK, $\mathcal{R}_2^w$ begins executing WRITE_LOCK. $\mathcal{R}_2^w$ increments *win* (Line 21), storing *wticket* $= 0$ and waits for *wticket* $=$ *wout* (Line 22). This serves as a ticket lock to ensure at most one write request is executing any of the following lines of WRITE_LOCK. Next, $\mathcal{R}_2^w$ sets the writer-present bit and flips the write-phase bit (Line 23), resulting in the last two bits of *win* holding 0b11. This is how the presence and phase of an active write request is shared with read requests. $\mathcal{R}_2^w$ then computes that a read waiting for the completion of its write phase would display a *read_status* value of 1. $\mathcal{R}_2^w$ next checks for active read requests on each core. For $P_0$ and $P_1$, it reads the *read_status* as `COMPLETED` and proceeds. However, for $P_2$, $\mathcal{R}_2^w$ reads *read_status*$[2] = 0$. Thus, the read request on $P_2$ is not waiting for $\mathcal{R}_2^w$ but is satisfied; $\mathcal{R}_2^w$ waits for this request to complete.

**Time $t = 2$.** As illustrated in Fig. 2, while $\mathcal{R}_2^w$ waits, $\mathcal{R}_0^r$ is issued. At $t = 2$, the resource is in a read phase, but the waiting write request requires $\mathcal{R}_0^r$ to wait until the subsequent read phase (by Rule PF4). This is accomplished in READ_LOCK as follows. $\mathcal{R}_0^r$ sets *read_status*[0]=`PRESENT`, stores $w = 3$, and sets *read_status*$[0] = 1$. It then awaits a change in the `WBITS` of *win*, which will not occur until $\mathcal{R}_2^w$ completes. Note that from the perspective of $\mathcal{R}_2^w$, $P_0$ was already checked for active read requests, but the newly issued read request will safely wait based on the check of *win*, so no additional checks are required.

**Time $t = 4$.** Once $\mathcal{R}_3^r$ completes, and $\mathcal{R}_2^w$ becomes satisfied by the phase-fair rules. We now illustrate how that is accomplished in the PF-L. When $\mathcal{R}_3^r$ completes, it marks *read_status*[2] = `COMPLETED` (Line 18). Then $\mathcal{R}_2^w$ sees *read_status*[2] = `COMPLETED` and resumes checking cores. Next, $\mathcal{R}_2^w$ checks $P_3$ and sees *read_status*[3] = `PRESENT`, as $\mathcal{R}_4^r$ has just been issued. However, like $\mathcal{R}_0^r$, $\mathcal{R}_4^r$ soon sets *read_status*[3] = 1, indicating that the read request on $P_3$ is waiting for the execution of a write Phase 1 ($\mathcal{R}_2^w$'s write phase). $\mathcal{R}_2^w$ proceeds, reads *read_status*[4] = `COMPLETED`, and becomes satisfied.

**Time $t = 7$.** Once $\mathcal{R}_2^w$ completes, it clears the writer-present bit (Line 28); the last two bits of *win* subsequently hold 0b01, indicating that there is not a writer present and that the prior write phase was Phase 1. The waiting read requests, $\mathcal{R}_0^r$ and $\mathcal{R}_4^r$, observe this change and are satisfied immediately. Next $\mathcal{R}_2^w$ increments *wout* (Line 29), prompting $\mathcal{R}_5^w$ to execute the remaining logic of WRITE_LOCK.

**RMR time complexity.** The *remote memory references (RMR)* time-complexity measure was proposed in work on spin-based synchronization algorithms [33]. Under this measure, only operations that generate an interconnect traversal are counted; other operations are ignored. In applying this measure, architectural details are dealt with somewhat abstractly. In this work, we use a refined notion of RMR time complexity that incorporates such details.

Specifically, we assume a *write-back*, *write-invalidate* cache coherence protocol [17], which is consistent with many commodity processors (*e.g.*, x86 Intel and AMD processors). Abstractly, a write-back cache is one in which a memory write is cached and not written until later necessary (*e.g.*, due to a cache eviction). In a write-invalidate cache, when a memory write occurs, if that address is cached on a remote core, it is marked as *invalid* and subsequent accesses must be re-read. Any communication among caches is performed over an interconnect that all caches *snoop* or listen upon for any events that require updating their state. This interconnect introduces latency into cache and memory operations. We refer the reader to [18] for further discussion, but highlight the two most salient properties of such caches that influence the PF-L's design:

**C1** When a cache block is written it becomes *write hot.* Any subsequent core-local reads or writes of that block do not generate interconnect traffic while the block is write hot. The block stays write hot until it is evicted, or read or written by another core.

**C2** A cache block that is read that is not write hot becomes *read hot.* Any subsequent core-local reads of that block do not generate interconnect traffic while the block is read hot. The block stays read hot until it is evicted or modified by any core.

Given this model, we define a *local memory reference (LMR)* to be one in which no interconnect traversal is generated from the L1 data cache. For simplicity, we assume that atomic operations generate interconnect traffic, and are therefore not LMRs. Conversely, *remote memory references (RMRs)* are ones that are not local.

When analyzing RMR time complexity, we assume there are no *conflict misses*, *i.e.*, that there is sufficient cache space for all lock state to be cached concurrently. Furthermore, we assume cached lock state persists both during and between critical sections. Finally, we assume there are no cache evictions due to preemptions or migrations, as such costs are typically accounted for through separate analyses [6]. While in practice these assumptions may not always hold, they enable analysis of RMRs inherent to the protocol over a sequence of lock invocations, rather than on a per-invocation basis. This is relevant in cases where there is high lock contention, and potentially many requests to the same lock by one task.

**RMR time complexity of the PF-L.** Assuming the cache behavior defined above, the PF-L has $O(1)$ *amortized* RMR time complexity for an arbitrary sequence of $r$ consecutive read requests on the same core uninterrupted by a write request, instead of $\Omega(r)$ as in all prior phase-fair approaches. Towards establishing this, we define a *read interval* to be an interval $[t, t')$ in which there is no pending or completed write request. Note that read requests from any and all cores may be issued and satisfied during a read interval. Now consider a read interval $[t, t')$ and a sequence of read requests $\mathcal{R}^r_1, \ldots, \mathcal{R}^r_r$ on core $P^*$ that are issued after $t$ and completed before $t'$. We make no assumption about the initial cache state at time $t$, and therefore at Line 16, $\mathcal{R}^r_1$ incurs an RMR to cache *win*. This leaves *win* read hot and *read_status*$[P^*]$ write hot on $P^*$ when $\mathcal{R}^r_1$ completes. $\mathcal{R}^r_1$ therefore incurs $O(1)$ RMRs.

Continuing inductively, we show that each subsequent read request $\mathcal{R}^r_j$ where $j \in \{2, \ldots, r\}$, incurs no RMRs, and leaves the cache in the same state. First, observe that *win* is only modified by write requests (Lines 21, 23, and 28), which by definition do not occur in a read interval. Therefore, *win* will not be invalidated by $\mathcal{R}^r_j$, and will remain read hot (C2). Thus, any access to *win* by $\mathcal{R}^r_j$ will be an LMR.

Next, observe that *read_status*$[P^*]$ is **(i)** only modified by read requests on $P^*$ (Lines 13, 15, and 18), and **(ii)** only read by write requests, which by definition do not occur in the read interval. Thus, the accesses to *read_status*$[P^*]$ are writes to a write-hot block, which are LMRs, and leave *read_status*$[P^*]$ write hot (C1). Taken together, this reasoning inductively proves $O(1)$ amortized RMR time complexity as claimed.

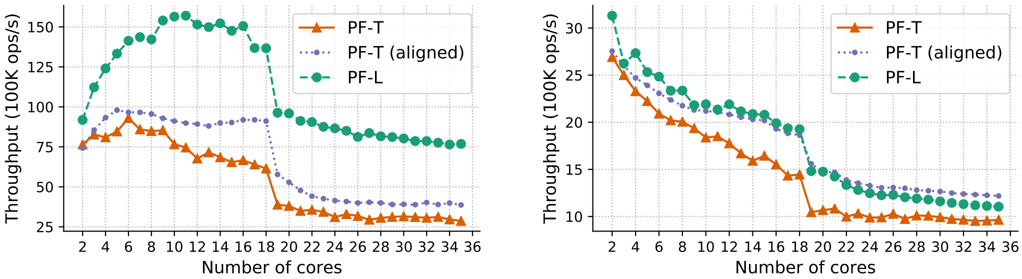
We note that, in the presence of write requests, reads in the PF-L have $O(m)$ RMR time complexity. In particular, before a read request is satisfied, it spins on *win*, which may be updated by at most $m-2$ other newly issued write requests. Write requests in the PF-L clearly have $O(m)$ RMR time complexity. The spinning at Line 22 generates $O(m)$ RMRs (like any ticket lock), as does the FOR loop at Lines 25-26.

## 4 Evaluation of the PF-L

We empirically compared the PF-L to Brandenburg's PF-T implementation [11]. The results of this comparison include throughput graphs, including Fig. 1, as well as overhead data measured as a function of varying workloads. We conducted all experiments on a two-socket, 18-cores-per-socket x86 machine running the Linux 4.9.30 LITMUS$^{\mathrm{RT}}$ kernel [2], with two Intel Xeon E5-2699 v3 CPUs @ 2.30 GHz, 128 GB of RAM, and three levels of cache: per-core 32 KB L1 data and instruction caches, 256 KB L2 caches shared by pairs of cores, and 46,080 KB L3 caches shared by all cores on the same socket. We performed each evaluation on $m \in \{2, \ldots, 36\}$ cores and two sockets. For $m \leq 18$, only one socket was used.

Recall that in the PF-L all lock-status variables are aligned to be cached on different lines. This allows each *read_status* variable to exist in a core-local L1 cache and never be invalidated by readers on other cores. Brandenburg's PF-T variables are all packed into a single cache line by design to minimize cache-line reloading costs [11]. All subsequent references to the PF-T are to Brandenburg's original implementation unless otherwise stated.

In conducting the following experiments, the contents of the cache were not protected. However, these experiments were conducted in isolation, so the cache behavior can be entirely attributed to the experiments. We did not conduct experiments in which another workload was designed to evict cache lines, as our focus was on capturing the overhead of cache evictions inherent to the execution of the protocol itself. There is prior work on protecting caches lines in real-time systems [4, 14, 15, 20, 21, 22, 29, 31, 32, 34], and one of these approaches could be applied to ensure competing workloads in a system do not evict the data structures of the locking protocol from the cache.



**(a)** 5% inserts, 95% lookups

**(b)** 50% inserts, 50% lookups

**Figure 6** Red-black tree throughput.

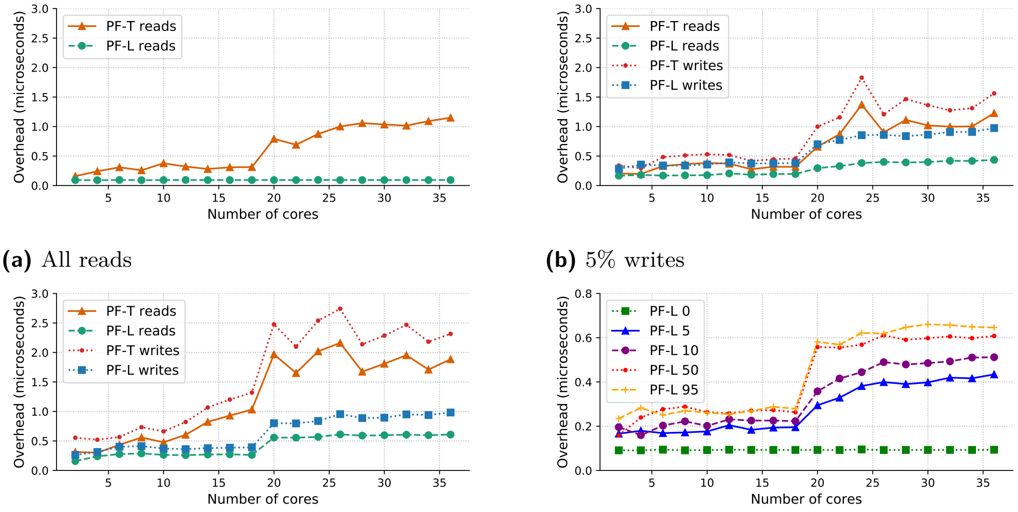
**Throughput.** We evaluated throughput using a realistic workload of lookups (reads) and inserts (writes) in a shared red-black tree, as this was the motivating use case that inspired this paper. Throughput was measured holistically to include locking overheads, blocking, and varying critical-section lengths based on whether an operation was a read or write. All operations were partitioned evenly across cores executing a single task per core. For each experiment, we averaged the throughput for ten unique random trees, each with a million nodes. The results for an *all-read* workload are shown in Fig. 1, which includes a plot for the same experimental setup with no synchronization. Fig. 6 presents throughput trends for a *read-dominant* workload and a workload with *evenly distributed* reads and writes.

▶ **Observation 3.** *The PF-L exhibited linear scaling with increasing core counts for an all-read workload.*

Fig. 1 highlights the pitfalls of the PF-T for an all-read workload. In comparison, throughput under the PF-L scaled linearly with the core count as the *read_status* variable was maintained write hot in the L1 data cache. Cache behavior enabled better scaling on one-socket for read-dominant workloads, as shown in Fig. 6a. On two sockets, throughput decreased with higher core counts due to more expensive interconnect operations. For more balanced workloads of reads and writes, as shown in Fig. 6b, throughput did not increase for either the PF-L or PF-T. This is because, with both reads and writes present, the RMR complexity for all requests in the PF-L is $O(m)$, as shown earlier. However, throughput was still higher by up to 25% than for the PF-T due to overheads.

After observing the benefits of cache-aligned variables in the PF-L, we tested aligning each PF-T variable in its own cache line. We discovered this version outperformed the original PF-T– a useful contribution in its own right. Throughput for the cache-aligned PF-T is also shown in Fig. 6; the cache-aligned PF-T actually performed similarly to the PF-L in the evenly distributed workload of reads and writes on high core counts.

**Overheads.** In measuring overheads, it is necessary to distinguish time spent in operations inherent to the algorithm (overheads) from those incurred while spinning (blocking). For overhead-measurement purposes only, we instrumented both the PF-T and the PF-L to measure overheads and blocking separately. We recorded blocking and overhead times for 100,000 lock and unlock calls across an increasing number of cores. To simulate high contention and record worst-case overheads, critical sections were empty. All figures present the $99^{th}$ percentile observed overheads to filter outliers due to interrupts and other jitter due to userspace timing. Fig. 7 shows overhead trends for several different workloads. Overheads were measured separately for reads and writes, each including both lock and unlock costs.



**(a)** All reads

**(b)** 5% writes

**(c)** 50% writes

**(d)** the PF-L with varying write %

**Figure 7** Total overheads for lock and unlock operations.

▶ **Observation 4.** *The PF-L exhibited constant overheads for an all-read workload.*

Fig. 7a shows that the PF-L exhibited constant lock and unlock overheads of about $0.1\mu s$ across both sockets, while the PF-T overheads were on average $0.4\mu s$ on one socket, and up to $1.3\mu s$ on two sockets. This is attributable to the fact that in the PF-L, read lock and unlock operations only modify a single core-local variable. The PF-T read lock atomically increments a shared variable, which in turn invalidates other caches and bounces the variable across cores and sockets, yielding increased overhead.

As write percentages increase, read operations become more costly as *read_status* variables are read by writers on other cores and the write-related variables are constantly updated on all cores with read requests. This behavior also causes an increase in write overheads. The PF-T experienced higher overheads for read-dominant (Fig. 7b) and evenly distributed (Fig. 7c) workloads. Since all PF-T variables are on a single cache line, each update invalidates cache-line values for all other cores, resulting in an RMR for every entry and exit section.

▶ **Observation 5.** *For all workloads with some writes, overheads increased by up to 3× on two sockets.*

All insets in Fig. 7 show higher overheads on two sockets other than the PF-L for an all-read workload. This is attributable to higher cross-socket RMR latencies for both the PF-T and the PF-L for mixed workloads. Fig. 7a highlights the case in which reads in the PF-L generate no RMRs (by design) and does not exhibit increased overheads when executing on two sockets. This claim is further supported by throughput results in Fig. 6, where execution on two sockets consistently yielded lower throughput.

▶ **Observation 6.** *Reading under the PF-L incurred less overhead than reading under the PF-T.*

For all tested scenarios across varying write percentages and core counts, read operations under the PF-L yielded lower overheads than the PF-T. Fig. 7d shows trends in read overheads with varying write percentages. Beyond 50% writes, overheads were consistent for all read operations and at most $0.7\mu s$. The PF-L overheads for write-dominant workloads did not appreciably increase beyond 50% writes. With more writes, cache-line invalidations become frequent and cause higher overheads.

## 5 Schedulability Analysis of Phase-Fair Reader-Writer Locks

Recent work [9] presented an analysis framework for P-EDF built around a prior schedulability test [5]. Within this framework, each processor is analyzed in turn, incorporating the delays caused to the execution of tasks on that processor due to waiting for access to shared resources. In the discussion below, the processor under consideration is denoted $P^*$. The schedulability framework uses a fixed point iteration to bound the length of the analysis interval on $P^*$, which we denote $\mathcal{I}$, by using the concept of an *arrival curve (AC)* [26] and *processor demand criterion (PDC)* [5]. At each iteration, a bound on the delays over $\mathcal{I}$ caused by shared resources is required; this bound is what we must provide. Along with the original presentation of the analysis framework, an integer linear programming approach to bounding delays for mutex locks was given [9]. In order to apply this analysis framework to a system in which shared resources are instead managed by phase-fair reader/writer locks, we must instead provide bounds for that locking protocol. The schedulability framework is described in full detail in prior work [9], and the remainder of this section is devoted to determining a bound on delays under phase-fair reader/writer locks.

We build on the previously presented inflation-free analysis for mutex locks [9] to obtain such analysis. We begin by describing the types of delay, along with the constants and variables used in the formulation of our optimization problem. The remainder of the section is devoted to showing that the constraints we apply hold.

**Types of delay.** To check schedulability, analysis is required to bound *synchronization delay*, which includes delays due to both spinning and non-preemptive execution. *Spin delay* is the delay incurred on $P^*$ when a task on $P^*$ waits for a resource by spinning. *Arrival delay* is the delay on $P^*$ that is incurred when a job is unable to begin executing due to the non-preemptive execution of a lower-priority job. Note that the job executing non-preemptively may be either spinning or executing with a satisfied request. Both types of blocking are illustrated in Fig. 2.

To constrain the computed arrival blocking, the inflation-free approach [9] leverages two key observations that are derived from existing schedulability analysis [5], generalized here:

- O1: Arrival blocking in an analysis interval $\mathcal{I}$ of length $t$ is caused only by tasks with a relative deadline larger than $t$.
- O2: Only a single blocking request can cause arrival blocking.

In the rest of this section, we describe the creation of the optimization problem that we define for $\mathcal{I}$ to compute the maximum synchronization delay (denoted $B(P^*, t)$). This problem can be solved with a linear-programming solver, such as GLPK [1]. While we build on an existing framework, the assumptions that informed the construction of the approach for mutex locks do not all hold for phase-fair locks, which require new reasoning.

We begin by describing the constants and variables of our optimization problem. Then, we briefly describe the set of constraints that are straightforward modifications of the original approach; the proofs of these constraints are given in App. A. Finally, we present the constraints that require new reasoning unique to PF locks.

**Constants and variables.** We conduct schedulability analysis for each partition separately. Here, we focus on the analysis for partition $P^*$. We denote the set of all partitions by $\mathcal{P}$, and the set of all tasks by $\Gamma$. We refer to the set of tasks partitioned to a remote processor (any processor other than $P^*$) as $\Gamma^r$, and the set of tasks on a given processor $P_k$ as $\Gamma(P_k)$. We denote the period of an arbitrary task, $\tau_i$, by $T_i$, and its relative deadline by $D_i$. We reason about an arbitrary resource $\ell_q$ in the set of all resources $Q$. We use constants for the number of requests each job issues and the duration of requests by type; we denote the maximum duration of a read (resp., write) request issued by a job of $\tau_i$ for a resource $\ell_q$ with $L^R_{i,q}$ (resp., $L^W_{i,q}$). A job of $\tau_i$ issues at most $N^R_{i,q}$ (resp., $N^W_{i,q}$) read (resp., write) requests.

The following variables are used in our optimization problem to bound blocking:

- $X^{S,R}_{i,q}$ is the spin delay caused by read requests issued by $\tau_i$ for $\ell_q$.
- $X^{S,W}_{i,q}$ is the spin delay caused by write requests issued by $\tau_i$ for $\ell_q$.
- $X^{A,R}_{i,q}$ is the arrival blocking caused by read requests issued by $\tau_i$ for $\ell_q$.
- $X^{A,W}_{i,q}$ is the arrival blocking caused by write requests issued by $\tau_i$ for $\ell_q$.
- $A^R_q$ is an indicator (*i.e.*, binary) variable. $A^R_q = 1$ indicates that arrival blocking is caused by a read request for $\ell_q$, whereas $A^R_q = 0$ indicates that no arrival blocking is caused by a read request for $\ell_q$.
- $A^W_q$ is similarly an indicator of arrival blocking caused by a write request for $\ell_q$.

For the specification of the optimization problem given below, we are applying the PDC. As such, the number of jobs $\tau_j$ on $P^*$ (a local task) that must be considered during the analysis interval $\mathcal{I}$ of length $t$ is $nljobs(\tau_j, t) = \left\lfloor \frac{t+T_j-D_j}{T_j} \right\rfloor$. The number of jobs of a remote task $\tau_j$ that must be accounted for is $nrjobs(\tau_j, t) = \left\lceil \frac{t+D_j}{T_j} \right\rceil$. The modifications described previously [9] allow for simple changes to the specification of the optimization problem to instead reason about the AC.

**Optimization problem.** The optimization problem we seek to solve is formulated to maximize the computed blocking subject to a set of constraints that limit this blocking by considering scenarios that cannot occur. This problem is as follows.

**maximize** $B(t) = \sum_{\forall \tau_i \in \Gamma} \sum_{\ell_q \in Q} [(X^{S,R}_{i,q} + X^{A,R}_{i,q}) \cdot L^R_{i,q} + (X^{S,W}_{i,q} + X^{A,W}_{i,q}) \cdot L^W_{i,q}]$

**subject to** the constrains in Tbl. 1.

**Foundational RW constraints.** The first set of constraints builds directly on the inflation-free analysis presented for mutex locks [9], with the distinction that we instead specify read- and write-versions of each variable, as detailed above. We describe these constraints briefly here and present the full versions in App. A.

Constraint (1) limits the computed arrival blocking terms for read and write requests by comparing the relative deadline of each task to the length of the deadline busy-period. Constraint (2) enforces that spin delay can be caused only by tasks remote to $P^*$. Constraints (3) and (4) limit the contribution of each request (read and write requests, resp.) to delays; each request can contribute to either arrival blocking or spin delay, but not both.

The next five constraints focus on arrival blocking. As arrival blocking can be caused by only a single request (Observation O2), it can be caused by either a read request or a write request (not both); this is enforced by Constraint (5). Constraints (6) and (7) leverage the fact that resources for which there are no read (resp., write) requests cannot cause read (resp., write) arrival blocking. Finally, Constraints (8) and (9) bound the total number of read (resp., write) requests that can cause arrival blocking by the binary variable indicating if arrival blocking is caused by a read (resp. write) request for that resource.

**Helper variables.** We introduce four helper variables, $X_{i,q}^{S,\text{R-to-W}}$, $X_{i,q}^{S,\text{R-to-R}}$, $X_{i,q}^{S,\text{W-to-W}}$, and $X_{i,q}^{S,\text{W-to-R}}$, to analyze the spin blocking caused by remote requests by cases. For example, $X_{i,q}^{S,\text{R-to-W}}$ is the number of read requests issued by $\tau_i$ that delay write requests.

**Constraints on spin blocking.** The following constraints limit the spin blocking that can be computed based on the possible interactions between read and write requests. These must account for the access patterns that can occur under phase-fair locks. Constraints (10) and (11) join the helper variables to those counting total read and write spin delay.

**Proof of (10).** Each read by a remote task $\tau_i$ can induce spin delay on a read request or a write request, but not both, on $P^*$, as all requests execute non-preemptively. Thus, the number of read requests of $\tau_i$ for $\ell_q$ that cause spin delay ($X_{i,q}^{S,R}$) is obtained by summing the number that delay read requests ($X_{i,q}^{S,\text{R-to-R}}$) and write requests ($X_{i,q}^{S,\text{R-to-W}}$), respectively. ◀

**Proof of (11).** Similar to that of Constraint (10). ◀

Constraints (12) and (13) limit the contribution of write requests to spin delay by considering the total number of read and write requests on $P^*$ during $\mathcal{I}$.

**Proof of (12).** By Rules PF1 and PF3, a given read request may be delayed by at most one write phase. There are at most $\sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^{R}$ read requests for $\ell_q$ on $P^*$ during $\mathcal{I}$. Thus, that number upper bounds the number of write requests from other processors that can cause delay to read requests for $\ell_q$ on $P^*$, which is $\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-R}}$. ◀

**Proof of (13).** There are at most $\sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^{W}$ write requests on $P^*$ during $\mathcal{I}$. At most one write request per processor can delay the execution of each write request on $P^*$, because write requests are satisfied in FIFO order (by Rule PF2), requests execute non-preemptively, and only one write request is satisfied during each write phase (by Rule PF3). Thus, for each processor $P_k$, the number of write requests delaying write requests on $P^*$ ($\sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$) is bounded by the number of write requests on $P^*$ in $\mathcal{I}$. ◀

Constraints (14) and (15) bound the impact of remote read requests on read requests on $P^*$. We use the following lemma and corollary in verifying them.

▶ **Lemma 7.** *A read request $\mathcal{R}_i^r$ is blocked by at most one read phase and one write phase.*

**Proof.** If there are no active requests when $\mathcal{R}_i^r$ is issued, it will be satisfied immediately.

If instead there are active read requests and no active write requests, $\mathcal{R}_i^r$ is satisfied immediately upon issuance by Rule PF4.

If there are active write requests and no active read requests when $\mathcal{R}_i^r$ is issued, it will be delayed by the current write phase and then satisfied after the currently satisfied write request completes by Rules PF1 and PF3.

If there is at least one active write request and one active read request when $\mathcal{R}_i^r$ is issued, then either a write request or a read request is currently satisfied. If a write request is satisfied, then $\mathcal{R}_i^r$ will be satisfied upon the completion of that request by Rules PF1 and PF3. If instead the resource is in a read phase, $\mathcal{R}_i^r$ must wait for the completion of this read phase (by Rule PF4) and the completion of a single write phase (by Rules PF1 and PF3).

Thus, in all cases, $\mathcal{R}_i^r$ is blocked by at most one read phase and one write phase. ◀

**Table 1** Linear-program constraints. Constraints (1)–(9) are described in App. A.

| Number | Constraint Specification |
|---|---|
| (1) | $\forall \tau \in \Gamma(P^*) \mid D_i \leq t,\, \forall \ell_q \in Q,\, X_{i,q}^{A,R} + X_{i,q}^{A,W} = 0$ |
| (2) | $\sum_{\tau_i \in \Gamma(P^*)} \sum_{\ell_q \in Q} X_{i,q}^{S,R} + X_{i,q}^{S,W} = 0$ |
| (3) | $\forall \tau_i \in \Gamma^r,\, \forall \ell_q \in Q,\, X_{i,q}^{S,R} + X_{i,q}^{A,R} \leq \textit{nrjobs}(\tau_i, t) \cdot N_{i,q}^R$ |
| (4) | $\forall \tau_i \in \Gamma^r,\, \forall \ell_q \in Q,\, X_{i,q}^{S,W} + X_{i,q}^{A,W} \leq \textit{nrjobs}(\tau_i, t) \cdot N_{i,q}^W$ |
| (5) | $\sum_{\ell_q \in Q} A_q^R + A_q^W \leq 1$ |
| (6) | $\forall \ell_q \in Q,\, A_q^R \leq \sum_{\tau_i \in \Gamma(P^*) \mid D_i > t} N_{i,q}^R$ |
| (7) | $\forall \ell_q \in Q,\, A_q^W \leq \sum_{\tau_i \in \Gamma(P^*) \mid D_i > t} N_{i,q}^W$ |
| (8) | $\forall \ell_q \in Q,\, \sum_{\forall \tau_i \in \Gamma(P^*)} X_{i,q}^{A,R} \leq A_q^R$ |
| (9) | $\forall \ell_q \in Q,\, \sum_{\forall \tau_i \in \Gamma(P^*)} X_{i,q}^{A,W} \leq A_q^W$ Constraints adapted from [9] |
| (10) | $\forall \tau_i \in \Gamma^r,\, \forall \ell_q \in Q,\, X_{i,q}^{S,R} = X_{i,q}^{S,\text{R-to-R}} + X_{i,q}^{S,\text{R-to-W}}$ New Constraints |
| (11) | $\forall \tau_i \in \Gamma^r,\, \forall \ell_q \in Q,\, X_{i,q}^{S,W} = X_{i,q}^{S,\text{W-to-R}} + X_{i,q}^{S,\text{W-to-W}}$ |
| (12) | $\forall \ell_q \in Q,\, \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-R}} \leq \sum_{\tau_i \in \Gamma(P^*)} \textit{nljobs}(\tau_i, t) \cdot N_{i,q}^R$ |
| (13) | $\forall P_k \in \mathcal{P},\, \forall \ell_q \in Q,\, \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}} \leq \sum_{\tau_i \in \Gamma(P^*)} \textit{nljobs}(\tau_i, t) \cdot N_{i,q}^W$ |
| (14) | $\forall \ell_q \in Q,\, \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-R}} \leq \sum_{\tau_i \in \Gamma(P^*)} \textit{nljobs}(\tau_i, t) \cdot N_{i,q}^R$ |
| (15) | $\forall \ell_q \in Q,\, \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-R}} \leq \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{i,q}^{S,\text{W-to-R}}$ |
| (16) | $\forall \ell_q \in Q,\, \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-W}}$<br>$\leq \sum_{\tau_i \in \Gamma(P^*)} (\textit{nljobs}(\tau_i, t) \cdot N_{i,q}^W) + \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$ |
| (17) | $\forall \tau_x \in \Gamma^R,\, \forall \ell_q \in Q,\, X_{x,q}^{S,\text{W-to-W}} \leq \textit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\textit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^W)$ |
| (18) | $\forall \tau_x \in \Gamma^R,\, \forall \ell_q \in Q,\, X_{x,q}^{S,\text{W-to-R}} \leq \textit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\textit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^R)$ |
| (19) | $\forall \tau_x \in \Gamma^R,\, \forall \ell_q \in Q,\, X_{x,q}^{S,\text{R-to-R}} \leq \textit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\textit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^R)$ |
| (20) | $\forall \ell_q \in Q,\, A_q^R + A_q^W = 0 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R} + X_{x,q}^{A,W} \leq 0$ |
| (21) | $\forall \ell_q \in Q,\, A_q^R = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R} \leq 1$ |
| (22) | $\forall \ell_q \in Q,\, A_q^R = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W} \leq 1$ |
| (23) | $\forall \ell_q \in Q,\, A_q^R = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$<br>$\leq \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$ |
| (24) | $\forall \ell_q \in Q,\, A_q^W = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$<br>$\leq 1 + \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$ |
| (25) | $\forall P_k \neq P^*,\, \forall \ell_q \in Q,\, A_q^W = 1 \Rightarrow \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W} \leq 1$ |

▶ **Corollary 8.** *If a read request $\mathcal{R}_i^r$ is blocked by $W$ write requests, it is blocked by at most $W$ read phases.*

**Proof.** In the proof of Lemma 7, which enumerated all possible blocking scenarios for a read request $\mathcal{R}_i^r$, the only scenario in which a request $\mathcal{R}_i^r$ is blocked by a read request is when a write request also blocks $\mathcal{R}_i^r$. ◀

**Proof of (14).** During $\mathcal{I}$, there are at most $\sum_{\tau_i \in \Gamma(P^*)} \textit{nljobs}(\tau_i, t) \cdot N_{i,q}^R$ read requests on $P^*$. By Lemma 7, each read requests can be delayed by at most one read phase. Thus the total number of read requests that cause spin blocking for read requests on $P^*$ ($\sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-R}}$) is bounded by the number of read requests on $P^*$. ◀

**Proof of (15).** By Cor. 8, a read request can be delayed by a read phase only if it is also delayed by a write phase. Thus, the total number of read requests causing spin blocking for read requests on $P^*$ ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-R}}$) is bounded by the total number of write requests causing spin blocking for those requests ( $\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{i,q}^{S,\text{W-to-R}}$). ◀

We now examine how read requests can delay write requests.

▶ **Lemma 9.** *If $W$ write phases block a write request $\mathcal{R}_i^w$, at most $W+1$ read phases block $\mathcal{R}_i^w$.*

**Proof.** By Rule PF1, read phases and write phases alternate. Before each of the $W$ write phases that block $\mathcal{R}_i^w$, a read phase can occur. Additionally, after the last blocking write phase and before the satisfaction of $\mathcal{R}_i^w$, an additional read phase can occur. Therefore, at most $W + 1$ read phases can block $\mathcal{R}_i^w$. ◀

Constraint (16) limits read-to-write blocking and its proof leverages Lemma 9.

**Proof of (16).** There are $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^W)$ write request to consider on $P^*$ during $\mathcal{I}$. For each of these requests individually, if some number $W$ write phases block the request, up to $W + 1$ read phases can also block that request, by Lemma 9. In total, $\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$ write requests can block these write requests, by definition. As each write request can incur one additional blocking by a read request, an additional $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^W)$ read requests can block write requests on $P^*$. Thus, in total the number of read requests that can delay write requests ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-W}}$) is bounded by $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^W) + \sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$. ◀

Finally, we constrain the impact of each remote task on tasks on $P^*$ by considering how jobs may overlap based on their respective periods and deadlines. The next two lemmas are used in verifying Constraints (17)–(19).

▶ **Lemma 10.** *The requests for $\ell_q$ issued by a single job of a remote task $\tau_x \in \Gamma^r$ overlap with at most $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^w$ write requests for $\ell_q$ issued by jobs of $\tau_i \in \Gamma(P^*)$.*

**Proof.** The number of jobs of $\tau_i$ that overlap with a single job of $\tau_x$ is at most $\mathit{nrjobs}(\tau_i, D_x)$. Each job of $\tau_i$ issues up to $N_{i,q}^w$ write requests. Thus, the requests from a single job of $\tau_x \in \Gamma^r$ overlap with at most $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^w$ write requests for $\ell_q$ issued by jobs of $\tau_i$. ◀

▶ **Lemma 11.** *The requests for $\ell_q$ issued by a single job of a remote task $\tau_x \in \Gamma^r$ overlap with at most $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^r$ read requests for $\ell_q$ issued by jobs of $\tau_i \in \Gamma(P^*)$.*

**Proof.** Follows as above, but for read requests. ◀

Constraint (17) limits blocking caused by write requests.

**Proof of (17).** By Lemma 10, a single job of a task $\tau_x \in \Gamma^r$ overlaps with up to $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^w$ write requests of an arbitrary task $\tau_i \in \Gamma(P^*)$. Thus, a single job of $\tau_x$ can overlap with a total of $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^W)$ write requests issued on $P^*$. Because of the non-preemptive execution and FIFO satisfaction order of write requests (Rule PF2), each of these write requests on $P^*$ can be delayed by at most one overlapping write request per job of a remote task. During $\mathcal{I}$, $\mathit{nrjobs}(\tau_x, t)$ jobs of $\tau_x$ must be considered. Thus, the total number of write requests of $\tau_x$ that can cause spin delay on $P^*$ ($X_{x,q}^{S,\text{W-to-W}}$) is bounded by $\mathit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^W)$. ◀

Constraints (18) and (19) limit blocking caused to read requests on $P^*$.

**Proof of (18).** Lemma 11 bounds the number of read requests that a job of $\tau_x \in \Gamma^r$ may overlap with. By Lemma 7, at most one write phase can delay each read request, implying that at most one write request per job can delay each read request. Thus, the constraint follows similarly to Constraint (17). ◀

**Proof of (19).** Follows similarly to Constraint (18) by instead applying that each read request can be blocked by at most one read request (by Lemma 7). ◀

**Constraints on arrival blocking.** A single request on $P^*$ can cause arrival blocking by its non-preemptive blocking and then execution. The duration of this arrival blocking is impacted by the type of request that causes it.

The following constraints are indicator constraints; if a variable in the optimization problem holds a specified value, an additional constraint is imposed. Some linear programming solvers allow the direct specification of indicator constraints. Alternatively, each indicator constraint can be converted to a set of linear constraints by using Big-M techniques [10].

Constraint (20) accounts for the case in which no requests for $\ell_q$ cause arrival blocking.

**Proof of (20).** If neither a read request nor a write request for $\ell_q$ can cause arrival blocking ($A_q^R + A_q^W = 0$), the total number of remote requests that can contribute to arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R} + X_{x,q}^{A,W}$) is 0. ◀

Because any arrival blocking is caused by a single request on $P^*$, we apply reasoning based on request type to eliminate blocking that cannot possibly occur. Constraints (21)–(23) apply if a read request causes arrival blocking. Recall that a single read request can be blocked by at most one read request and one write request by Lemma 7.

**Proof of (21).** If a read request on $P^*$ causes arrival blocking ($A_q^R = 1$), at most one read phase can contribute to its delay by Lemma 7. Thus, the total number of read requests from remote processors that can cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$) is bounded by 1. ◀

**Proof of (22).** Similarly, the total number of write requests from remote processors that can cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$) is bounded by 1. ◀

**Proof of (23).** If a read request on $P^*$ causes arrival blocking ($A_q^R = 1$), then by Cor. 8, if it is blocked by $W$ write requests, it will be blocked by at most $W$ read requests. Because of the non-preemptive execution of requests, any requests that contribute to the blocking of the read on $P^*$ that causes arrival blocking are requests issued by tasks on remote processors. Thus, the total number of write requests that block this read request ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$) upper bounds the number of read requests that block this read request ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$). ◀

Constraints (24) and (25) consider arrival blocking that is caused by a write request.

**Proof of (24).** If a write request on $P^*$ causes arrival blocking ($A_q^W = 1$), the number of read requests that can block it is bounded by one more than the write requests causing delay (by Lemma 9). Thus, the total number of remote read requests that cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$) is bounded by one more than the number write requests on remote processors that cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$). ◀

**Table 2** Summary of percentage TSA improvement of LP:PF.

| | Min | Q1 | Median | Q3 | Max |
|---|---|---|---|---|---|
| Inflation:PF | 0.383 | 10.1 | 28.5 | 54.5 | 158.6 |

**Proof of (25).** If a write request on $P^*$ causes arrival blocking ($A_q^W = 1$), by Rules PF2 and PF3 and the non-preemptive execution of requests, at most one write request per remote processor can delay that write request, as requests execute non-preemptively. Thus, the total number of write requests that cause arrival blocking issued by tasks on $P_k$ ($\sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$) is bounded by 1. ◀

## 6 Schedulability Evaluation

To explore the benefit of our new approaches we conducted a schedulability study by using the SchedCAT toolkit [3] and building upon a prior implementation [9].

**Schedulability improvements.** We begin by comparing our inflation-free analysis for phase-fair reader-writer locking protocols (labeled “LP:PF”) to the existing per-request inflation-based PF analysis (labeled “Inflation:PF”). To reduce the time it took to compute schedulability, we applied our holistic analysis for phase-fair locks only if the per-request inflation-based approach failed to be schedulable. The line labeled “NOLOCK” shows the computed schedulability if the delays for resource accesses are ignored.

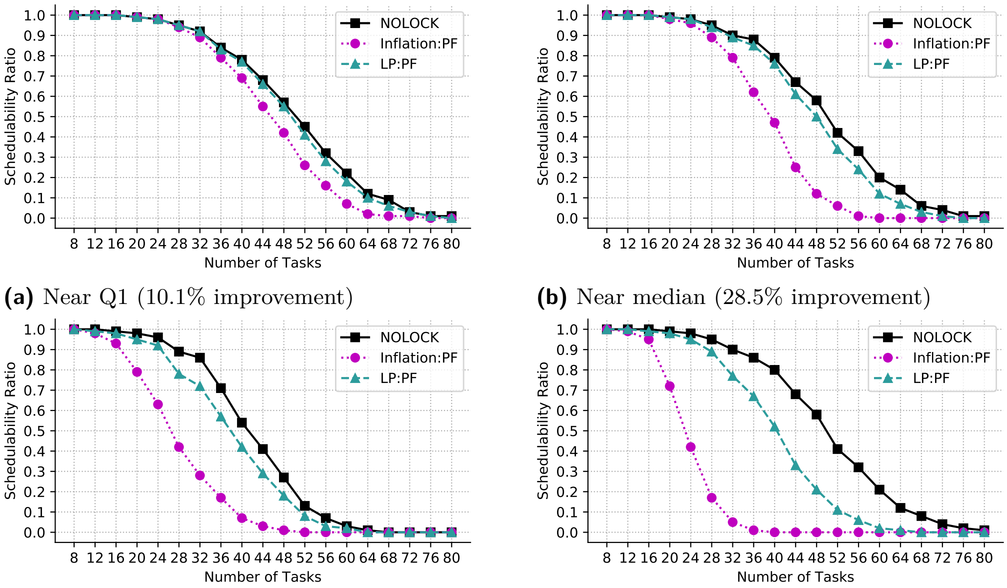
In this study, we computed schedulability for increasing task counts under different scenarios, with 216 scenarios total. Each scenario is a different combination of certain system parameters. We considered a system with eight processors. Task periods were selected from a log-uniform distribution in $[10ms, 100ms]$ or in $[1ms, 1000ms]$. Each task's utilization was chosen from an exponential distribution with a mean of 0.1. The number of resources ($n_r$) in a scenario was selected from $\{4, 8, 16\}$. For each resource, the probability that a task requires that resource was chosen from $\{0.1, 0.25, 0.5\}$. The number of times a task accesses a given resource was either 1 or was selected from $\{1, \ldots, 5\}$. For a given access to be write access (instead of a read access) was chosen with a probability selected from $\{0.01, 0.1, 0.5\}$. Request durations were either short (selected uniformly from $[1\mu s, 25\mu s]$) or medium (selected uniformly from $[25\mu s, 100\mu s]$). These parameters closely reflect those on which the original holistic analysis framework [9] was analyzed.

This study resulted in 216 schedulability graphs (one per scenario), which show the ratio of schedulable tasks systems out of the 1,000 systems generated for each data point. Performance is evaluated on the basis of *task schedulable area (TSA)*, the area under a given curve as computed by a midpoint Riemann sum. In Tbl. 2, we summarize the data on the percentage TSA improvement of LP:PF, and we highlight some key scenarios in Fig. 8.

▶ **Observation 12.** *The LP:PF approach always resulted in a higher TSA than Inflation:PF.*

This is illustrated in Fig. 8. The cases in which LP:PF resulted in the largest percentage improvement (50.% to 158.6%) were primarily for scenarios with write probability of 0.1 or 0.5; 96.9% of these scenarios had write probability of 0.1 or 0.5.

▶ **Observation 13.** *In some scenarios, the LP:PF resulted in only small increases in schedulability.*



**(a)** Near Q1 (10.1% improvement)

**(b)** Near median (28.5% improvement)

**(c)** Near Q3 (54.5% improvement)

**(d)** Significant improvements (110%)

**Figure 8** Comparisons against Inflation:PF.

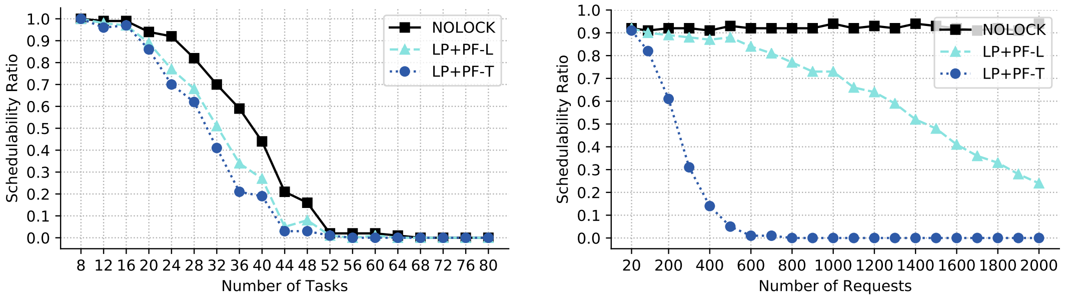
This is illustrated in Tbl. 2 and Fig. 8a. For the scenarios with TSA improvements in the first quartile, in which the LP:PF had a small percentage of improvement, both approaches tended to yield a TSA close to that of NOLOCK.

**Overhead-aware schedulability.** We conducted an additional scheduability study in which we incorporated protocol overheads. We inflated requests by the corresponding overhead and analyzed the resulting systems with our PF analysis; "LP+PF-T" (resp., "LP+PF-L") represents the computed schedulability with the PF-T (resp., PF-L) overheads added. We measured overheads as described in Sec. 4 with eight cores across two sockets for scenarios with up to 10% write requests. For the PF-T, this resulted in read (resp., write) overhead of 2.2$\mu$s (resp., 1.7$\mu$s), and for the PF-L, read (resp., write) overhead of 0.5$\mu$s (resp., 0.9$\mu$s).

In our study of read-dominated workloads (write probability in $\{0.01, 0.1\}$), we observed moderate differences, with an average TSA improvement for LP+PF-L of 1.01%. In some scenarios, the overhead was negligible relative to the blocking. In others, generally those with more resource accesses, the TSA difference was more pronounced. We observed scenarios with up to a 10.4% improvement, as depicted in Fig. 9a. These results support the following.

▶ **Observation 14.** *For read-dominant workloads, our new PF-L protocol and schedulability analysis dominated prior state-of-the art approaches.*

The schedulability improvements initially seemed modest relative to the impacts of lower overhead on throughput. However, the task systems considered in Sec. 4 are quite different (*e.g.*, significant execution time spent in the execution of requests) from those detailed in the schedulability study just discussed. Therefore, to assess the impact of overheads alone (without blocking) in a system with significant resource requirements, we conducted an additional overhead-aware schedulability study that focused on read-only workloads with a variable number of requests for a single shared resource. Here, we applied overheads measured from a system with 0% write requests; thus, we applied overheads of 1.2$\mu$s for



**(a)** Periods selected from {1,1000}ms, 16 resources, 0.5 access probability, up to 5 requests per resource, short request durations, and write probability of 0.01.

**(b)** 32 tasks with periods selected from {1,1000}ms, one resource, access probability of 1.0, and write probability of 0.

**Figure 9** Schedulability with protocol overhead incorporated.

the PF-T and 0.2$\mu$s for the PF-L. Fig. 9b gives the schedulability graph that resulted from this study. These findings are consistent with the throughput experiments (*e.g.*, Fig. 1), and confirm that small overheads can significantly affect throughput and schedulability for synchronization-heavy read-dominant workloads.

## 7 Conclusion

We presented a new phase-fair reader/writer lock implementation and inflation-free PF schedulability analysis that, taken together, can improve both throughput and schedulability in comparison to prior alternatives when supporting read-mostly workloads. While this work was motivated by heavily read-dominant workloads, our findings suggest that the presented lock implementation may be competitive, if not superior, to previous RW locking protocols in most applications. We have demonstrated these improvements via experiments on real hardware and via a schedulability study. In future work, we intend to explore how other concurrent algorithms can be adapted based on cache coherence and performance properties to improve scalability similar to that we have demonstrated herein.

## References

1 GLPK (GNU Linear Programming Kit). `https://www.gnu.org/software/glpk/`.

2 LITMUS$^{RT}$ home page. `http://www.litmus-rt.org/`.

3 SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`, 2020. Accessed: 2020-06-21.

4 S. Altmeyer, R. Douma, W. Lunniss, and R. Davis. Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014.

5 S. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 2006.

6 A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of the 6th Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2010.

7 V. Bhatt and P. Jayanti. Constant RMR solutions to reader writer synchronization. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2010.

8 V. Bhatt and P. Jayanti. Specification and constant RMR algorithm for phase-fair reader-writer lock. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, 2011.

9 A. Biondi and B. Brandenburg. Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, 2016.

10 S. Bradley, A. Hax, and T. Magnanti. Applied mathematical programming, Chapter 9 (Addison-Wesley, 1977). `http://web.mit.edu/15.053/www/AMP-Chapter-09.pdf`, 2021.

11 B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

12 B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87, 2010.

13 B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and $k$-exclusion locks. In *Proceedings of the 9th ACM International Conference on Embedded Software*, 2011.

14 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*, 2001.

15 M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the 36th IEEE International Real-Time Systems Symposium*, December 2015.

16 P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

17 James R Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

18 J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

19 M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

20 J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

21 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013.

22 D. Kirk and J. Strosnider. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *Proceedings of the 11th Real-Time Systems Symposium*, 1990.

23 Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, 2009.

24 P. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, Beaverton, OR, 2004.

25 J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, 1991.

26 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2000.

27 B. Ward. Relaxing resource-sharing constraints for improved hardware management and schedulability. In *Proceedings of the 36th International IEEE Real-Time Systems Symposium*, 2015.

28 B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

29 B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013.

30 A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, 2013.

31 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 *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016.

32 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, 2017.

33 J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

34 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

# A Additional Constraints

Constraints (1)-(9), proven here, are similar to constraints in prior inflation-free analysis [9].

**Proof of (1).** Follows directly from Observation O1. ◀

**Proof of (2).** In order to cause spin delay, a local task must have a satisfied request while another request is blocked. However, because tasks spin and execute critical sections non-preemptively, there can be at most one active request on $P^*$ at any given time. Therefore, tasks on $P^*$ cannot cause spin delay; only remote tasks can cause spin delay. ◀

The following lemma can be proven using reasoning on the behavior of the PF lock similar to that used to prove Lemmas 7 and 9.

▶ **Lemma 15.** *Each remote request $\mathcal{R}_x$ can contribute to delaying requests on $P^*$ at most once, and that delay is realized as either arrival blocking or spin delay, but not both.*

**Proof of (3) and (4).** Both follow from Lemma 15. ◀

Constraints (5)–(9) concern arrival blocking.

**Proof of (5).** Follows from Observation O2; only one request can cause arrival blocking, and each request is only for a single resource and is either a read request or a write request. ◀

**Proof of (6).** Recall that $A_q^R$ is a binary indicator variable. By Observation O1, arrival blocking is only caused by tasks with a relative deadline larger than $t$. If no read request for $\ell_q$ is issued by any task with a deadline greater than $t$ (*i.e.*, the sum on the right-hand side is 0), then is is not possible to have a read request for $\ell_q$ cause arrival blocking. ◀

**Proof of (7).** Similarly, we constrain arrival blocking due to a write request. ◀

**Proof of (8).** For each resource $\ell_q$, the number of read requests that can cause arrival blocking is upper-bounded by $A_q^R$; at most one request can cause arrival blocking (Observation O2), and if $A_q^R = 0$, no request for that resource can cause arrival blocking. ◀

**Proof of (9).** Similarly, the arrival blocking caused by write requests is constrained. ◀

# Schedulability Analysis for Multi-Core Systems Accounting for Resource Stress and Sensitivity

**Robert I. Davis**
Department of Computer Science, University of York, UK

**David Griffin**
Department of Computer Science, University of York, UK

**Iain Bate**
Department of Computer Science, University of York, UK

## Abstract

Timing verification of multi-core systems is complicated by contention for shared hardware resources between co-running tasks on different cores. This paper introduces the Multi-core Resource Stress and Sensitivity (MRSS) task model that characterizes how much stress each task places on resources and how much it is sensitive to such resource stress. This model facilitates a separation of concerns, thus retaining the advantages of the traditional two-step approach to timing verification (i.e. timing analysis followed by schedulability analysis). Response time analysis is derived for the MRSS task model, providing efficient context-dependent and context independent schedulability tests for both fixed priority preemptive and fixed priority non-preemptive scheduling. Dominance relations are derived between the tests, and proofs of optimal priority assignment provided. The MRSS task model is underpinned by a proof-of-concept industrial case study.

**2012 ACM Subject Classification** Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

**Keywords and phrases** real-time, multi-core, scheduling, schedulability analysis, cross-core contention, resource stress, resource sensitivity

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.7

**Funding** Innovate UK HICLASS project (113213) and the ESPRC grants STRATA (EP/N023641/1) and MARCH (EP/V006029/1). EPSRC Research Data Management: No new primary data was created during this study.

**Acknowledgements** The authors would like to thank Rolls-Royce PLC for providing the object code for one of their aero-engine controllers for use in real-time systems research.

# 1 Introduction

## 1.1 Background

The survey published by Akesson et al. in 2020 [1], shows that about 80% of industry practitioners developing real-time systems are using multi-core processors, about twice the number that are using single-cores. On a single-core processor, when a task executes without interruption or pre-emption it has exclusive access to the hardware resources that it needs. The execution time of the task therefore depends *only* on its own behavior and the initial state of the hardware. This is in marked contrast to what happens when a task executes on one core of a multi-core processor. Multi-core processors are typically designed to provide high average-case performance at low cost, with hardware resources shared between cores. These shared hardware resources typically include, the interconnect, caches, and main memory, as well as other platform specific components. As a consequence, the execution time of a task running on one core of a multi-core system can be extended by *interference* due to contention for shared hardware resources emanating from co-running tasks on the other cores.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 7; pp. 7:1–7:26
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This problem of cross-core contention and interference has led to timing verification of multi-core systems becoming a hot topic of real-time systems research in the decade to 2020. The survey published by Maiza et al. in 2019 [32] classifies approximately 120 research papers in this area. Much of this research relies on detailed information about shared hardware resources and the policies used to arbitrate access to them. This information is then used to derive analytical bounds on the maximum interference possible due to contending tasks running on the other cores. In practice, however, there can be substantial difficulties in obtaining and using such detailed low-level information, since it is not typically disclosed by hardware vendors. This is because the complex resource arbitration policies and low-level hardware design features employed comprise valuable intellectual property. Further, even if such information is available, then the overall behavior can be so complex as to preclude a static analysis that provides meaningful bounds, as opposed to substantial overestimates.

The predominant industry practice is to use measurement-based timing analysis techniques to estimate worst-case execution times[1] (WCETs). However, the simple extension of measurement-based techniques to multi-core systems cannot provide an adequate solution that bounds the impact of cross-core interference. This is because cross-core interference is highly dependent on the timing of accesses to shared hardware resources by both the task under analysis and its co-runners. In practice, it is not possible to choose the worst-case combination of behavior (inputs, paths, and timing) for co-running tasks that will result in the maximum interference occurring [33]. A potential solution to this problem, which is being taken up commercially [37], is to employ a more nuanced measurement-based approach using *micro-benchmarks* [36, 16, 33, 24]. These micro-benchmarks sustain a high level of resource accesses, ameliorating the timing alignment issues inherent in the naive approach discussed above. Micro-benchmarks can be used to characterize tasks in terms of the interference that they can cause, or be subject to, due to contention over a particular hardware resource.

The timing verification of single-core systems has traditionally been solved via a *two-step* approach [32]. First context-independent WCET estimates are obtained, either via static or measurement-based timing analysis. Second, these estimates are used as parameter values in a task model, with schedulability analysis employed to determine if all of the tasks can meet their timing constraints when executed under a specific scheduling policy. This separation of concerns between timing analysis and schedulability analysis brings many benefits; however, its effectiveness is greatly diminished in multi-core systems due to the fact that execution times heavily depend on co-runner behavior and the cross-core interference that they bring. Inflating individual task execution time estimates to account for the maximum amount of context-independent interference that could potentially occur during the time interval in which each task executes can result in gross over-estimates that are not viable in practice [27]. Rather, research [2, 10] has shown that it is more effective to consider contention over the longer time frame of task response times.

## 1.2 Contribution and Organization

In Section 2, we introduce the *Multi-core Resource Stress and Sensitivity* (MRSS) task model that characterizes how much each task *stresses* shared hardware resources and how much each task is *sensitive* to such resource stress. The MRSS task model provides a simple interface and a separation of concerns between timing analysis and schedulability analysis, thus retaining

[1] About 66% of the industry practitioners surveyed by Akesson et al. [1] used some form of measurement-based timing analysis, whereas only about 33% used some form of static timing analysis.

the advantages of the traditional two-step approach to overall timing verification. The MRSS task model relies on timing analysis, either measurement-based or static, to provide task parameter values characterizing stand-alone (i.e. no contention) WCETs, resource stresses, and resource sensitivities. Thus, it provides the information needed by schedulability analysis to integrate cross-core interference into the computation of bounds on task response times, and hence determine the schedulability of tasks running on multi-core systems. The MRSS task model is generic and versatile. It supports different types of interference that occur via cross-core contention for shared hardware resources, as follows:

**(i)** *Limited interference* where contention for the resource is ameliorated by parallelism in the hardware. Here, the interference is *sub-additive*, i.e. less than the time that the co-running task on another core spends accessing the resource.

**(ii)** *Direct interference* where the bandwidth of the resource is shared between contending cores, for example with Round-Robin bus. Here, the interference is *additive*, directly matching the time that co-running tasks spend accessing the resource.

**(iii)** *Indirect interference* where contention causes additional interference, over and above the bandwidth consumed by co-running tasks (i.e. a *super-additive* effect), due to changes in the state of the resource that cause further delays to subsequent accesses. An example of indirect interference occurs with main memory (DRAM) [22] when interleaved accesses target different rows, resulting in additional row close and row open operations, increasing memory access latency.

The MRSS task model is not however a panacea, it cannot support *unbounded interference* where task execution is disproportionately impacted by contending accesses. This includes cases where contenders can effectively lock a resource for an extended or unbounded amount of time, or change the information stored in the resource in such a way that it needs to be obtained from elsewhere. Problems of cache thrashing [36], cache coherence [17], and cache miss status handling registers [41] can all cause effectively unbounded interference, and need to be eliminated from systems aimed at providing real-time predictability.

Section 3 introduces schedulability analysis for the MRSS task model, considering task sets scheduled according to partitioned fixed priority preemptive scheduling (pFPPS) and partitioned fixed priority non-preemptive scheduling (pFPNS) policies[2]. Two types of schedulability test are derived: (i) *context-dependent* tests that make use of information about the co-running tasks on the other cores, and (ii) *context-independent* tests that use only information about the tasks running on the same core. The latter are less precise, but *fully composable*, meaning that if the tasks on one core are changed, then only those tasks need have their schedulability re-assessed; task schedulability on the other cores is unaffected. *Composability* is an important issue for industry, particularly when different companies or departments are responsible for the sub-systems running on different cores.

In systems that use fixed priority scheduling, appropriate priority assignment is a crucial aspect of achieving a schedulable system [14]. Section 4 investigates optimal priority assignment, proving that Deadline Monotonic [31] priority ordering is optimal for both the context-independent and the simpler context-dependent schedulability tests for pFPPS. Similarly, Audsley's optimal priority assignment algorithm [4] is proven to be applicable and optimal for the equivalent tests for pFPNS. The more complex and precise context-dependent tests are proven incompatible with Audsley's algorithm [4].

[2] The most commonly used real-time scheduling polices in industry practice [1].

Section 5, provides a systematic evaluation of the effectiveness of the schedulability tests derived in Section 3. The results of this evaluation follow the dominance relationships demonstrated earlier, indicating the superiority of the more complex context-dependent schedulability tests, while also highlighting the additional contention that adding further cores brings. Section 6 concludes with a summary and directions for future work.

The appendix presents the findings from a case study examining 24 tasks from a Rolls-Royce aero-engine control system. These tasks were assessed using measurement-based timing analysis to obtain broad-brush estimates of their stand-alone WCETs, as well as characterizing their resource stress and resource sensitivity parameters. The purpose of the case study was not to try to determine definitive values for these parameters, in itself a challenging research problem, rather our aim was to obtain proof-of-concept data to act as an exemplar underpinning the MRSS task model and its analysis.

## 1.3 Related Work

Prior publications that relate to the research presented in this paper include work on micro-benchmarks [36, 16, 33, 24, 37] that can be used to stress resources in multi-core systems, and work on the integration of interference effects into schedulability analysis. Many of the latter papers are summarized in Section 4 of the survey [32] by Maiza et al. Unlike the analysis presented in this paper, which uses a generic task model that is applicable to many different types of interference and a variety of different shared hardware resources, most of these prior works focus on the details of one or more specific hardware resources. They require detailed information about the resource arbitration policy used, the number of resource accesses made by each task, and in some cases the timing of those accesses. By contrast, this paper takes a more abstract, but nonetheless valid view, that interference can be modeled in terms of its execution time impact via resource sensitivity and resource stress parameters for each task. This approach requires less detail about the resource behavior, and is more amenable to practical use, since it can still be used when full details of shared resource behavior are not available from the hardware vendor.

Early work on the integration of interference effects into schedulability analysis by Schliecker et al. [39] used arrival curves to model the resource accesses of each task, and hence how resource access delays due to contention impact upon task response times. Schliecker's work focused on contention over the memory bus. Further work in this area by Schranzhofer et al. [40], Pellizzoni et al. [35], Giannopoulou et al. [19], and Lampka et al. [28] used the superblock model that divides each task into a sequence of blocks, and uses information about the number of resource accesses within different phases of these blocks.

Dasari et al. [9] used a request function to model the maximum number of resource accesses from each task in a given time interval, and integrated this request function into response time analysis. Kim et al. [26] and Yun et al. [42] provided a detailed analysis of contention caused by DRAM accesses, accounting for access scheduling and variations in latencies due to differing states e.g. open and closed rows. The delays due to contention were then integrated into response time analysis. Altmeyer et al. [2, 10] introduced a multi-core response time analysis framework, aimed at combining the demands that tasks place on difference types of resources (e.g. CPU, memory bus, and DRAM) with the resource supply provided by those hardware resources. The resulting explicit interference was then integrated directly into response time analysis. Rihani et al. [38] built on this framework, using it to analyze complex bus arbitration policies on a many-core processor. Huang et al. [23] and Cheng et al. [8] leveraged the symmetry between processing and resource access, viewing tasks as executing and then suspending execution while accessing a shared resource. Using this suspension model in the schedulability analysis, they obtained results that were broadly comparable to those of Altmeyer et al. [2].

Paolieri et al. [34] proposed using a WCET-matrix and WCET-sensitivity values to characterize the variation in task execution times in different execution environments (e.g. with different numbers of contending cores, and different cache partition sizes). This information was then used to determine efficient task partitioning and task allocation strategies. Andersson et al. [3] presented a schedulability test where tasks have different execution times dependent on their co-runners. Here, tasks are represented by a sequence of segments, each of which has execution requirements and co-runner slowdown factors with respect to sets of other segments that could execute in parallel with it. The schedulability test involves solving a linear program to bound the longest response time given the possible ways in which different segments could execute in parallel and the slowdown in execution that this would entail. The method has significant scalability issues that effectively limit the total number of tasks it can handle to approximately 32 tasks on a 4 core system (i.e. 8 tasks per core).

### 1.4 Inspiration

The research presented in this paper was inspired by the desire to combine a practical approach to characterizing contention via micro-benchmarks and measurement-based techniques with a generic form of schedulability analysis that can be applied to a wide range of homogeneous multi-core systems with different types of shared hardware resources. The aim being to provide an effective form of timing verification that, while retaining the traditional two-step approach, is able to avoid undue pessimism by accounting for interference over long time intervals equating to task response times rather than short time intervals equating to task execution times. With industry practice in mind, the schedulability analysis derived includes context-dependent (non-composable), context-independent (fully composable), and partially composable schedulability tests. The overall method enables task timing behavior on multi-cores to be assessed without necessitating recourse to detailed information about the hardware behavior, something that most chip vendors do not make publicly available.

## 2 System Model and Assumptions

We assume a multi-core system with $m$ homogeneous cores that executes tasks under either partitioned fixed priority preemptive (pFPPS) or partitioned fixed priority non-preemptive (pFPNS) scheduling. With partitioning, tasks are assigned to a specific core and do not migrate. The tasks are assumed to be independent, but may access a set of shared hardware resources $r \in H$, thus causing interference on the execution of tasks on other cores via cross-core contention. We omit from consideration the effects of resource contention between tasks on the same core, since they are executed sequentially rather than in parallel. We assume that appropriate techniques are used to avoid substantial preemption effects when preemptive scheduling is employed, for example cache partitioning can be used to eliminate cache-related preemption delays. The costs of scheduling decisions and any context switching are assumed to be subsumed into the task execution times. Each task $\tau_i$ is characterised by: the minimum inter-arrival time or period between releases of its jobs, $T_i$, its relative deadline, $D_i$, and its WCET, $C_i$, when executing stand-alone, i.e. with no co-runners. All task deadlines are assumed to be *constrained* i.e. $D_i \le T_i$.

Further aspects of the model are based on the concept of *resource sensitive contenders* and *resource stressing contenders*. A resource stressing contender maximizes the stress on a resource $r$ by repeatedly making accesses to it that cause the most contention. Hence, running a resource stressing contender in parallel with a task creates the maximum increase in execution time for the task due to contention over resource $r$ from any single co-runner. A resource sensitive contender for a resource $r$, suffers the maximum possible interference by repeatedly making accesses to the resource that suffer from the most contention. Hence, running a resource sensitive contender in parallel with a task creates the maximum increase in execution time for any single co-running contender due to contention over resource $r$ from the task. Note, resource stressing and resource sensitive contenders for a given resource are not necessarily one and the same.

Each task is further characterised by its *resource sensitivity* $X_i^r$ and *resource stress* $Y_i^r$ for each shared hardware resource $r \in H$. $X_i^r$ captures the increase in execution time of task $\tau_i$ (from $C_i$ to $C_i + X_i^r$) when it is executed in parallel with a resource stressing contender for resource $r$. Thus $X_i^r$ models how much task $\tau_i$ behaves like a resource sensitive contender. Similarly, $Y_i^r$ captures the increase in execution time of a resource sensitive contender (from $C$ to $C + Y_i^r$) for resource $r$, when it is executed in parallel with task $\tau_i$. Hence $Y_i^r$ models how much task $\tau_i$ behaves like a resource stressing contender. With this model, the execution time of a task $\tau_i$ running on one core, subject to interference via shared hardware resource $r$ from task $\tau_k$ running in parallel on another core, is increased by at most $\min(X_i^r, Y_k^r)$ i.e. from $C_i$ to $C_i + \min(X_i^r, Y_k^r)$. The notation $\Gamma_x$ is used to denote the set of tasks that execute on the same core (with index $x$) as the task of interest $\tau_i$. Similarly, $\Gamma_y$ is used to denote the set of tasks that execute on some other core (with index $y$). Each task $\tau_i$ is assumed to have a unique priority. $hp(i)$ (resp. $lp(i)$) is used to denote the set of tasks with higher (resp. lower) priority than task $\tau_i$. Similarly, $hep(i)$ (resp. $lep(i)$) is used to denote the set of tasks with higher (resp. lower) than or equal priority to task $\tau_i$.

The schedulability tests introduced in this paper are named using the following convention: **Cp*Sched-m-X***, where **C** indicates a contention-based test for **p** partitioned scheduling, using scheduling policy ***Sched***, which is either **FPPS** or **FPNS**. The test is applicable to systems with ***m*** cores, and makes use of information ***X***, which is either **D** or **R** meaning the deadlines or the response times of the tasks on other cores, or **fc** meaning fully composable, i.e. the test does not rely on any information about the tasks running on the other cores.

The MRSS task model assumes that the resource sensitivity $X_i^r$ and resource stress $Y_i^r$ parameters for each task $\tau_i$ are provided by timing analysis. Obtaining precise bounds for these parameters is a challenging timing analysis problem that is beyond the scope of this paper; nevertheless, below we give a brief overview of how such values could be estimated.

Using measurement-based timing analysis techniques, the resource sensitivity $X_i^r$ can be obtained by capturing the maximum difference between the execution time of task $\tau_i$ when it runs in parallel with a resource stressing contender, and the corresponding execution time when it runs stand-alone, assuming that the same inputs and initial state are used in each case. Similarly, the resource stress $Y_i^r$ can be obtained by capturing the maximum difference between the execution time of a resource sensitive contender when it runs in parallel with task $\tau_i$, and the corresponding execution time of the contender when it runs stand-alone. As with measurement-based WCET estimation, such an approach needs to explore a representative set of inputs and initial states in order to obtain valid estimates. Further, resource stressing and resource sensitive contenders need to be carefully designed to meet their requirements in terms of creating/suffering the maximum amount of interference via contention over the resource [24]. Bounds on resource sensitivity and resource stress can also be obtained via static timing analysis. Static analysis first needs to compute an upper bound on the maximum number of accesses $A_i^r$ that task $\tau_i$ can make to the resource. The resource sensitivity $X_i^r$ can then be computed by determining the maximum increase in the execution time of task $\tau_i$ assuming that $A_i^r$ accesses are made in contention with an arbitrary number of accesses emanating from one other core. Similarly, the resource stress $Y_i^r$ equates to the maximum increase in the execution time of any arbitrary resource sensitive contender, due to contention over the resource caused by $A_i^r$ accesses emanating from one other core.

The schedulability analysis presented in Section 3 assumes that the total interference occurring via multiple different resources can be upper bounded by the sum of the interference occurring via each of those resources individually. This assumption can reasonably be expected to hold provided that the resource contention is independent. In other words, that contention over one resource does not create additional contention over another resource. An example that breaks this assumption occurs with a cache that is shared between cores. In this case, cache thrashing [36] can result in additional accesses to main memory, and hence further contention and interference over that disparate resource. Cache partitioning (per core) would be an effective way of addressing this issue, thus improving timing predictability.

The analysis also assumes that the total interference occurring due to co-running tasks on multiple other cores can be upper bounded by the sum of the interference occurring due to co-running tasks on each of those cores individually. This assumption can reasonably be expected to hold provided that there are no discontinuities in the amount of interference that can occur that can be triggered by co-running tasks on a multiple cores, but not by co-runners on just one core. An example that breaks this assumption occurs with cache miss status handling registers (MSHR) [41]. In this case, contention from tasks on multiple cores can exhaust all of the available MSHRs, resulting in substantial blocking delays. Depending on the local memory level parallelism, utilizing all of the MSHRs is typically not be possible with just one contending core. Increasing the number of MSHRs, or reducing the local memory level parallelism, such that contention from all $m$ cores cannot exhaust the set of MSHRs, are effective ways of addressing this problem [41] and restoring timing predictability. To validate the use of the analysis given in Section 3, each of the above assumptions needs to be assessed for the hardware architecture considered.

# 3 Schedulability Analysis

In this section, we introduce schedulability tests for the MRSS task model, assuming partitioned fixed priority preemptive scheduling (pFPPS) (Section 3.1), and partitioned fixed priority non-preemptive scheduling (pFPNS) (Section 3.2). In Section 3.3 we consider *composability* and derive context-independent schedulability tests for both pFPPS and pFPNS. The dominance relationships between the various tests are derived in Section 3.4.

## 3.1 pFPPS Schedulability Analysis

In the absence of any interference via shared hardware resources, the worst-case response time of task $\tau_i$ under pFPPS is given via standard response time analysis [25, 5]:

$$R_i \;=\; C_i \;+ \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1}$$

Adding cross-core interference considering each resource $r \in H$, we may compute the worst-case response time as follows:

$$R_i \;=\; C_i \;+ \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \sum_{r \in H} I_i^r(R_i) \tag{2}$$

where $I_i^r(R_i)$ is an upper bound on the interference that may occur within the response time of task $\tau_i$, via shared hardware resource $r$, due to tasks executing on the other cores.

The interference term $I_i^r(R_i)$ depends on: (i) the total resource sensitivity for resource $r$, denoted by $S_i^r(R_i, x)$, for the tasks executing on the same core $x$ as task $\tau_i$ within its response time $R_i$; and (ii) the total resource stress on resource $r$, denoted by $E_i^r(R_i, y)$, that can be produced by tasks executing on each of the other cores $y$ within an interval of length $R_i$. The total resource sensitivity $S_i^r(R_i, x)$ is computed based on the jobs that may execute within the worst-case response time of task $\tau_i$, hence with reference to (1) we have:

$$S_i^r(R_i, x) \;=\; X_i^r \;+ \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil X_j^r \tag{3}$$

The total resource stress $E_i^r(R_i, y)$ due to tasks that execute on another core $y$ in the interval $R_i$ can be upper bounded as follows. Here, unlike in (3), the worst-case does not occur when these tasks are released synchronously, but rather when the resource contention occurs as late as possible for one job of a task, and then as early as possible for subsequent jobs. Further, tasks of any priority can cause interference when executing on other cores. Thus we have:

$$E_i^r(R_i, y) \;=\; \sum_{j \in \Gamma_y} \left\lceil \frac{R_i + D_j}{T_j} \right\rceil Y_j^r \tag{4}$$

The analysis in (4) does not make any assumptions about how long task $\tau_j$ needs to execute in order to cause an increase in execution time of up to $Y_j^r$ in a task running on another core. In particular, there is no assumption that task $\tau_j$ needs to run for at least $Y_j^r$, since $Y_j^r$ is a measure of the maximum increase in execution time of another task due to contention from task $\tau_j$, not a measure of the time for which task $\tau_j$ needs to execute to cause that contention. Assuming that the execution causing contention can occur instantaneously, as is done in (4), is potentially pessimistic; however, it ensures that the analysis is sound even when there is considerable asymmetry in the (small) execution time required to stress a resource and the (large) increase in execution time of another task, which is sensitive to that resource stress. Since $X_k^r$ represents the maximum sensitivity of a task $\tau_k$ when subject to continuous interference via resource $r$ from a maximally resource stressing contender on one single other core, the maximum interference from other cores that can impact the response time of task $\tau_i$ via resource $r$ can be upper bounded by:

$$I_i^r(R_i) = \sum_{\forall y \neq x} \min(E_i^r(R_i, y), S_i^r(R_i, x)) \tag{5}$$

This is the case, since the maximum interference due to contention from each core $y$ cannot exceed the total resource stress $E_i^r(R_i, y)$ emanating from that core within a time $R_i$.

We refer to the schedulability test given by (2), (3), (4), and (5) as the **CpFPPS-*m*-D test**, since this test uses information about the *deadlines* of the tasks running on other cores.

A more precise analysis may be obtained by substituting $R_j$ for $D_j$ in (4) as follows, since a schedulable job of task $\tau_j$ cannot execute beyond its worst-case response time.

$$E_i^r(R_i, y) \;=\; \sum_{j \in \Gamma_y} \left\lceil \frac{R_i + R_j}{T_j} \right\rceil Y_j^r \tag{6}$$

Using this formulation, the response times of the tasks become interdependent. This problem can be solved via fixed point iteration. Here, an outer iteration starts with $R_i = C_i$, $R_j = C_j$ etc. for all tasks in the system, and repeatedly computes the response times for all tasks on all cores. This is done using the $R_j$ values in the right hand side of (6) from the previous round, until all response times either converge (i.e. are unchanged from the previous round) or one of them exceeds the associated deadline. Since $E_i^r(R_i, y)$ in (6) is a monotonically non-decreasing function of each $R_j$, then on each round, each $R_j$ value can only increase or remain the same, it cannot decrease. Thus, the outer fixed point iteration is guaranteed to either converge giving the set of schedulable $R_i \le D_i$ for all tasks in the system, or to result in some $R_i > D_i$, in which case that task and the system as a whole is unschedulable. We refer to the schedulability test given by (2), (3), (5), and (6) as the **CpFPPS-*m*-R test**, since it uses information about the *response times* of the tasks running on the other cores.

## 3.2 pFPNS Schedulability Analysis

In the absence of any cross-core contention and interference via shared hardware resources, the worst-case response time of task $\tau_i$ under pFPNS can be upper bounded via a sufficient response time analysis [13]:

$$R_i \ = \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)}(C_k) \ + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left( \left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) C_j + \ C_i \tag{7}$$

Here, we have reformulated the sufficient analysis for FPNS [13] into a single equation. The changes involve compacting the blocking term (max()), and bringing the execution time $C_i$ of the task under analysis into the equation. To compensate for the latter, the time interval in which higher priority tasks can execute is changed to $(R_i - C_i)$. This excludes the time at the end of the interval when task $\tau_i$ is executing non-preemptively. We also use a $\lfloor\ \rfloor + 1$ formulation rather than $\lceil\ \rceil$ to avoid the need for a term equal to the time unit granularity.

Similar to the case for pFPPS in (2), adding cross-core interference considering each resource $r \in H$, we may compute an upper bound on the worst-case response time as follows:

$$R_i \ = \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)}(C_k) \ + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left( \left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) C_j + \ C_i \ + \sum_{r \in H} I_i^r(R_i) \tag{8}$$

where $I_i^r(R_i)$ is an upper bound on the interference that may occur within the response time of task $\tau_i$, via shared hardware resource $r$, due to tasks executing on other cores. Here, we make the sound, but potentially pessimistic, assumption that even though the execution time of task $\tau_i$ may be increased to more than $C_i$ due to contention, only during the final $C_i$ time units of the task's response time are other tasks on core $x$ precluded from executing (i.e. we continue to use $(R_i - C_i)$ in the $\lfloor\ \rfloor$ function). Further, we use $R_i$ in the final term, since cross-core contention still occurs during non-preemptive execution.

The interference term $I_i^r(R_i)$ depends on: (i) the total resource sensitivity for resource $r$, denoted by $S_i^r(R_i, x)$, for the tasks executing on the same core $x$ as task $\tau_i$ within its response time $R_i$; and (ii) the total resource stress on resource $r$, denoted by $E_i^r(R_i, y)$, that can be produced by tasks executing on each of the other cores $y$ within an interval of length $R_i$. The total resource sensitivity $S_i^r(R_i, x)$ is computed based on the jobs that may execute within the worst-case response time of task $\tau_i$, hence with reference to (7) we have:

$$S_i^r(R_i, x) \ = \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)}(X_k^r) \ + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left( \left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) X_j^r + \ X_i^r \tag{9}$$

The two equations (4) and (6) for the total resource stress $E_i^r(R_i, y)$ due to tasks that execute on another core $y$ in the interval $R_i$ depend only on the tasks parameters and response times, but not the scheduling policy per se. Thus by redefining $S_i^r(R_i, x)$ according to (9) for the non-preemptive case, we obtain the following pFPNS schedulability tests for the MRSS task model.

The **CpFPNS-$m$-D** test given by (8), (9), (4), and (5) makes use of the *deadlines* of the tasks running on the other cores.

The **CpFPNS-$m$-R** test given by (8), (9), (6), and (5) makes use of the *response times* of the tasks running on the other cores.

## 3.3 Composability

The schedulability analyses derived in Sections 3.1 and 3.2 make use of information about the resource contention due to tasks executing on other cores. In other words, these analyses requires that the resource stress ($Y_j^r$) values are known for all tasks executing on the other cores, as well as their other parameters i.e. $T_j$, $D_j$, $R_j$. While this results in tighter response time bounds, it also means that the analyses are not *fully composable*, since the schedulability of the tasks running on one core depend on the parameters of the tasks running on the other cores. A *fully composable* analysis can, however, be obtained by redefining (5) as follows:

$$I_i^r(R_i) = \sum_{\forall y \neq x} S_i^r(R_i, x) = (m-1) \cdot S_i^r(R_i, x) \tag{10}$$

This equates to assuming a worst-case scenario of resource stressing contenders for each resource $r$ running on every core. This may be pessimistic on two counts: Firstly, the resource stressing contenders may cause significantly more interference than the tasks actually running on the other cores, and secondly, with more than one resource it may not be possible to maximally stress all resources simultaneously.

Using (10) results in *fully composable* context-independent schedulability tests. These tests are able to check the schedulability of task sets on each of the $m$ cores in a system, without needing to know any of the parameters of the tasks on the other cores. We refer to the schedulability test given by (2), (3), and (10) as the **CpFPPS-$m$-fc** test. Similarly, we refer to the schedulability test given by (8), (9), and (10) as the **CpFPNS-$m$-fc** test.

Finally, an intermediate *partially composable* analysis can be provided if resource access regulation mechanisms or budgets are employed to limit the amount of contention emanating from each core. Let $F_i^r(t, y)$ be the maximum increase in execution time of a resource sensitive contender on another core that can occur due to contention over resource $r$ caused by a resource stressing contender running on core $y$ for a time period of $t$, subject to resource regulation. Partially composable analysis can be obtained by redefining (5) as follows:

$$I_i^r(R_i) = \sum_{\forall y \neq x} \min(F_i^r(R_i, y), S_i^r(R_i, x)) \tag{11}$$

Note, this analysis only holds if the resource regulation on each core $y$ does not actually limit the accesses to each resource $r$ made by tasks on that core over any time interval. Provided that is guaranteed, no actual runtime enforcement is necessary, the budget function $F_i^r(t, y)$ simply acts as an intermediate value that permits a separation of concerns and composition.

## 3.4 Dominance Relations

A schedulability test $S$ is said to *dominate* another test $Z$ for a given task model and scheduling algorithm, if every task set that is deemed schedulable according to test $Z$ is also deemed schedulable by test $S$, and there exists some task sets that are schedulable according to test $S$, but not according to test $Z$.

Comparing the definitions of $E_i^r(R_i, y)$ given by (6) for the **CpFPPS-$m$-R** and **CpFPNS-$m$-R** tests and by (4) for the **CpFPPS-$m$-D** and **CpFPNS-$m$-D** tests, it is evident that each of the former tests deems schedulable all task sets that are schedulable according to the corresponding latter test. This is the case, since in any schedulable system, the response time of a task is no greater than its deadline ($R_j \leq D_j$), and hence the $E_i^r(R_i, y)$ term for the former tests, given by (6), is less then or equal to the equivalent term, given by (4), for the latter tests. Further, it is easy to see that there exist task sets that are schedulable according to the each of the former tests, but not according to the corresponding latter test due to a larger contention contribution emanating from the larger $E_i^r(R_i, y)$ term. Hence the **CpFPPS-*m*-R** test dominates the **CpFPPS-*m*-D** test, and the **CpFPNS-*m*-R** test dominates the **CpFPNS-*m*-D** test.

Comparing the definitions of $I_i^r(R_i)$ given by (5) for the **CpFPPS-*m*-D CpFPNS-*m*-D** tests and by (10) for the **CpFPPS-*m*-fc** and **CpFPNS-*m*-fc** tests, it is evident that the former tests deems schedulable all task sets that are schedulable according to the corresponding latter test. Further, it is easy to see that there exist task sets that are schedulable according to the each of the former tests, but not according to the corresponding latter test due to a larger contention contribution emanating from the larger $I_i^r(R_i)$ term. Hence the **CpFPPS-*m*-D** test dominates the **CpFPPS-*m*-fc** test, and the **CpFPNS-*m*-D** test dominates the **CpFPNS-*m*-fc** test.

As dominance is transitive, we have: **CpFPPS-*m*-R** → **CpFPPS-*m*-D** → **CpFPPS-*m*-fc** and **CpFPNS-*m*-R** → **CpFPNS-*m*-D** → **CpFPNS-*m*-fc** where $S \rightarrow Z$ indicates that test $S$ dominates test $Z$.

Finally, comparing a system of $m$ cores to one with $m + 1$ cores, where in each case the first $m$ cores execute exactly the tasks, and the $m + 1$ core system has extra tasks that executes on core $m + 1$, then there is a dominance relationship between the systems as analysed by any of the schedulability tests. In other words, adding a core and the contention that it brings cannot improve schedulability for the tasks running on the existing cores, but may make their schedulability worse. Schedulability for $m$ cores thus dominates that for $m + 1$ cores with added tasks: **Cp*Sched*-m-*X*** → **Cp*Sched*-$(m + 1)$-*X***

## 4 Priority Assignment

To maximize schedulability it is necessary to assign task priorities in an optimal way [14]. This section considers optimal priority assignment for the schedulability tests introduced in Section 3.

### 4.1 pFPPS Priority Assignment

Leung and Whitehead [31] showed that Deadline Monotonic Priority Ordering (DMPO) is optimal for constrained-deadline task sets with parameters $(C, D, T)$ under fixed priority preemptive scheduling. We observe that this result also holds for constrained-deadline MRSS task sets compliant with model described in Section 2 and analysed according to the **CpFPPS-*m*-fc** test introduced in Section 3.3. This is because that formulation can be re-arranged to match the basic response time analysis (1), with the execution time of each task $\tau_k$ increased by $\sum_{r \in H}(m-1)X_k^r$. DMPO is also optimal for constrained-deadline MRSS task sets analysed according to the **CpFPPS-*m*-D** test, introduced in Section 3.1. Proof is given below using the standard apparatus for proving the optimality of such priority orderings, as described in section IV of [14]. This proof technique is applicable in cases where task priorities can be defined directly from fixed task parameters, for example periods and deadlines. To show that a priority assignment policy $P$ (i.e. DMPO) is optimal, it suffices to prove that any task set that is schedulable according to the schedulability test considered using some priority assignment policy $Q$ is also schedulable using priority ordering $P$. Proof is obtained by transforming priority ordering $Q$ into priority ordering $P$, while ensuring that no tasks become unschedulable during the transformation. The proof proceeds by induction.

▶ **Theorem 1.** *Deadline Monotonic Priority Ordering is optimal for constrained-deadline MRSS task sets compliant with the model described in Section 2 and analysed according to the* ***CpFPPS-m-D*** *test introduced in Section 3.1.*

**Proof.** *Base case:* The task set is schedulable with priority order $Q = Q^k$, where $k$ is the iteration count.

*Inductive step:* We select a pair of tasks that are at adjacent priorities $i$ and $j$ where $j = i + 1$ in priority ordering $Q^k$, but out of Deadline Monotonic relative priority order. Let these tasks be $\tau_A$ and $\tau_B$, with $\tau_A$ having the higher priority in $Q^k$. Note that $D_A > D_B$ as the tasks are out of Deadline Monotonic relative order. Let $i$ be the priority of task $\tau_A$ in $Q^k$ and $j$ be the priority of task $\tau_B$. We need to prove that all of the tasks remain schedulable with priority order $Q^{k-1}$, which switches the priorities of these two tasks. There are four groups of tasks to consider:

$hp(i)$: tasks in this set have higher priorities than both $\tau_A$ and $\tau_B$ in both $Q^k$ and $Q^{k-1}$. Since the schedulability of these tasks is unaffected by the relative priority ordering of $\tau_A$ and $\tau_B$, they remain schedulable in $Q^{k-1}$.

$\tau_A$: Let $w = R_B$ be the response time of task $\tau_B$ in priority order $Q^k$. Since task $\tau_B$ is schedulable in $Q^k$, we have $w = R_B \leq D_B < D_A \leq T_A$, hence in (2), the contribution from $\tau_A$ within the response time of $\tau_B$ is exactly one job (i.e. $C_A$), and there is also a contribution of $C_B$ from task $\tau_B$ itself. Considering interference, the total resource sensitivity $S^r_B(w, x)$ given by (3) depends only on the value $w$ and fixed parameters of the set of tasks with priorities higher than or equal to task $\tau_B$ in $Q^k$ that is $\tau_A$, $\tau_B$, and $hp(i)$. Further, the total resource stress $E^r_B(w, y)$ due to tasks executing on some other core $y$ depends only on the value of $w$ and the fixed parameters of the tasks executing on that core. It follows that the interference term $I^r_B(w)$ given by (5) and used in (2) depends only on the value of $w$ and the fixed parameters of the set of tasks $\tau_A$, $\tau_B$, and $hp(i)$, as well as the fixed parameters of the tasks executing on all other cores. Now consider the response time of task $\tau_A$ under priority order $Q^{k+1}$. This response time is $R_A = w$ , as there is exactly the same contribution from tasks $\tau_A$, $\tau_B$ and all the higher priority tasks, and further the interference due to resource contention is the same, in other words $I^r_B(w)$ for $Q^k$ equates to $I^r_A(w)$ for $Q^{k+1}$, since the value of $w$ is the same, and the set of tasks that this term is dependent upon is unchanged ($\tau_A$, $\tau_B$, and $hp(i)$ on core $x$, and all of the task on the other cores). Since $w < D_A$, task $\tau_A$ remains schedulable.

$\tau_B$: as the priority of $\tau_B$ has increased its response time is no greater in $Q^{k+1}$ than in $Q^k$. This is the case because the only change to the response time calculation for $\tau_B$ is the removal of the contribution from task $\tau_A$, and also the removal of its contribution to the total resource sensitivity, and hence from the interference term $I^r_B(w)$. Thus $\tau_B$ remains schedulable.

$lp(j)$ : tasks in this set have lower priorities than tasks $\tau_A$ and $\tau_B$ in both $Q^k$ and $Q^{k+1}$. Since the schedulability of these tasks is unaffected by the relative priority ordering of tasks $\tau_A$ and $\tau_B$, they remain schedulable.

All tasks therefore remain schedulable in $Q^{k+1}$.

At most $k = n(n-1)/2$ steps are required to transform priority ordering $Q$ into $P$ without any loss of schedulability ◀

Next, we consider optimal priority assignment with respect to the **CpFPPS-*m*-R** test introduced in Section 3.1. Davis and Burns proved in [12] that it is both sufficient and necessary to show that a schedulability test meets three simple conditions in order for Audsley's Optimal Priority Assignment (OPA) algorithm [4] algorithm to be applicable.

**Condition 1:** The schedulability of a task according to the test must be independent of the relative priority order of higher priority tasks.

**Condition 2:** The schedulability of a task according to the test must be independent of the relative priority order of lower priority tasks.

**Condition 3:** The schedulability of a task according to the test must not get worse if the task is moved up one place in the priority order (i.e. its priority is swapped with that of the task immediately above it in the priority order).

▶ **Theorem 2.** *The **CpFPPS-m-R** test, given in Section 3.1, is not compatible with Audsey's Optimal Priority Assignment (OPA) algorithm [4], and hence that algorithm cannot be used to obtain an optimal priority assignment with respect to the test.*

**Proof.** To prove non-compatibility, it suffices to show that any one of the three conditions set out in [12] and listed above is broken by the test. In this case, we show that Condition 1 does not hold. According to the **CpFPPS-*m*-R** test, the schedulability of a task $\tau_i$ on core $x$ can depend on the response time of task $\tau_j$ on a different core $y$ via $E_j^r(R_i, y)$ given by (6). In turn, the response time of task $\tau_j$ can depend on the response time of some higher priority task $\tau_k$ on the same core $x$ as task $\tau_i$ via $E_k^r(R_j, x)$ also given by (6). Since the response time of task $\tau_k$ depends on its relative priority order among those tasks with higher priority than task $\tau_i$, Condition 1 does not hold and therefore the **CpFPPS-*m*-R** test is not compatible with Audsley's OPA algorithm ◀

Although the **CpFPPS-*m*-R** test is not compatible with Audsley's OPA algorithm, the form of the test, with its dependence on the response times of other tasks, means that a back-tracking search, as proposed in [11], could potential be used to obtain a schedulable priority assignment without having to explore all possible priority orderings. The same applies to the **CpFPNS-*m*-R** test discussed in Section 4.2 below.

## 4.2 pFPNS Priority Assignment

George et al. [18] showed that Deadline Monotonic Priority Ordering (DMPO) is not optimal for constrained-deadline task sets with parameters $(C, D, T)$ under fixed priority non-preemptive scheduling, and proved that Audsley's algorithm [4] is able to provide an optimal priority ordering in this case. We observe that this result also holds for constrained-deadline MRSS task sets compliant with the model described in Section 2 and analysed according to the **CpFPNS-*m*-fc** test introduced in Section 3.3. This is the case because the formulation can be re-arranged to match the basic response time analysis (7), with the execution time of each task $\tau_k$ increased by $(m-1)X_k^r$. Audsley's algorithm [4] is also optimal with respect to the **CpFPNS-*m*-D** test, as proved below.

▶ **Theorem 3.** *Audsley's algorithm [4] is optimal for constrained-deadline MRSS task sets compliant with the model described in Section 2 and analysed according to the **CpFPNS-m-D** test introduced in Section 3.2.*

**Proof.** It suffices to show that the schedulability test meets the three conditions, given in [12] and set out in Section 4.1. With respect to **Condition 1** and **Condition 2**, inspection of (8) shows that the first two terms are dependent on the set of lower and equal priority tasks $lep(i)$ and the set of higher priority tasks $hp(i)$ respectively, but do not depend on the relative priority order of the tasks within those sets. Considering the fourth term in (8), $I_i^r(t)$ is given by (5). In the definition of $I_i^r(t)$, the total resource sensitivity $S_i^r(t, x)$ is given by (9), which is dependent on the set of tasks $lep(i)$ and the set of tasks $hp(i)$, but does not depend on the relative priority order of the tasks within those sets. Finally, the total resource contention $E_i^r(t, y)$ given by (4) has no dependence on the relative priority order of the tasks in the sets $hp(i)$ and $lep(i)$ (or $lp(i)$), thus **Condition 1** and **Condition 2** hold.

With respect to **Condition 3**, moving task $\tau_i$ up one place in the priority order is equivalent to moving another task $\tau_h$ that also executes on core $x$ from the set $hp(i)$ to the set $lep(i)$. Considering (8), such a change may increase the first term by no more than $C_h$, but is guaranteed to also reduce the second term by at least $C_h$. Further, with respect to the total resource sensitivity $S_i^r(t, x)$, given by (9), such a change may increase the first term by no more than $X_h^r$, but is guaranteed to also reduce the second term by at least $X_h^r$. There is no change to the total resource stress $E_i^r(t, y)$ given by (4). Hence the schedulability of task $\tau_i$ cannot get worse if the task is moved up one place in the priority order. ◀

Finally, we note that the **CpFPNS-*m*-R** test is not compatible with Audsley's OPA algorithm, since it breaks Condition 1, as proven below.

▶ **Theorem 4.** *The **CpFPNS-m-R** test given in Section 3.1, is not compatible with Audsey's Optimal Priority Assignment (OPA) algorithm [4], and hence that algorithm cannot be used to obtain an optimal priority assignment with respect to the test.*

**Proof.** Proof follows via exactly the same argument as given in the proof of Theorem 2 in Section 4.1, replacing the **CpFPPS-*m*-R** test with the **CpFPNS-*m*-R** test. ◀

## 5 Evaluation

In this section, we present an empirical evaluation of the schedulability tests introduced in Section 3 for MRSS task sets executing on a multi-core system, assuming a single hardware resource shared between all cores. (Note, multiple shared hardware resources resulting in the same total interference would have the same impact on schedulability, due to the summation terms in (2) and (8)). Experiments were performed for 1, 2, 3, and 4 cores[3], with the single core case considered for comparison purposes.

### 5.1 Task Set Parameter Generation

The task set parameters used in our experiments were generated as follows:

- Task utilizations ($U_i = C_i/T_i$) were generated using the Dirichlet-Rescale (DRS) algorithm [21] (open source Python software [20]) providing an unbiased distribution of utilization values that sum to the total utilization $U$ required.
- Task periods $T_i$ were generated according to a log-uniform distribution [15] with a factor of 100 difference between the minimum and maximum possible period. This represents a spread of task periods from 10ms to 1 second, as found in many real-time applications. (When considering non-preemptive scheduling, a factor of 10 difference was used, otherwise most task sets would not be schedulable).
- Task deadlines $D_i$ were set equal to their periods $T_i$.
- The stand-alone execution time of each task was given by: $C_i = U_i \cdot T_i$.

[3] The analysis scales to more than 4 cores; however, we limited consideration to this range, since 4 cores represents a typical cluster size beyond which sharing hardware resources can become a significant performance bottleneck.

- Task resource sensitivity values $X_i^r$ were determined as follows. The DRS algorithm was used to generate task resource sensitivity utilization values $V_i^r$, such that the total resource sensitivity utilization was $SF$ (the Sensitivity Factor, default $SF = 0.25$) times the total task utilization (i.e. $\sum_{\forall i} V_i^r = U \cdot SF$), and each individual task resource sensitivity utilization was upper bounded by the corresponding task utilization (i.e. $V_i^r \leq U_i$). Each task resource sensitivity value was then given by $X_i^r = V_i^r \cdot T_i$.
- Task resource stress values $Y_i^r$ were set to a fixed proportion of the corresponding resource sensitivity value $Y_i^r = X_i^r \cdot RF$, where $RF$ is the Stress Factor, default $RF = 0.5$.

The default value for the Sensitivity Factor ($SF = 0.25$) was set to approximately twice the average value (13.6%) obtained for the tasks in the industry case study described in the Appendix. This is justified since the case study considers a single shared hardware resource, whereas in practice contention would likely occur via multiple shared hardware resources, resulting in higher levels of interference. The default value for the Stress Factor ($RF = 0.5$) was set within the range found in the case study (0.23 to 1.58). Further, specific experiments were also used to evaluate performance over a wide range of values for these parameters.

## 5.2 Experiments

The experiments considered systems with 1, 2, 3, and 4 cores, with a different task set (generated according to the same parameters) assigned to each core. The per core task set utilization $U$ (shown on x-axis of the graphs) was varied from 0.05 to 0.95. For each utilization value examined, 1000 task sets were generated for each core considered, (100 in the case of experiments using the weighted schedulability measure [6]). The default cardinality of the task sets on each core was $n = 10$. In the experiments, a system was deemed schedulable if and only if the different task sets assigned to each of its $m$ cores were schedulable, i.e. if all $m \cdot n$ tasks in the system were schedulable. With a single core, there is no cross-core resource contention and hence no interference, and so task set schedulability can be determined via standard response time analysis. With multiple cores, contention and the resulting interference was modelled as described in Section 2. The experiments investigated the performance of the following schedulability tests for partitioned fixed priority preemptive and non-preemptive scheduling:

- **No-CpFPPS-*m***: The exact analysis of pFPPS [25, 5] with no contention, recapped in Section 3.1, and given by (1).
- **CpFPPS-*m*-R**: The response time based analysis of pFPPS with contention, introduced in Section 3.1, and given by (2), (3), (5), and (6).
- **CpFPPS-*m*-D**: The deadline based analysis of pFPPS with contention, introduced in Section 3.1, and given by (2), (3), (4), and (5).
- **CpFPPS-*m*-fc**: The fully composable analysis of pFPPS with contention, introduced in Section 3.3, and given by (2), (3), and (10).
- **No-CpFPNS-*m***: The sufficient analysis of pFPNS [13] with no contention, recapped in Section 3.2, and given by (7)).
- **CpFPNS-*m*-R**: The response time based analysis of pFPNS with contention, introduced in Section 3.2, and given by (8), (9), (6), and (5).
- **CpFPNS-*m*-D**: The deadline based analysis of pFPNS with contention, introduced in Section 3.2, and given by (8), (9), (4), and (5).
- **CpFPNS-*m*-fc**: The fully composable analysis of pFPNS with contention, introduced in Section 3.3, and given by (8), (9), and (10).

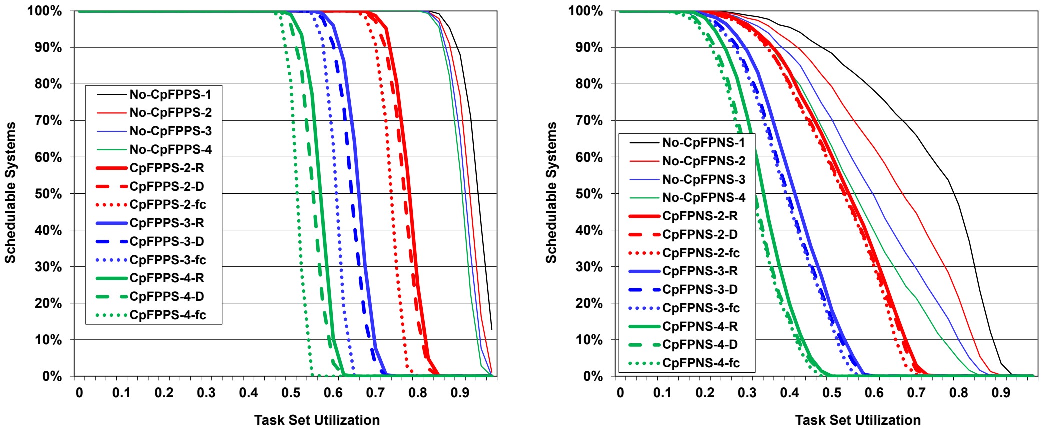
For consistency of comparison, Deadline Monotonic Priority Ordering (DMPO) [31] was used to assign priorities to tasks on the individual cores. As shown in Section 4, DMPO is optimal with respect to the **No-CpFPPS-$m$**, **CpFPPS-$m$-fc**, and **CpFPPS-$m$-D** tests, but only a heuristic policy with respect to the **CpFPPS-$m$-R** test and the tests for fixed priority non-preemptive scheduling.

Note, the results for the fully composable analyses (tests **CpFPPS-$m$-fc** and **CpFPNS-$m$-fc**) equate to the performance obtained via the use of resource sensitivity information only, as outlined in prior works [36, 16, 33, 24].

## 5.3 Results

In the first experiment, we compared the performance of the various schedulability tests, assuming 1, 2, 3, and 4 cores, using the default parameters given in Section 5.1. The *Success Ratio*, i.e. the percentage of systems generated that were deemed schedulable, is shown for each of the pFPPS schedulability tests in Figure 1a, and for the pFPNS schedulability tests in Figure 1b. The dominance relationships between the tests, discussed in Section 3.4, are evidenced by the lines on the graphs. Note, schedulability depends on the number of cores even when contention is not taken into account. This is because for an $m$-core system the task sets on all $m$ cores have to be schedulable for the system to be deemed schedulable.

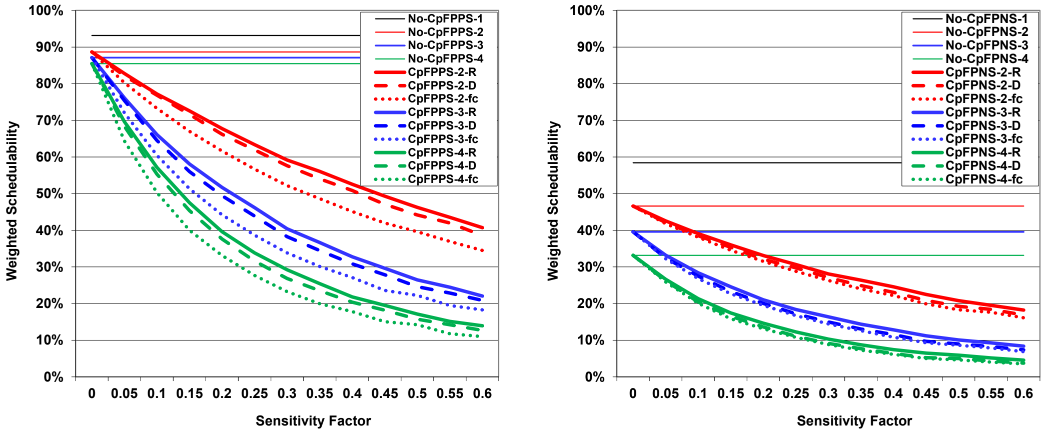
Observe, that the performance advantage that the context-independent tests have over their context-dependent counterparts is more pronounced with pFPPS than with pFPNS. The reason for this is that the increased response times due to the blocking factor with pFPNS mean that the critical task(s) (those that become unschedulable as utilization is increased) are much more likely to be medium or high priority tasks than is the case with pFPPS. For higher priority tasks, the balance between total resource sensitivity $S_i^r(R_i, x)$ and total resource stress $E_i^r(R_i, y)$ tends towards the latter being larger, since $E_i^r(R_i, y)$ includes a contribution from all of the tasks on core $y$, while $S_i^r(R_i, x)$ only includes a contribution from a single lower priority (blocking) task in the case of pFPNS, and no lower priority tasks at all in the case of pFPPS. When $E_i^r(R_i, y)$ exceeds $S_i^r(R_i, x)$ then the performance of the context-independent tests is reduced to that of their context-dependent counterparts.



**(a)** pFPPS.

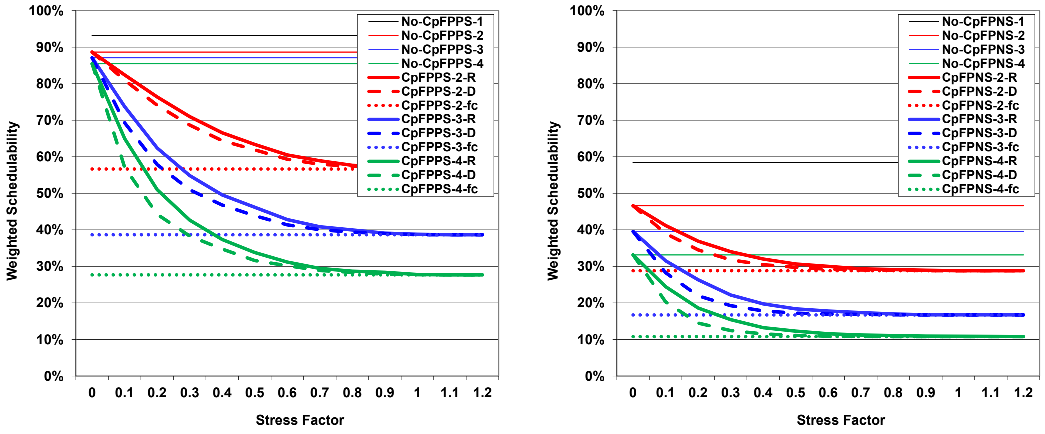
**(b)** pFPNS.

**Figure 1** Success Ratio: Varying task set utilization.



**(a)** pFPPS. **(b)** pFPNS.

**Figure 2** Weighted Schedulability: Varying Sensitivity Factor (SF).



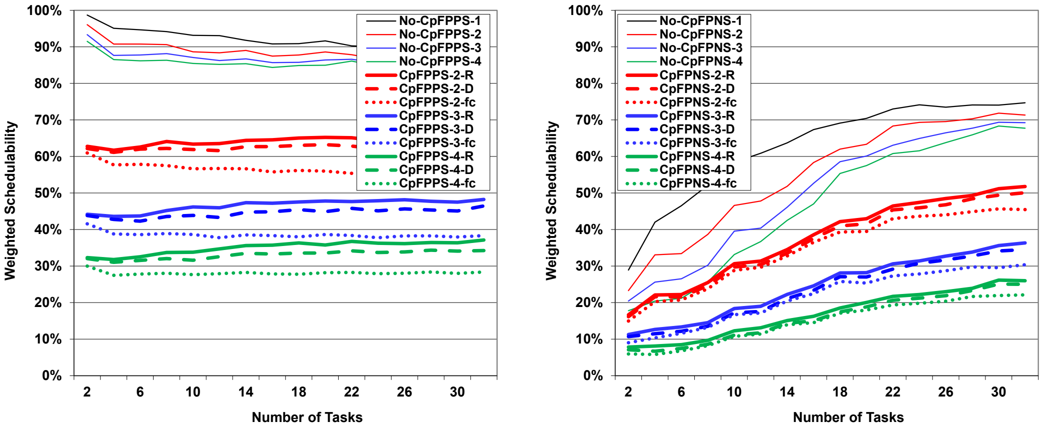
**(a)** pFPPS. **(b)** pFPNS.

**Figure 3** Weighted Schedulability: Varying Stress Factor (RF).

In the second set of experiments, we used the weighted schedulability measure [6] to assess schedulability test performance, while varying an additional parameter. In these experiments, the other parameters were set to their default values given in Section 5.1. In all of the weighted schedulability experiments the relative performance of the different tests follows the pattern illustrated in the first experiment, as dictated by the dominance relationships.

The results of varying the Sensitivity Factor $SF$ from 0.05 to 0.5 in steps of 0.05, are shown in Figure 2a for pFPPS, and Figure 2b for pFPNS. Recall that the Sensitivity Factor determines the ratio of the total resource sensitivity utilization to the total task utilization. As expected, increasing the Sensitivity Factor and hence the amount of interference that tasks can be subject to due to cross-core contention for resources results in a rapid decline in the weighted schedulability measure for all of the tests that take into account contention.

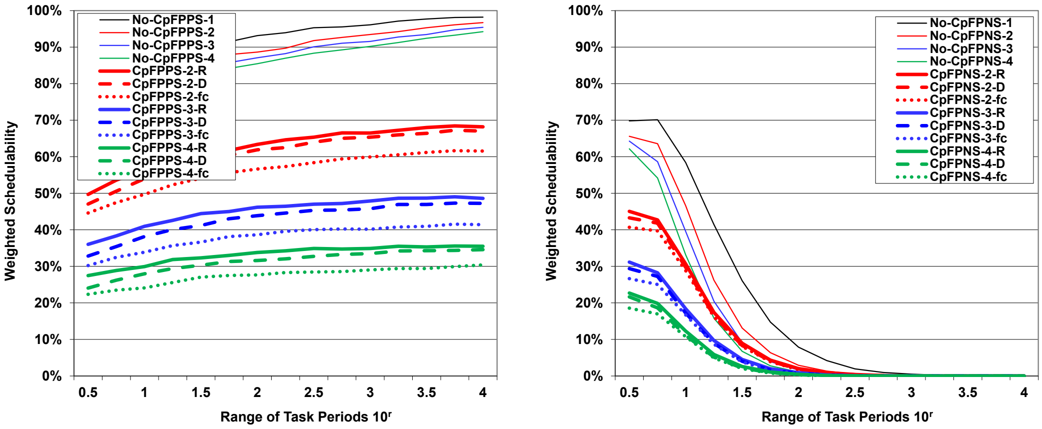
The results of varying the Stress Factor $RF$ from 0 to 1.2 in steps of 0.1 are shown in Figure 3a for pFPPS, and Figure 3b for pFPNS. Recall that the Stress Factor determines the ratio of the resource stress for each task to its resource sensitivity. Here, it is interesting to note that interference effective saturates once the Stress Factor reaches 1.0. By then, the



**(a)** pFPPS.

**(b)** pFPNS.

**Figure 4** Weighted Schedulability: Varying number of tasks in each task set.

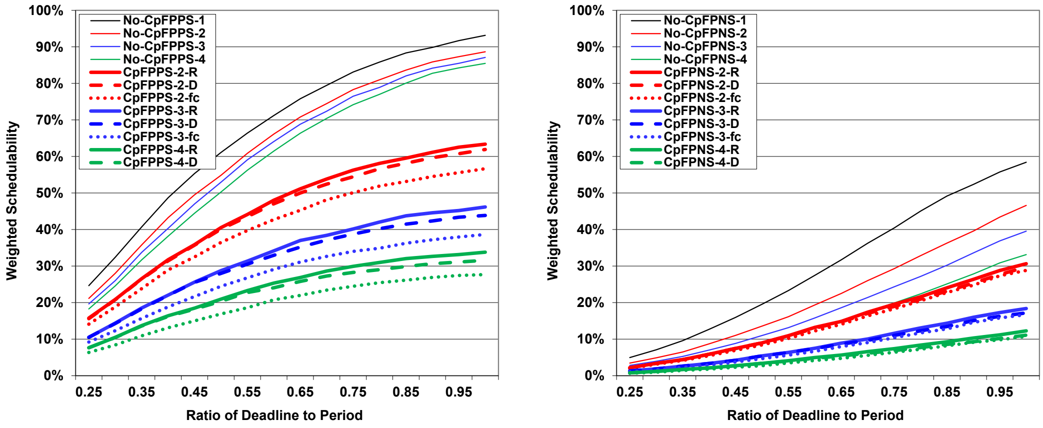


**(a)** pFPPS.

**(b)** pFPNS.

**Figure 5** Weighted Schedulability: Varying range of task periods.

total resource stress $E_i^r(t, y)$, given by (4) or (6), emanating from each additional core $y$ in a time interval $t$ tends to exceed the total resource sensitivity $S_i^r(t, x)$, given by (3), for core $x$ in that same time interval. Hence, for pFPPS the **CpFPPS-*m*-R** and **CpFPPS-*m*-D** tests reduce to exactly the same performance as the **CpFPPS-*m*-fc** test. Similarly, for pFPNS the **CpFPNS-*m*-R** and **CpFPNS-*m*-D** tests reduce to exactly the same performance as the **CpFPNS-*m*-fc** test. This is because the $\min(E_i^r(t, y), S_i^r(t, x))$ term in (5) ceases to reduce the value in the summation below $S_i^r(t, x)$. At the other extreme a Stress Factor $RF$ of zero means that $E_i^r(t, y) = 0$ whether computed via (4) or (6). For pFPPS, the **CpFPPS-*m*-R** and **CpFPPS-*m*-D** tests therefore have the same performance as the no contention **No-CpFPPS-*m*** test, and similarly for pFPNS the **CpFPNS-*m*-R** and **CpFPNS-*m*-D** tests have the same performance as the **No-CpFPNS-*m*** test. Between the two extremes, the smaller values of $E_i^r(t, y)$ given by (6) as opposed to (4) mean that the **CpFPPS-*m*-R** test outperforms the **CpFPPS-*m*-D** test, and similarly the **CpFPNS-*m*-R** test outperforms the **CpFPNS-*m*-D** test.



**(a)** pFPPS.

**(b)** pFPNS.

**Figure 6** Weighted Schedulability: Varying ratio of deadlines to periods.

The results of varying the cardinality of task sets running on each core from 2 to 32 in steps of 2 are shown in Figure 4a for pFPPS, and Figure 4b for pFPNS. In the preemptive case, task set cardinality typically has only a limited effect on schedulability test performance; however, in the non-preemptive case (Figure 4b), larger task sets equate to smaller execution times for each task and hence smaller blocking factors. Thus schedulability improves with increasing cardinality for all of the pFPNS schedulability tests. In the preemptive case (Figure 4a) the gap between the **CpFPPS-$m$-R** and **CpFPPS-$m$-D** tests and the **CpFPPS-$m$-fc** test increases with larger numbers of tasks. This is due to changes in the shape of the total resource stress function $E_i^r(t, y)$, which typically consists of many small steps when there are a large number of tasks, and fewer larger steps when there are a smaller number of tasks. As the function $E_i^r(t, y)$ is above the same gradient line in both cases, this difference acts to improve schedulability for the **CpFPPS-$m$-R** and **CpFPPS-$m$-D** tests at higher task set cardinality. The same effect is also evident in Figure 4a for the pFPNS schedulability tests.

The effects of varying the range of task periods (ratio of the max/min possible task period) from $10^{0.5} \approx 3$ to $10^4 = 10,000$ are shown in Figure 5a for pFPPS, and Figure 5b for pFPNS. As expected, increasing the range of task periods results in a gradual improvement in pFPPS schedulability test performance, a well-known effect with fixed priority preemptive scheduling. In contrast, with non-preemptive scheduling, once the range of task periods exceeds 100 (i.e. $r = 2$), hardly any task sets are schedulable. This happens because tasks with short periods (and deadlines) cannot tolerate being blocked by tasks with long periods and commensurate large execution times.

Finally, the results of varying task deadlines from 25% to 100% of the task's period are shown in Figure 6a for pFPPS, and Figure 6b for pFPNS. As expected, schedulability improves for all approaches as task deadlines are increased. Further, the performance advantage of the **CpFPPS-$m$-R** test over the **CpFPPS-$m$-D** test increases with increasing deadlines. This occurs because larger deadlines provide a more pessimistic approximation of response times for schedulable tasks, impacting the total resource stress as assumed by the **CpFPPS-$m$-D** test.

## 6 Conclusions

The main contributions of this paper are the Multi-core Resource Stress and Sensitivity (MRSS) task model and its accompanying schedulability analyses. The MRSS task model:

- Characterizes how much each task stresses shared hardware resources and how much it is sensitive to such resource stress.
- Provides a simple yet effective interface between timing analysis and schedulability analysis, facilitating a separation of concerns that retains the advantages of the traditional two-step approach to timing verification.
- Caters for a variety of different shared hardware resources in a way that is both generic and versatile.

The accompanying schedulability analyses:

- Provide efficient context-dependent and context independent schedulability tests for both fixed priority preemptive and fixed priority non-preemptive scheduling.
- Exhibit dominance relationships illustrating the trade-off between context independence and schedulability.
- Were proven compatible or incompatible with efficient optimal priority assignment algorithms.
- Were subject to a systematic evaluation illustrating their effectiveness across a wide range of parameter values.

In future, we aim to investigate task allocation strategies for partitioned fixed priority scheduling of MRSS tasks. Details of a preliminary case study that explores the resource stress and resource sensitivity of tasks from a Rolls-Royce aero-engine control system are given in the appendix. This case study provides an underpinning proof-of-concept for the MRSS task model.

## A Case Study

In this appendix, we present a preliminary case study that investigates the resource stress and resource sensitivity of tasks from a real-time industrial application. The purpose of this case study is not to try to determine definitive values for task WCETs, resource sensitivities and resource stresses, in itself a challenging research problem that is beyond the scope of this work. Rather our aim is to obtain proof-of-concept data to act as an exemplar underpinning the MRSS task model and its accompanying schedulability analysis.

The case study focuses on a set of 24 tasks from a Rolls-Royce aero engine control system or FADEC (Full Authority Digital Engine Controller). The industrial software was developed in SPARK-Ada and has been verified according to DO-178C standards (level A). The software was provided in an anonymized object code format, derived from that used in the case studies reported in [29] and [30]. The tasks have object code libraries ranging in size from 300 KBytes to 40 MBytes, including compiled in data structures, but not including any framework or Linux additions. The software was originally designed to run on a Rolls-Royce specific packaged processor that integrates a single core, memory, I/O, and tracing interfaces; however, for research purposes, it was ported to run on a Raspberry Pi 3B+ [30], along with a framework that facilitates taking timing measurements [7].

The Raspberry Pi 3B+ uses a Broadcom BCM2837 System-on-Chip with a quad-core ARM Cortex-A53 processor. It has a 16 KByte L1 data cache, 16 KByte L1 instruction cache, 512 KByte L2 shared cache, and 1 GByte of DDR2-DRAM. The L2 cache was, as is the default, configured for use as local memory for the GPU[4], and so was not available to the four CPUs. The experimental hardware set-up involved a cluster of Raspberry Pi 3B+s, configured to run at a clock frequency of 600MHz, so as to eliminate any possible disruption due to thermal throttling. The cluster was powered by specialized power rails to ensure a stable supply voltage and frequency. The Pi 3B+s used the Raspberry Pi OS Lite (updated on 01/25/2021) and the Linux Kernel 5.10.11-v7+. The `isolcpus` Linux option was used to minimize activity on the two cores used for the experiments. Timing measurements were obtained using a nanosecond clock, and cross-referenced against a 600MHz cycle counter. Prior to each run of a task, the L1 data and L1 instruction caches were flushed. Given that the L2 cache was not accessible to the CPUs, the case study focussed on the key shared hardware resource, main memory (DDR2-DRAM).

## A.1 Case Study Experiments

For each of the 24 tasks, we considered 10,000 randomly selected traces of execution. When a task was called for a specific trace, each of its input parameters was set to a random value based on the type (float, integer, or boolean) and the range of values permitted. The inputs were thus randomized, but nevertheless reproducible via the trace number, which controlled the random seed used. In the following, for brevity we use *trace* to mean a task with a specific set of input parameters corresponding to the trace number.

In **Experiment A.1**, for each trace we obtained the stand-alone execution time, the resource sensitivity, and the resource stress as measured against each of the three contenders described below. These values were obtained by: (i) running the trace stand alone, (ii) running the trace in parallel with the contender, (iii) running the contender stand alone. In (i) and (ii) the execution time of the trace was recorded. In addition, in (ii) the number of times $L$ that the contender looped while the trace was running was recorded, along with the execution time of the contender for that number of loops. Finally, in (iii) the stand-alone execution time of the contender was recorded for $L$ loops. The loop count $L$ thus enabled comparable measurements to be made irrespective of the trace execution times. The stand-alone execution time of the trace came directly from (i), while the resource sensitivity (per contender) for the trace was given by the difference between the trace execution times in (i) and (ii), and the resource stress for the trace by the difference in the contender's execution times in (ii) and (iii).

We repeated the runs for each trace 9 times to ensure consistency. Post processing of the raw timing data was used to eliminate anomalies caused by the kernel scheduler tick and the clock synchronization interrupt, neither of which could be disabled. The cycle counter was configured to pause when the scheduler was running, and so we were able to detect and eliminate anomalies due to the scheduler by comparing nanosecond clock and cycle counter readings. Measurement noise caused by the clock synchronization interrupt was more difficult to detect; however, we were able to filter out these anomalies by taking the median value for the 9 repeated runs for each trace, and by using the 95th percentile value (over the 10,000 traces) as the reference "maximum" increase in execution time for each task and contender.

Three contenders were used that cause contention by repeatedly accessing main memory. The contenders both stress the resource and are sensitive to contention. The three contenders have a similar structure, they differ only in the instruction patterns used: Read-Read (RR), Read-Write (RW), and Write-Write (WW). The read and write operations both compile down

[4] The case study software does not use the GPU.

to a single instruction. Each contender loop body included 100 memory access instructions, ensuring that the loop overhead, i.e. checking when the contender should stop, was small in comparison to the loop body. Hence each contender achieved close to the maximum possible load in terms of instructions that access memory and cause contention. The addresses used were such that the accesses had to go to memory, rather than being satisfied by the L1 cache. A handshaking protocol was used between task and contender to ensure that the contender started before and finished after the task. Further, dummy loops with no memory accesses were added before and after each task, to ensure that the experimental framework did not cause extra interference on the contender when it was running but the task was not.

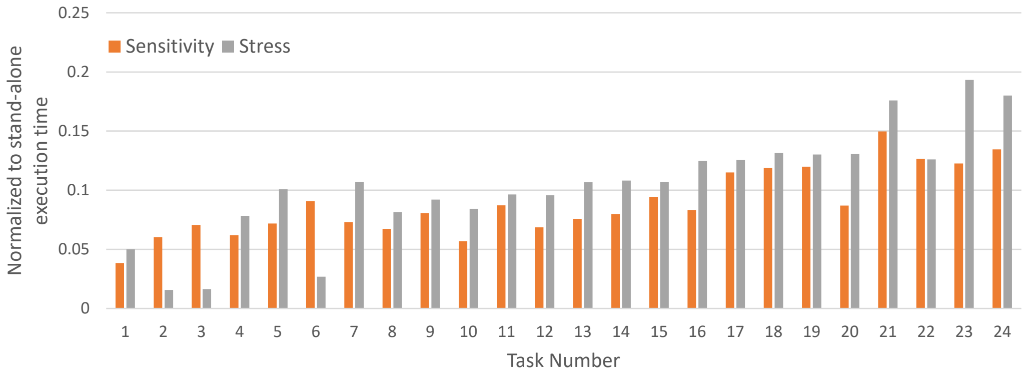


**Figure 7** Estimated resource stress and resource sensitivity values for 24 tasks from a Rolls-Royce aero-engine control systems normalized to the task's estimated stand-alone WCET.

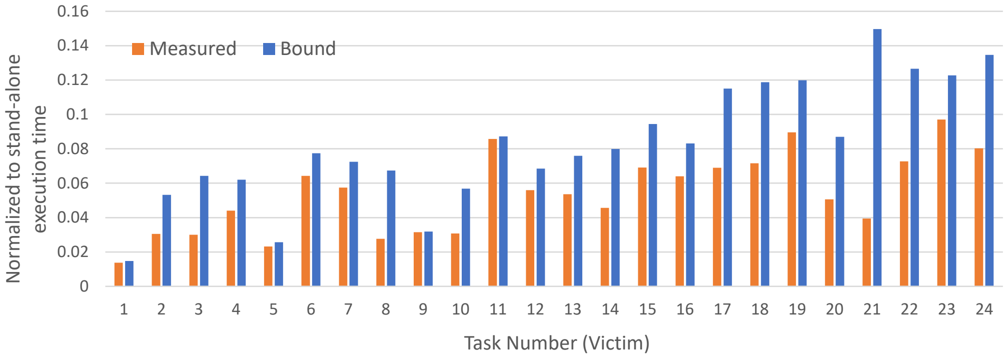


**Figure 8** Increase in execution time of a (victim) task co-running with its paired task. Maximum observed value and computed bound derived from resource sensitivity and resource stress values, normalized to the stand-alone execution time of the victim task.

Figure 7 shows the results of Experiment A.1, for 24 tasks from the Rolls-Royce aero-engine application, giving their maximum resource sensitivity and maximum resource stress normalized to the task's maximum stand-alone execution time. Note, the tasks appear in the figure ordered by their maximum stand-alone execution time, largest first. The RW contender was responsible for the maximum increase in task execution time (resource sensitivity) in all 24 cases. However, in terms of which contender suffered the maximum increase in execution time due to the task (i.e. resource stress), this was the RR contender in 2 cases, the RW contender in 3 cases, and the WW contender in 19 cases. Running a contender in parallel with a task increased the task's execution time by between 3.8% and 15.0% compared to stand-alone execution, thus characterizing the tasks' resource sensitivity. Further, the contender's execution time increased by between 1.5% and 19.3% of the task's stand-alone execution time, thus characterizing the tasks' resource stress. The ratio of resource stress to resource sensitivity for each task varied from 0.23 to 1.58. Some negative correlation can be observed between the stand-alone execution time and the percentage resource sensitivity and resource stress, with longer running tasks often having lower percentage values for these metrics. This is to be expected, since longer tasks typically spend more of their execution time in loops, running code that is cached, and therefore causes less resource contention.

As well as running tasks (traces) in parallel with the synthetic contenders, we also conducted **Experiment A.2**, running pairs of tasks in parallel on different cores. For each pair of tasks, we ran 10,000 pairs of their traces in parallel, with the inputs randomly selected as described previously. Figure 8 shows the maximum increase in execution time for each (victim) task due to cross-core contention from the task it was paired with. (The tasks were sorted by stand-alone execution time and then paired 1-2, 3-4, 5-6 and so on). The values shown are the maximum over the 10,000 pairs of traces, and are normalized to the stand-alone execution time of the victim task. Also shown is the bound computed from the minimum of (i) the resource sensitivity for the victim task and (ii) the resource stress for the task it was paired with, both obtained via Experiment A.1 using the synthetic contenders. The maximum measured increase in execution time is no greater than the computed bound. On average it is approx. 69% of the bound, and varies between 26% and 99%.

This preliminary case study underpins the MRSS task model, illustrating the relevance of using both resource sensitivity and resource stress to characterize cross-core contention, and thus bound interference.

## References

1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 3–11. IEEE, 2020. doi:10.1109/RTSS49844.2020.00012.

2 Sebastian Altmeyer, Robert I. Davis, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. A generic and compositional framework for multicore response time analysis. In Julien Forget, editor, *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 129–138. ACM, 2015. doi:10.1145/2834848.2834862.

3 Björn Andersson, Hyoseung Kim, Dionisio de Niz, Mark H. Klein, Ragunathan Rajkumar, and John P. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Trans. Embed. Comput. Syst.*, 17(3):71:1–71:29, 2018. doi:10.1145/3203407.

4 Neil C. Audsley. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, 2001. doi:10.1016/S0020-0190(00)00165-4.

5 Neil C. Audsley, Alan Burns, Michael Richardson, Kenneth W. Tindell, and Andrew J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292(8), September 1993. URL: https://digital-library.theiet.org/content/journals/10.1049/sej.1993.0034.

6 Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44, 2010.

7 Iain Bate, David Griffin, and Benjamin Lesage. Establishing confidence and understanding uncertainty in real-time systems. In Liliana Cucu-Grosjean, Roberto Medina, Sebastian Altmeyer, and Jean-Luc Scharbarg, editors, *28th International Conference on Real Time*

*Networks and Systems, RTNS 2020, Paris, France, June 10, 2020*, pages 67–77. ACM, 2020. doi:10.1145/3394810.3394816.

8 Sheng-Wei Cheng, Jian-Jia Chen, Jan Reineke, and Tei-Wei Kuo. Memory bank partitioning for fixed-priority tasks in a multi-core system. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 209–219. IEEE Computer Society, 2017. doi:10.1109/RTSS.2017.00027.

9 Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan M. Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2011, Changsha, China, 16-18 November, 2011*, pages 1068–1075. IEEE Computer Society, 2011. doi:10.1109/TrustCom.2011.146.

10 Robert I. Davis, Sebastian Altmeyer, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real Time Syst.*, 54(3):607–661, 2018. doi:10.1007/s11241-017-9285-4.

11 Robert I. Davis and Alan Burns. On optimal priority assignment for response time analysis of global fixed priority pre-emptive scheduling in multiprocessor hard real-time systems. Technical Report YCS-2010-451, University of York, Computer Science Dept., 2010.

12 Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real Time Syst.*, 47(1):1–40, 2011. doi:10.1007/s11241-010-9106-5.

13 Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real Time Syst.*, 35(3):239–272, 2007. doi:10.1007/s11241-007-9012-7.

14 Robert I. Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *J. Syst. Archit.*, 65:64–82, 2016. doi:10.1016/j.sysarc.2016.04.002.

15 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, July 2010. URL: http://retis.sssup.it/waters2010/waters2010.pdf.

16 Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 175–184. ACM, 2012. doi:10.1145/2380356.2380389.

17 Rudolf Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *29th Digital Avionics Systems Conference*, pages 5.E.3–1–5.E.3–11, 2010. doi:10.1109/DASC.2010.5655461.

18 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and nonpreemptive real-time uniprocessor scheduling. Technical report, INRIA Research Report, No. 2966, 1996. URL: https://hal.inria.fr/inria-00073732.

19 Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 63–72. ACM, 2012. doi:10.1145/2380356.2380372.

20 David Griffin, Iain Bate, and Robert I. Davis. Dirichlet-Rescale (DRS) algorithm software: dgdguk/drs: v1.0.0 available at https://doi.org/10.5281/zenodo.4118059, 2020.

21 David Griffin, Iain Bate, and Robert I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium, RTSS 2020,*

*Houston, TX, USA, December 1-4, 2020*, pages 76–88. IEEE, 2020. doi:10.1109/RTSS49844.2020.00018.

22 Mohamed Hassan. On the off-chip memory latency of real-time systems: Is DDR DRAM really the best option? In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 495–505. IEEE Computer Society, 2018. doi:10.1109/RTSS.2018.00062.

23 Wen-Hung Huang, Jian-Jia Chen, and Jan Reineke. MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 158:1–158:6. ACM, 2016. doi:10.1145/2897937.2898046.

24 Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. Slow and steady: Measuring and tuning multicore interference. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*, pages 200–212. IEEE, 2020. doi:10.1109/RTAS48715.2020.000-6.

25 Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986. doi:10.1093/comjnl/29.5.390.

26 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark H. Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real Time Syst.*, 52(3):356–395, 2016. doi:10.1007/s11241-016-9248-1.

27 Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, James H. Anderson, and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real Time Syst.*, 53(5):709–759, 2017. doi:10.1007/s11241-017-9272-9.

28 Kai Lampka, Georgia Giannopoulou, Rodolfo Pellizzoni, Zheng Wu, and Nikolay Stoimenov. A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real Time Syst.*, 50(5-6):736–773, 2014. doi:10.1007/s11241-014-9211-y.

29 Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 189–199. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.21.

30 Benjamin Lesage, Stephen Law, and Iain Bate. TACO: an industrial case study of test automation for coverage. In Yassine Ouhammou, Frédéric Ridouard, Emmanuel Grolleau, Mathieu Jan, and Moris Behnam, editors, *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018, Chasseneuil-du-Poitou, France, October 10-12, 2018*, pages 114–124. ACM, 2018. doi:10.1145/3273905.3273910.

31 Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Evaluation*, 2(4):237–250, 1982. doi:10.1016/0166-5316(82)90024-4.

32 Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019. doi:10.1145/3323212.

33 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In Cristian Constantinescu and Miguel P. Correia, editors, *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 132–143. IEEE Computer Society, 2012. doi:10.1109/EDCC.2012.27.

34 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. Iaˆ3: An interference aware allocation algorithm for multicore hard real-time systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 280–290. IEEE Computer Society, 2011. doi:10.1109/RTAS.2011.34.

35 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In Giovanni De Micheli, Bashir M. Al-Hashimi, Wolfgang Müller, and Enrico Macii, editors, *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, pages 741–746. IEEE Computer Society, 2010. doi:10.1109/DATE.2010.5456952.

36 Petar Radojkovic, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, 2012. doi:10.1145/2086696.2086713.

37 Rapita Systems. Multicore timing analysis for do-178c. https://www.rapitasystems.com/downloads/multicore-timing-analysis-do-178c.

38 Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In Alain Plantec, Frank Singhoff, Sébastien Faucou, and Luís Miguel Pinho, editors, *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 67–76. ACM, 2016. doi:10.1145/2997465.2997472.

39 Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, 2010. doi:10.1145/1880050.1880058.

40 Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 332–337. ACM, 2010. doi:10.1145/1837274.1837359.

41 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 161–172. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461361.

42 Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 184–195. IEEE Computer Society, 2015. doi:10.1109/ECRTS.2015.24.

# Response Time Bounds for DAG Tasks with Arbitrary Intra-Task Priority Assignment

**Qingqiang He** ✉
Department of Computing, The Hong Kong Polytechnic University, Hong Kong

**Mingsong Lv**[1] ✉
Department of Computing, The Hong Kong Polytechnic University, Hong Kong

**Nan Guan** ✉
Department of Computing, The Hong Kong Polytechnic University, Hong Kong

## Abstract

Most parallel real-time applications can be modeled as directed acyclic graph (DAG) tasks. Intra-task priority assignment can reduce the nondeterminism of runtime behavior of DAG tasks, possibly resulting in a smaller worst-case response time. However, intra-task priority assignment incurs dependencies between different parts of the graph, making it a challenging problem to compute the response time bound. Existing work on intra-task task priority assignment for DAG tasks is subject to the constraint that priority assignment must comply with the topological order of the graph, so that the response time bound can be computed in polynomial time. In this paper, we relax this constraint and propose a new method to compute response time bound of DAG tasks with *arbitrary* priority assignment. With the benefit of our new method, we present a simple but effective priority assignment policy, leading to smaller response time bounds. Comprehensive evaluation with both single-DAG systems and multi-DAG systems demonstrates that our method outperforms the state-of-the-art method with a considerable margin.

**2012 ACM Subject Classification** Software and its engineering → Real-time schedulability

**Keywords and phrases** real-time systems, response time bound, DAG tasks, intra-task priority assignment

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.8

## 1 Introduction

Multi-cores are becoming the mainstream of real-time systems for performance and energy efficiency. To utilize the power of multi-cores, software must be parallelized. Many parallel real-time applications can be modeled as directed acyclic graph (DAG) tasks. The DAG task model has gained much attention in the past few years [6,22,23,25]. In real-time community, researchers studied how to derive safe upper bounds for the response time of DAG tasks, which is a crucial characteristic for schedulability test.

When scheduling a DAG task, the execution order of eligible vertices has a large impact on the system schedulability [17,26]. A recent work [17] proposed to assign different priorities to vertices of a DAG task (i.e., intra-task priority assignment) to control the execution order of eligible vertices of a DAG task, and developed efficient algorithms to calculate safe response time bound of the DAG task in polynomial time. However, the approach in [17] is subject to the constraint that the intra-task priority must comply with the topological order of the graph (i.e., a vertex's priority cannot be higher than any of its ancestors). In general, allowing priority orders not complying with the topological order can lead to smaller response time bounds. The target of this work is to get rid of this constraint and further improve the schedulability of DAG tasks. More precisely, we develop algorithms to compute response time bounds of DAG tasks with *arbitrary* intra-task priority assignment.

[1] corresponding author

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 8; pp. 8:1–8:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The major challenge we face is how to deal with dependencies between different parts of the graph incurred by intra-task priority assignment when it does not comply with the topological order (see Example 4). We explore insights into these dependencies and the structure of the problem, and propose solving the problem by abstracting the graph in a context-free manner. An essential observation is that vertices with lower priorities can serve to isolate dependencies. With our computing method unleashing possibilities for arbitrary priority assignment, we devise a quite simple but effective priority assignment policy, leading to a much smaller bound. Experiments with both single-DAG systems and multi-DAG systems show that our method outperforms the previous method in [17] with a considerable margin.

## 2 Preliminary

### 2.1 Task Model

The parallel real-time task is modeled as a DAG $G = (V, E)$, where $V$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. Each vertex $v_i \in V$ represents a piece of sequential workload with worst-case execution time (WCET) $c(v_i)$ (for brevity, also denoted as $c_i$). An edge $(v_i, v_j) \in E$ represents the precedence relation between $v_i$ and $v_j$, i.e., $v_j$ can only start execution after vertex $v_i$ finishes its execution. A vertex with no incoming (outgoing) edges is called a *source vertex* (*sink vertex*). Without loss of generality, we assume that $G$ has exactly one source (denoted as $v_{src}$), and one sink (denoted as $v_{sink}$). In case $G$ has multiple source/sink vertices, a dummy source/sink vertex with zero WCET can be added to comply with our assumption.

A *path* $\lambda$ starting from vertex $\pi_0$ and ending at vertex $\pi_k$ ($\neq \pi_0$) is a sequence of vertices $(\pi_0, \cdots, \pi_k)$ such that $\forall i \in [0, k)$, $(\pi_i, \pi_{i+1}) \in E$, where $\pi_0$, $\pi_k$ are the *start vertex* and *end vertex* of path $\lambda$ respectively. We also use $\lambda$ to denote the set of vertices which are in the path $\lambda$. The length of a path $\lambda$ is defined as $len(\lambda) = \sum_{\pi_i \in \lambda} c(\pi_i)$. A *complete path* is a path $(\pi_0, \cdots, \pi_k)$ such that $\pi_0 = v_{src}$ and $\pi_k = v_{sink}$, i.e., a complete path is a path starting from the single source vertex and ending at the single sink vertex.

If there is an edge $(u, v) \in E$, $u$ is a *predecessor* of $v$, and $v$ is a *successor* of $u$. If there is a path in $G$ from $u$ to $v$, $u$ is an *ancestor* of $v$ and $v$ is a *descendant* of $u$. We use $pred(v)$, $succ(v)$, $ance(v)$ and $desc(v)$ to denote the set of predecessors, successors, ancestors and descendants of $v$, respectively.

For any vertex set $V' \subset V$, we define $vol(V') = \sum_{v_i \in V'} c_i$. The volume of a DAG $G$ denoted as $vol(G)$ is defined as $vol(V)$, i.e., $\sum_{v_i \in V} c_i$, which is the total WCET of all vertices in $G$. The longest path is a complete path with largest $len(\lambda)$ in $G$. $len(G)$ is defined as the length of the longest path.

### 2.2 Scheduling Model

We consider vertices of DAG $G$ scheduled on a multi-core platform with $m$ identical cores. The approach is divided into two phases:

- **Analysis phase.** In this phase, first, priorities are assigned to vertices (different vertices with identical priority are allowed). Formally, we assign a priority $p(v_i)$ to each vertex $v_i$ of the DAG, and say vertex $v_i$ has a higher priority than vertex $v_j$, if $p(v_i) < p(v_j)$. Second, response time bound is computed (the focus of this paper), and schedulability test is applied (not the focus of this paper).

- **Scheduling phase.** The scheduling algorithm for one DAG task with priority assignment is *prioritized list scheduling* [17], which is work-conserving and preemptive and always chooses at most $m$ highest-priority eligible vertices for execution.

## 2.3 Problem Formulation

We first state some notations to present the problem. The *parallel set* of a vertex $v \in V$ is defined as $para(v) = \{u \in V \setminus \{v\} | u \notin ance(v) \wedge u \notin desc(v)\}$.

▶ **Definition 1** (Interference Set [17]). *The interference set of a vertex $v \in V$ is defined as $I(v) = \{u \in V | u \in para(v) \wedge p(u) \leq p(v)\}$. The interference set of a path $\lambda$ is defined as $I(\lambda) = \bigcup_{\pi_i \in \lambda} I(\pi_i)$.*

▶ **Definition 2.** *For a path $\lambda$, $R(\lambda)$ is defined in [17] as*

$$R(\lambda) = len(\lambda) + \frac{vol(I(\lambda))}{m}$$

$R(\lambda)$ can be think of as the response time bound of this path $\lambda$. A response time bound for a DAG task was derived in [17] as stated in the following. The response time $R$ of a DAG $G$ with priority assignment scheduled by prioritized list scheduling on a platform with $m$ cores is bounded by:

$$R \leq \max_{\lambda \in \Pi(G)} \{R(\lambda)\} \tag{1}$$

where $\Pi(G)$ is the set of all complete paths of $G$.

It can be easily checked that this bound is timing-anomaly free or sustainable [8], thus providing a safe bound if some vertices execute for less than its WCET and self-sustainable [2], thus the bound not increasing if the number of cores increases. Actually, when some vertices execute for less than its WCET, for a path $\lambda$, its inference set $I(\lambda)$ being not change, $len(\lambda)$ and $vol(I(\lambda))$ do not increase. As a result, the computed $R$ in Equation 1 does not increase.

For a DAG task, the bound defined above varies for different priority assignments. In the computation of Equation 1, first, for a path, its interference set is computed which includes vertices that may interfere with the execution of this path. Second, the response time bound of the graph can be computed by searching all paths in this graph.

Although the bound is clearly defined in Equation 1, the computation of it can be a challenging task. Since the number of paths can be exponential in the size of the DAG, it is impractical to enumerate all the paths to compute the bound. Moreover, since the interference sets of two vertices in a path may contain the same vertex (see Example 4), which means dependencies exist among different parts of the graph, or different subproblems of the whole problem, it can be challenging to find a clear abstraction to compute the bound. We call the computation of Equation 1 as graph interference problem formulated as follows:

**Graph Interference Problem.** *For a DAG $G$ with priority assignment and the number of cores $m$, the objective of this problem is to compute the bound defined in Equation 1.*

## 2.4 An Illustrating Example

An example is given to explain concepts introduced in Section 2.

▶ **Example 3.** Figure 2b shows a DAG task with a priority assignment. The number inside the circles (representing vertices) is the WCET of vertices, and the red number besides vertices is its priority. $v_0, v_5$ are the single source vertex and the single sink vertex respectively (both are dummy vertices with zero WCET). The longest path is $\lambda_1 = (v_0, v_1, v_4, v_5)$, and path $\lambda_2 = (v_0, v_2, v_4, v_5)$ is a complete path. We can compute $vol(G) = 18$ and $len(G) = 9$. For vertex $v_4$, $pred(v_4) = \{v_1, v_2\}$, $succ(v_4) = \{v_5\}$, $ance(v_4) = \{v_0, v_1, v_2\}$, $desc(v_4) = \{v_5\}$. The priority of $v_1$ is higher than that of $v_2$ with $p(v_1) = 1$, $p(v_2) = 5$. $para(v_2) = \{v_1, v_3\}$, $I(v_2) = \{v_1, v_3\}$. $len(\lambda_2) = 4$, $I(\lambda_2) = \{v_1, v_3\}$.

Suppose that the number of cores is $m = 2$. $R(\lambda_2) = len(\lambda_2) + vol(I(\lambda_2))/m = 4 + 14/2 = 11$. The graph has three complete paths. We can compute the bound defined in Equation 1 being 11 by searching all three complete paths exhaustively, and path $\lambda_2 = (v_0, v_2, v_4, v_5)$ leads to this bound.

# 3 Motivation

## 3.1 Discussion on Existing Work

In [17], a dynamic programming algorithm was proposed to compute the bound defined in Equation 1, alongside its response time analysis. But its computing method assumes the intra-task priority assignment should comply with the topological order of the DAG. In the following, we briefly introduce the computing method of [17] given in Algorithm 1, and use an example to show that its method may not produce a correct bound for a DAG with a priority assignment not complying with the topological order.

**Algorithm 1** Bound Computation in [17].

```
1 $\sigma \leftarrow TopologicalOrder(G)$
2 $\lambda_{v_{src}} \leftarrow \{v_{src}\}$
3 for $v_i \in \sigma$ from $v_{src}$ to $v_{sink}$ do
4     if $v_i \neq v_{src}$ then
5         $u^* \leftarrow \arg\max_{u \in pred(v_i)} \{len(\lambda_u) + c_i + \frac{vol(I(\lambda_u) \cup I(v_i))}{m}\}$
6         $\lambda_{v_i} \leftarrow \lambda_{u^*} \cup \{v_i\}$
7     end
8 end
9 return $R(\lambda_{v_{sink}})$
```

The Algorithm first computes a topological order of the graph $G$ (Line 1), then searches through the topological order for a path with max $R(\lambda)$. In Line 5, whenever two paths join at a vertex $v_i$, it search through the predecessors of $v_i$ to find a path with maximum $R(\lambda)$ in the subgraph consisting of ancestors of $v_i$. This path is stored in $\lambda_{v_i}$. In Line 9, the searching reaches $v_{sink}$, and $R(\lambda_{v_{sink}})$ is returned as the response time bound of the whole graph.

The following example shows that Algorithm 1 may not compute a correct bound defined in Equation 1.

▶ **Example 4.** Figure 1 presents a DAG with a priority assignment. Red numbers besides vertices are priorities, and the sets above vertices are interference sets. Suppose that the number of cores is $m = 2$. We can compute the bound being 8 by searching all three complete paths exhaustively, and path $(v_0, v_1, v_4, v_5, v_6)$ leads to this bound. However, according to Algorithm 1, the computed bound is 7.5 and path $(v_0, v_2, v_4, v_5, v_6)$ leads to this bound, which is wrong and not the bound in Equation 1.

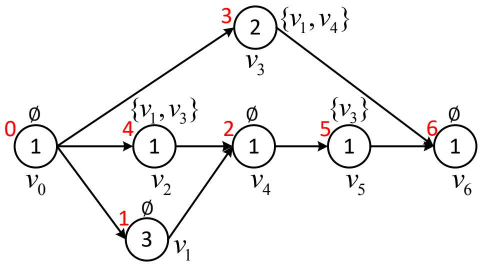
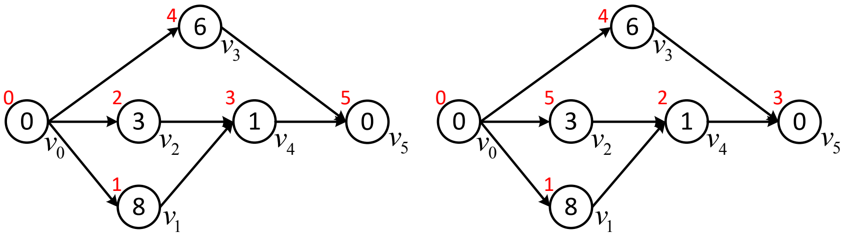


**Figure 1** A counterexample of Algorithm 1.



**(a)** With topology constraint. **(b)** Without topology constraint.

**Figure 2** A motivational example.

During Algorithm 1 for Example 4, when computing $v_4$ ($v_4$ as $v_i$ in Line 5), it chooses path $\lambda_1 = (v_0, v_2, v_4)$ as $\lambda_{v_4}$ because $\lambda_1$ suffers interference from $v_3$, thus $\lambda_1$ having a larger $R(\lambda)$ than that of path $(v_0, v_1, v_4)$. When computing $v_5$ in Line 5, it chooses path $(v_0, v_2, v_4, v_5)$ as $\lambda_{v_5}$ because $v_5$ has only one predecessor $v_4$ and the path stored in $\lambda_{v_4}$ is $(v_0, v_2, v_4)$. However, this choice leads to a wrong result as shown in Example 4.

The reason why Algorithm 1 produces a wrong bound for the above example is that vertex $v_3$, being in the interference sets of both $v_2$ and $v_5$, incurs dependencies between subgraph $\{v_0, v_1, v_2, v_4\}$ and subgraph $\{v_4, v_5\}$. In fact, priority assignment, when it becomes arbitrary, may incur dependencies between different parts of the graph, which the method in [17] cannot resolve. Actually, the computing method in [17] is only valid when the priority assignment satisfies *topology constraint*, i.e., a vertex's priority is not higher than any of its ancestors.

## 3.2 Motivation of this Work

Two reasons motivate us to study the problem of computing response time bound for DAG tasks with arbitrary intra-task priority assignment.

First, there are situations where priorities are predetermined (e.g, by industry practitioners) before the schedulability analysis phase. For example, in OpenMP [1], practitioners can use the *priority* clause to specify a priority for a *task* construct. It is not necessary that these priority assignments satisfy topology constraint. In these cases, the computing method in [17] cannot be applied.

Second, for priority assignment determined during analysis phase as the scheduling model in Section 2.2 assumes, it is possible that much smaller response time bounds can be achieved by relaxing topology constraint and allowing arbitrary priority assignment. Example 5 is an illustration of this finding.

▶ **Example 5.** Figure 2 shows a DAG with two priority assignments (red numbers besides vertices are priorities). Figure 2a is the priority assignment according to [17] with topology constraint, while the priority assignment in Figure 2b is without this constraint (because $v_2$, as an ancestor of $v_4$, has a priority lower than that of $v_4$). Suppose that the number of cores is $m = 2$. In Figure 2a, we can compute the bound being 12 by searching all three complete paths exhaustively, and path $\lambda_1 = (v_0, v_3, v_5)$ leads to this bound. In Figure 2b, as shown in Example 3, the computed bound is 11, and path $\lambda_2 = (v_0, v_2, v_4, v_5)$ leads to it.

As shown in [17], assigning higher priorities to vertices in a longer path may lead to a smaller response time. However, in Figure 2a, under topology constraint without which the method in [17] is invalid, $v_3$ in a path with length 6 has a priority lower (i.e., $p(v_3) > p(v_2)$) than that of $v_2$ in a path with length 4, leading to a larger bound than that of Figure 2b.

With the two motivations, this paper focuses on solving the graph interference problem with arbitrary priority assignment, thus unleashing the possibilities for a better (possibly optimal) priority assignment policy without topology constraint.

## 4 Computing Response Time Bound

In this section, we solve the graph interference problem precisely through abstraction of the graph in a context-free manner, assuming priority assignment is arbitrary.

We first give an overview of our abstraction framework. A DAG with priority assignment is treated as a sentence of a formal language, and the graph interference problem is how to parse the graph to compute an abstraction of the graph (i.e., the bound in Equation 1) under a context-free grammar [18]. The abstraction of (part of) the graph is represented as tuple (Definition 6). Starting from edges represented as simple tuples, through tuples connecting into new tuples, the abstraction of the whole graph (i.e. the bound) is constructed gradually. On one hand, the context-free grammar is expressed by the connection principle (Definition 9), which functions to identify the context-free parts of the graph to isolate dependencies on other parts in the graph. On the other hand, why these parts of graph are context-free and can be computed independently without having to consider the other parts of the graph connecting to them is explained in Lemma 13, which states that these tuple connections only depend on a limited number of vertices, whose information is included in these tuples themselves, not the other parts of the graph. An illustrative example of the above concepts is shown in Figure 3 located after the computing algorithm (Algorithm 2).

Next, we will go into the details of our abstraction framework.

▶ **Definition 6** (Tuple). *For a path $\lambda = (\pi_0, \cdots, \pi_k)$, we define a tuple*

$$\langle \pi_0, \pi_k, R(\lambda) \rangle$$

*where $\pi_0$, $\pi_k$ are the start vertex, end vertex of path $\lambda$.*

We say the tuple defined above corresponds to path $\lambda$, or path $\lambda$ corresponds to this tuple. We also call $\pi_0$, $\pi_k$ as the start vertex, end vertex of this tuple.

▶ **Definition 7** (Connection Vertex). *For a tuple $\langle u, v, R(\lambda) \rangle$, the connection vertex (denoted as $\kappa(u, v)$) of this tuple is defined as:*

$$\kappa(u,v) = \begin{cases} \perp & u = v_{src} \wedge v = v_{sink} \\ v & u = v_{src} \wedge v \neq v_{sink} \\ u & u \neq v_{src} \wedge v = v_{sink} \\ u & u \neq v_{src} \wedge v \neq v_{sink} \wedge p(u) \leq p(v) \\ v & u \neq v_{src} \wedge v \neq v_{sink} \wedge p(u) > p(v) \end{cases}$$

*where $\perp$ means no connection vertex.*

Although the equation for connection vertex seems complex, actually only two guidelines need to be kept in mind: (1) never choose a terminate vertex (i.e., $v_{src}$, $v_{sink}$); (2) always choose the vertex with a higher priority (i.e., $p(v)$ with a smaller value). According to Definition 7, except for tuples corresponding to a complete path, there is exactly one connection vertex in a tuple. For a tuple $\langle u, v, R(\lambda)\rangle$ with $u \neq v_{src} \vee v \neq v_{sink}$, we denote the priority of the connection vertex of this tuple as $p(\kappa(u, v))$(for brevity, also denoted as $p(u, v)$).

Two paths $\lambda_0$, $\lambda_1$ can be connected into a new path $\lambda$, if the end vertex of $\lambda_0$ is the same as the start vertex of $\lambda_1$, denoted as $\lambda = \lambda_0 \cup \lambda_1$. Similarly, two tuples can be connected into a new tuple, if the end vertex of the first tuple is the same as the start vertex of the second tuple.

▶ **Example 8.** For the graph in Figure 2b, path $\lambda_0 = (v_2, v_4)$ corresponds to a tuple $\alpha_0 = \langle v_2, v_4, R(\lambda_0)\rangle$, and path $\lambda_1 = (v_4, v_5)$ corresponds to a tuple $\alpha_1 = \langle v_4, v_5, R(\lambda_1)\rangle$. $\kappa(v_2, v_4) = v_4$, $\kappa(v_4, v_5) = v_4$. Since $v_4$ is the end vertex of $\alpha_0$ and the start vertex of $\alpha_1$, tuple $\alpha_0$ and $\alpha_1$ can be connected into a new tuple $\alpha = \langle v_2, v_5, R(\lambda)\rangle$, where $\lambda = \lambda_0 \cup \lambda_1 = (v_2, v_4, v_5)$.

With respect to tuple connection, we introduce the following connection principle.

▶ **Definition 9** (Connection Principle). *For two tuples $\langle u, v, R(\lambda_0)\rangle$, $\langle v, w, R(\lambda_1)\rangle$, if vertex $v$ is the connection vertex of these two tuples, then they can be connected into a new tuple.*

We call these two tuples are connected under the connection principle denoted by

$$\langle u, v, R(\lambda_0)\rangle + \langle v, w, R(\lambda_1)\rangle \rightsquigarrow \langle u, w, R(\lambda)\rangle \tag{2}$$

where $\lambda = \lambda_0 \cup \lambda_1$.

Note that since $R(\lambda_0)$ is just a value, the detailed information of a path is not stored in a tuple, which means $R(\lambda)$ cannot be computed by the above equation. Later in Lemma 14, the formula of computing the resulting tuple will be given.

For an edge $(u, v) \in E$, there is a path $\lambda = (u, v)$ and a tuple $\langle u, v, R(\lambda)\rangle$. A *simple tuple* is defined as a tuple $\langle u, v, R(\lambda)\rangle$, where $\lambda$ is actually an edge.

▶ **Definition 10** (Tuple under Connection Principle). *The definition is given by the following two recursive rules:*

- *A simple tuple is under connection principle;*
- *A tuple, which is computed according to Equation 2 by tuples under connection principle, is also under connection principle.*

▶ **Example 11.** In Example 8, since $\alpha_0$, $\alpha_1$ are simple tuples, these two tuples are under connection principle. Since $v_4$ is the connection vertex of $\alpha_0$, $\alpha_1$, this tuple connection is also under connection principle, thus the resulting tuple $\alpha$ being under connection principle.

For the rest of this paper, unless explicitly specified, all tuples and tuple connections are under connection principle.

▶ **Lemma 12** (Connection Property). *For a tuple $\langle u, w, R(\lambda)\rangle$ with $u \neq v_{src} \vee w \neq v_{sink}$ under connection principle, the following holds:*

$$\forall \pi_i \in \lambda \setminus \{u, w\}, p(\pi_i) \leq p(u, w)$$

**Proof.** We prove it by induction.
*Base case:* For a simple tuple, since $\lambda \setminus \{u, v\} = \varnothing$, the lemma holds trivially.
*Induction step:* For a tuple $\alpha = \langle u, w, R(\lambda)\rangle$ that is not a simple tuple. Since $\alpha$ is under

connection principle, by Definition 10, there exist two tuples $\alpha_0 = \langle u, v, R(\lambda_0)\rangle$, $\alpha_1 = \langle v, w, R(\lambda_1)\rangle$, both being under connection principle, satisfying $\alpha_0 + \alpha_1 \rightsquigarrow \alpha$. Since $\alpha_0$, $\alpha_1$ are under connection principle, by induction hypothesis, both $\alpha_0$ and $\alpha_1$ satisfy connection property.

It is clear that $v \neq v_{src} \land v \neq v_{sink}$, and we already have $\kappa(u, v) = v$, $\kappa(v, w) = v$, which means $p(u, v) = p(v)$, $p(v, w) = p(v)$.

There are three cases: (1) $u = v_{src} \land w \neq v_{sink}$; (2) $u \neq v_{src} \land w = v_{sink}$; (3) $u \neq v_{src} \land w \neq v_{sink}$. For the first case, according to Definition 7, we have $\kappa(u, w) = w$, which means $p(u, w) = p(w)$. Since $\kappa(v, w) = v$, we have $p(v) \leq p(w)$, which means $p(v) \leq p(u, w)$. For the second case, according to similar reasons, we have $p(v) \leq p(u, w)$. For the third case, since $\kappa(u, v) = v$, $\kappa(v, w) = v$, according to Definition 7, we have $p(v) \leq p(u)$ and $p(v) \leq p(w)$, which means $p(v) \leq p(u, w)$. In summary, for the three cases, $p(v) \leq p(u, w)$.

Since $\alpha_0$ satisfies connection property, we have

$$\forall \pi_i \in \lambda_0 \setminus \{u, v\}, p(\pi_i) \leq p(u, v)$$

Note that $p(u, v) = p(v)$ and $p(v) \leq p(u, w)$, we have

$$\forall \pi_i \in \lambda_0 \setminus \{u, v\}, p(\pi_i) \leq p(u, w)$$

For the same reason, we have

$$\forall \pi_i \in \lambda_1 \setminus \{v, w\}, p(\pi_i) \leq p(u, w)$$

Note that $\lambda = \lambda_0 \cup \lambda_1$, we have

$$\forall \pi_i \in \lambda \setminus \{u, w\}, p(\pi_i) \leq p(u, w)$$

which means that tuple $\alpha$ satisfies connection property. The lemma follows. ◀

Under the principle, a key observation for the connection of tuples is that the computation does not depend on the whole path necessarily, actually only depends on a limited number of vertices on this path, as stated in the following lemma.

▶ **Lemma 13.** *If two tuples $\langle u, v, R(\lambda_0)\rangle$, $\langle v, w, R(\lambda_1)\rangle$ under connection principle are connected into a new tuple $\langle u, w, R(\lambda)\rangle$ according to Equation 2, then*

$$I(\lambda_0) \cap I(\lambda_1) = I(v) \cup (I(u) \cap I(w)) \tag{3}$$

**Proof.** We use LHS and RHS to represent the left-hand side and right-hand side of Equation 3. Next, we shall prove the lemma by showing that both RHS⊆LHS and LHS⊆RHS hold.

**(1)** RHS⊆LHS. Since $v \in \lambda_0$ and $v \in \lambda_1$, we have

$$I(v) \subseteq I(\lambda_0) \cap I(\lambda_1)$$

Since $u \in \lambda_0$ and $w \in \lambda_1$, we have

$$I(u) \cap I(w) \subseteq I(\lambda_0) \cap I(\lambda_1)$$

In summary, we reach that

$$I(v) \cup (I(u) \cap I(w)) \subseteq I(\lambda_0) \cap I(\lambda_1)$$

**(2)** LHS⊆RHS. Obviously, $\kappa(u,v)=v$, $\kappa(v,w)=v$, $p(u,v)=p(v)$, $p(v,w)=p(v)$. According to Definition 1, $\forall x\in I(\lambda_0)\cap I(\lambda_1)$, $\exists\pi_i\in\lambda_0$ and $\exists\pi_j\in\lambda_1$, such that vertex $x\in I(\pi_i)$ and $x\in I(\pi_j)$, which means $p(x)\le p(\pi_i)$ and $p(x)\le p(\pi_j)$.

Next, we shall prove that $x\in para(v)$. If $x\in ance(v)$, since $v\in ance(\pi_j)\vee v=\pi_j$, then $x\in ance(\pi_j)$, which contradicts $x\in I(\pi_j)$. We have $x\notin ance(v)$. By similar reasons, $x\notin desc(v)$. In summary, $x\in para(v)$.

In the following, we prove by exhaustion. There are three cases:

**(a)** $\pi_i\ne u$. Since $\kappa(u,v)=v$, by Lemma 12, we have $p(\pi_i)\le p(v)$. Note that $p(x)\le p(\pi_i)$, so $p(x)\le p(v)$. Together with $x\in para(v)$, we have $x\in I(v)$, which means $x\in I(v)\cup(I(u)\cap I(w))$.

**(b)** $\pi_j\ne w$. For similar reasons to the first case, $x\in I(v)$, so $x\in I(v)\cup(I(u)\cap I(w))$.

**(c)** $\pi_i=u\wedge\pi_j=w$. Since $x\in I(\pi_i)$ and $x\in I(\pi_j)$, we have $x\in I(u)$ and $x\in I(w)$, which means $x\in I(u)\cap I(w)$. We reach that $x\in I(v)\cup(I(u)\cap I(w))$.

Summarizing these three cases, we have $\forall x\in I(\lambda_0)\cap I(\lambda_1)$, $x\in I(v)\cup(I(u)\cap I(w))$.

In conclusion, the lemma follows. ◀

▶ **Lemma 14.** *If two tuples $\langle u,v,R(\lambda_0)\rangle$, $\langle v,w,R(\lambda_1)\rangle$ under connection principle are connected into a new tuple $\langle u,w,R(\lambda)\rangle$ according to Equation 2, then*

$$R(\lambda)=R(\lambda_0)+R(\lambda_1)-c(v)-\frac{vol(I(v)\cup(I(u)\cap I(w)))}{m}\tag{4}$$

**Proof.** By Definition 2,

$$R(\lambda)=len(\lambda)+\frac{vol(I(\lambda))}{m}$$

Since $\lambda=\lambda_0\cup\lambda_1$, we have $len(\lambda)=len(\lambda_0)+len(\lambda_1)-c(v)$ and $I(\lambda)=I(\lambda_0)\cup I(\lambda_1)$. Further,

$$R(\lambda)=R(\lambda_0)+R(\lambda_1)-c(v)-\frac{vol(I(\lambda_0))+vol(I(\lambda_1))-vol(I(\lambda))}{m}$$

Since $I(\lambda)=I(\lambda_0)\cup I(\lambda_1)$, we have

$$vol(I(\lambda_0))+vol(I(\lambda_1))-vol(I(\lambda))=vol(I(\lambda_0)\cap I(\lambda_1))$$

By Lemma 13, we have

$$vol(I(\lambda_0))+vol(I(\lambda_1))-vol(I(\lambda))=vol(I(v)\cup(I(u)\cap I(w)))$$

Together, we reach the conclusion. ◀

The meaning of the above lemma is twofold. First, it gives a formula to compute the connection of tuples iteratively. Second, it implies that for two paths with the same start and end vertex, the path with a smaller $R(\lambda)$ cannot result in a path with a larger $R(\lambda)$, as stated in Lemma 16 formally.

▶ **Definition 15** (Domination)**.** *Given two paths with the same start and end vertex, there are two tuples $\langle u,v,R(\lambda)\rangle$, $\langle u,v,R(\lambda')\rangle$. We say $\langle u,v,R(\lambda)\rangle$ dominates $\langle u,v,R(\lambda')\rangle$, denoted by*

$$\langle u,v,R(\lambda)\rangle\succcurlyeq\langle u,v,R(\lambda')\rangle$$

*if and only if $R(\lambda)\ge R(\lambda')$.*

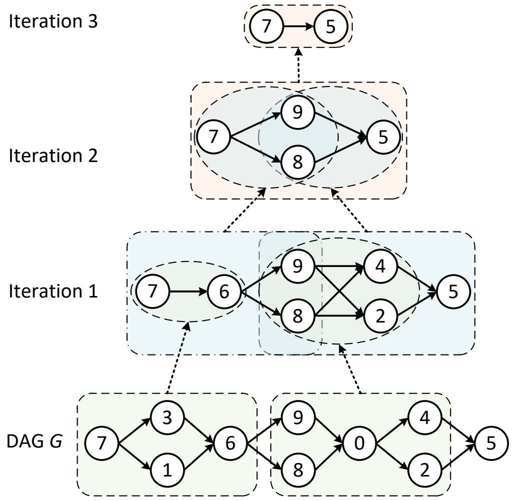


**Figure 3** An example illustrating Algorithm 2.

▶ **Lemma 16.** *Under connection principle, given*

$$\langle u, v, R(\lambda_0)\rangle + \langle v, w, R(\lambda_1)\rangle \rightsquigarrow \langle u, w, R(\lambda)\rangle$$

*and*

$$\langle u, v, R(\lambda_0')\rangle + \langle v, w, R(\lambda_1)\rangle \rightsquigarrow \langle u, w, R(\lambda')\rangle$$

*If*

$$\langle u, v, R(\lambda_0)\rangle \succcurlyeq \langle u, v, R(\lambda_0')\rangle$$

*then*

$$\langle u, w, R(\lambda)\rangle \succcurlyeq \langle u, w, R(\lambda')\rangle$$

**Proof.** According to Definition 15, we have $R(\lambda_0) \geq R(\lambda_0')$

$$\begin{aligned}&\Rightarrow R(\lambda_0) + R(\lambda_1) - c(v) - \frac{vol(I(v) \cup (I(u) \cap I(w)))}{m}\\ &\qquad \geq R(\lambda_0') + R(\lambda_1) - c(v) - \frac{vol(I(v) \cup (I(u) \cap I(w)))}{m}\\ &\Rightarrow R(\lambda) \geq R(\lambda')\\ &\Rightarrow \langle u, w, R(\lambda)\rangle \succcurlyeq \langle u, w, R(\lambda')\rangle\end{aligned}$$

The conclusion is reached. ◀

The above lemma means when computing Equation 1, tuples with a smaller $R$ can be discarded safely, since in future computation they cannot result in a tuple with a larger $R$. We summarize the above discussion into Algorithm 2 to compute the response time bound of a DAG as defined in Equation 1.

Figure 3 provides an illustrative example of Algorithm 2 to show the steps of abstraction of the graph. The graph is at the bottom of Figure 3 where the number inside each vertex is its priority (The WCET of each vertex is irrelevant to the example, and we omit such

**Algorithm 2** Computing graph interference problem.

```
   Input  : DAG $G = (V, E)$; every vertex $v_i \in V$ is with its WCET $c_i$ and its priority
            $p(v_i)$; the number of cores $m$
   Output : the response time bound defined in Equation 1
1  $TS \leftarrow \{\langle u, v, R(\lambda)\rangle \mid \lambda = (u, v) \in E\}$
2  repeat
3      for each $(\alpha_i, \alpha_j) \in TS \times TS$ do
4          if $\alpha_i$, $\alpha_j$ can be connected by Definition 9 then
5              $\alpha \leftarrow \alpha_i + \alpha_j$ by Equation 2, 4
6              if $\exists \beta \in TS$ such that $\beta \succcurlyeq \alpha$ then
7                  continue
8              else if $\exists \beta \in TS$ such that $\alpha \succcurlyeq \beta$ then
9                  $TS \leftarrow (TS \setminus \{\beta\}) \cup \{\alpha\}$
10             else
11                 $TS \leftarrow TS \cup \{\alpha\}$
12             end
13         end
14     end
15 until nothing changes in $TS$
16 return $R$ such that $\langle v_{src}, v_{sink}, R\rangle \in TS$
```

information in the figure). In the example, since each vertex has a unique priority, we also use the priority to identify the vertex. After the first iteration of the loop in Line 2-15, with tuple connections under the connection principle, the original graph is transformed as illustrated in the figure. After three iterations of the loop, the tuple with start vertex and end vertex being $v_{src}$ and $v_{sink}$ respectively appears, which means the bound in Equation 1 is computed. It is easy to observe that all tuple connections in Figure 3 follow the connection principle in Definition 9. The context-free parts of the graph are indicated as colored rectangles in Figure 3. The relations between the context-free parts of the graph and their abstractions during each iteration are indicated as dashed arrows, which form an abstract syntax tree of the original graph.

Note that for clear and concise presentation of the principle behind Algorithm 2, the illustration of Figure 3 is not exactly the same as Algorithm 2. In Algorithm 2, during one iteration, plenty of tuples are connected, much of them being redundant and having been connected in the previous iterations. Since these redundant tuple connections are irrelevant to the correctness and theoretical computational complexity of Algorithm 2, we do not show these connections in Figure 3.

Next, we introduce the concept of *abstract path*, which is useful for proving the correctness of Algorithm 2.

▶ **Definition 17** (Abstract Path). *An abstract path $\lambda = (\pi_0, \cdots, \pi_k)$ $(k > 0)$ is a sequence of vertices such that $\forall i \in [0, k)$, there is a path $\lambda_i$ with start and end vertex being $\pi_i$, $\pi_{i+1}$ respectively. Further, for an abstract path $\lambda = (\pi_0, \cdots, \pi_k)$, we define $TS(\lambda) = \{\langle \pi_i, \pi_{i+1}, R(\lambda_i)\rangle \mid i \in [0, k)\}$ as the set of tuples corresponding to $\lambda_i$.*

Note that an abstract path is always an abstraction of a concrete path(i.e., $\cup_{i \in [0,k)} \lambda_i$ by using the above notation). To compute $TS(\lambda)$, the concrete path behind the abstract path $\lambda$ should be given. Since $TS(\lambda)$ only serves as an intermediate concept when proving the correctness of Algorithm 2, for brevity, we will omit this concrete path. According to the definition of abstract path, a path is an abstract path, while an abstract path is not necessarily a path.

▶ **Lemma 18** (Connection Lemma). *For an arbitrary abstract path $\lambda = (\pi_0, \cdots, \pi_k)$ with $k > 1 \wedge \pi_0 = v_{src} \wedge \pi_k = v_{sink}$, $\exists \alpha, \beta \in TS(\lambda)$, such that tuple $\alpha$, $\beta$ can be connected under connection principle.*

**Proof.** Since $k > 1$, there are at least two tuples in $TS(\lambda)$. In the following, we prove this by contradiction. Assume that there are not two tuples which can be connected under connection principle, next we will consider the priority assignment along this abstract path starting from $\pi_0 = v_{src}$. There are four cases.

**(1)** $p(\pi_0) > p(\pi_1) \wedge p(\pi_1) \leq p(\pi_2)$. Since $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, then $\langle \pi_0, \pi_1, R(\lambda_0) \rangle$, $\langle \pi_1, \pi_2, R(\lambda_1) \rangle$ can be connected under connection principle, which is a contradiction. Note that the reasoning in this case does not rely on $\pi_0 = v_{src}$, which means if $\pi_0$ is an arbitrary vertex in $\lambda$, the above reasoning is valid.

**(2)** $p(\pi_0) > p(\pi_1) \wedge p(\pi_1) > p(\pi_2)$. The reasoning in this case does not rely on $\pi_0 = v_{src}$ either.

**(2a)** If $\pi_2 = v_{sink}$, then $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, which means a contradiction.

**(2b)** If $\pi_2 \neq v_{sink}$, consider $p(\pi_3)$. If $p(\pi_2) \leq p(\pi_3)$, we have the pattern $p(\pi_1) > p(\pi_2) \wedge p(\pi_2) \leq p(\pi_3)$. Since case (1) does not rely on $\pi_0 = v_{src}$, this pattern is the same as case (1) and finally leads to a contradiction. Actually in this case, to ensure two tuples cannot be connected under connection principle as indicated in the assumption, by the above reasoning, the priority of the next vertex $\pi_{i+1}$ must be higher than that of the previous vertex $\pi_i$, formally $p(\pi_i) > p(\pi_{i+1})$. Considering all vertices along $\lambda$, finally we reach $\pi_k = v_{sink}$, and we have $p(\pi_{k-2}) > p(\pi_{k-1}) \wedge p(\pi_{k-1}) > p(v_{sink})$, which is the case in (2a) and finally leads to a contradiction.

**(3)** $p(\pi_0) \leq p(\pi_1) \wedge p(\pi_1) \leq p(\pi_2)$. Since $\pi_0 = v_{src}$, then $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, which means a contradiction.

**(4)** $p(\pi_0) \leq p(\pi_1) \wedge p(\pi_1) > p(\pi_2)$.

**(4a)** If $\pi_2 = v_{sink}$, since $\pi_0 = v_{src}$, then $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, which means a contradiction.

**(4b)** If $\pi_2 \neq v_{sink}$, consider $p(\pi_3)$. If $p(\pi_2) \leq p(\pi_3)$, we have $p(\pi_1) > p(\pi_2) \wedge p(\pi_2) \leq p(\pi_3)$, which is the case in (1) and finally leads to a contradiction. If $p(\pi_2) > p(\pi_3)$, we have $p(\pi_1) > p(\pi_2) \wedge p(\pi_2) > p(\pi_3)$, which is the case in (2) and finally leads to a contradiction.

In summary, all cases lead to a contradiction, therefore the initial assumption must be false. We reach the conclusion. ◀

For an abstract path $\lambda = (\pi_0, \cdots, \pi_i, \pi_{i+1}, \pi_{i+2}, \cdots, \pi_k)$, tuple $\alpha = \langle \pi_i, \pi_{i+1}, R(\lambda_i) \rangle$, $\beta = \langle \pi_{i+1}, \pi_{i+2}, R(\lambda_{i+1}) \rangle \in TS(\lambda)$, suppose $\alpha, \beta$ can be connected under connection principle, this connection will result in a new abstract path $\lambda' = (\pi_0, \cdots, \pi_i, \pi_{i+2}, \cdots, \pi_k)$ and a new $TS(\lambda')$ with $|TS(\lambda')| = |TS(\lambda)| - 1$.

▶ **Example 19.** For the graph in Figure 2b, for path $\lambda = (v_0, v_2, v_4, v_5)$ (which is also an abstract path by definition), $TS(\lambda) = \{\alpha_0 = (v_0, v_2, R_0), \alpha_1 = (v_2, v_4, R_1), \alpha_2 = (v_4, v_5, R_2)\}$. From Example 8, by Lemma 18, we know in $TS(\lambda)$, there exist $\alpha_1$, $\alpha_2$ that can be connected under connection principle. This tuple connection results in a new abstract path $\lambda' = \{v_0, v_2, v_5\}$ and $TS(\lambda') = \{(v_0, v_2, R'_0), (v_2, v_5, R'_1)\}$. It is obvious that $|TS(\lambda')| = |TS(\lambda)| - 1$.

Concerning the correctness and complexity of Algorithm 2, two aspects need to be considered. On one hand, according to connection principle in Definition 9, if two tuples are to be connected, first the end vertex of a tuple should be the start vertex of another tuple; second the connection vertex of these two tuples should be the same. Since we do not make any assumption about priority assignment, which is different from [17] where priority assignment should comply with topological order, is it possible that after the loop in Line 2-15 finishes, there is not a tuple $\langle u, v, R\rangle$ with $u = v_{src} \wedge v = v_{sink}$ in $TS$ as required by Line 16? On the other hand, since the loop in Line 2-15 does not exit until nothing changes in $TS$, how can we guarantee that the loop will be completed within a reasonable number of iterations? By using Lemma 18, we address these concerns in the following theorem.

▶ **Theorem 20.** *The return value of Algorithm 2 equals the bound in Equation 1.*

**Proof.** We define $TS_{max} = \{\langle u, v, R\rangle \in TS | u = v_{src} \wedge v = v_{sink}\}$. The theorem is proved by the following three steps:

**(1)** In Line 16 of Algorithm 2, $|TS_{max}| = 1$.

- **a.** $|TS_{max}| \geq 1$. For $\forall\lambda \in \Pi(G)$, if only two vertices in this path (i.e., $|\lambda| = 2$), then these two vertices must be $v_{src}$ and $v_{sink}$, which means $\alpha = \langle v_{src}, v_{sink}, R(\lambda)\rangle$ is added to $TS$ in Line 1. Although $\alpha$ might be removed from $TS$ in Line 9, this only happens when $\alpha$ is dominated by a new tuple with start vertex and end vertex being $v_{src}$, $v_{sink}$ respectively. If $|\lambda| > 2$, according to Lemma 18, after one iteration of the loop in Line 2-15, at least two tuples in $TS(\lambda)$ must be connected under connection principle, resulting in a new abstract path $\lambda'$ with $|TS(\lambda')| \leq |TS(\lambda)| - 1$. This fact means after at most $|V|$ iterations of the loop in Line 2-15, a tuple with start vertex being $v_{src}$ and end vertex being $v_{sink}$ corresponding to $\lambda$ (or a tuple dominating the tuple corresponding to $\lambda$) will be eventually computed. Since $\Pi(G) \neq \varnothing$, $|TS_{max}| \geq 1$.
- **b.** $|TS_{max}| \leq 1$. According to Definition 15, It is sufficient to show that in any step of the algorithm, $\nexists\alpha, \beta \in TS$, such that $\alpha \succcurlyeq \beta \vee \beta \succcurlyeq \alpha$. First, obviously in Line 1, the statement is true. Second, during the loop in Line 2-15, according to the conditions in Line 6, 8, it is evident that Line 9, 11 will not lead to domination between tuples in $TS$.

We denote the only tuple in $TS_{max}$ as $\alpha_{max}$.

**(2)** There is a complete path $\lambda_{max} \in \Pi(G)$ corresponding to $\alpha_{max}$. It is sufficient to show that $\forall\alpha \in TS$, there is a path corresponding to $\alpha$. First, in Line 1, all simple tuples are added to $TS$. It is evident that a simple tuple corresponds to an edge, which is also a path. Second, in Line 5, according to Equation 2, for every tuple connection, two paths are connected into a new path corresponding to the newly computed tuple.

**(3)** There is no complete path $\lambda \in \Pi(G)$ such that $R(\lambda) > R(\lambda_{max})$. We prove this statement by showing that $\forall\lambda \in \Pi(G)$, $\langle v_{src}, v_{sink}, R(\lambda_{max})\rangle \succcurlyeq \langle v_{src}, v_{sink}, R(\lambda)\rangle$. First, in Line 1, all simple tuples are added to $TS$. We have $\forall\lambda \in \Pi(G)$, $TS(\lambda) \subseteq TS$, which means all complete paths have a representation (i.e. $TS(\lambda)$) in $TS$. Second, it is obvious that all tuples in $TS$ are under connection principle. According to Definition 15, together with the discussion in 1(a), $\forall\lambda \in \Pi(G)$, after at most $|V|$ iterations of the loop in Line 2-15, $\langle v_{src}, v_{sink}, R(\lambda)\rangle$ will either be computed in $TS$ (in this case, $R(\lambda) = R(\lambda_{max})$), or be dominated (in this case, $R(\lambda) \leq R(\lambda_{max})$). We reach the conclusion.

Summarizing above three steps, the theorem is proved. ◀

▶ **Theorem 21.** *The time complexity of Algorithm 2 is polynomially bounded in $|V|$.*

In the above proof, from (1a), the number of iterations of the loop in Line 2-15 will not exceed $|V|$; from (1b), since no tuple domination in $TS$, the number of tuples in $TS$ will not exceed $|V|^2$. In Line 3, there is $TS \times TS$. So the number of iterations of the loop in Line 3-14 will not exceed $|V|^4$. The time complexity of Algorithm 2 is $O(|V|^5)$. However, in Line 4, when tuple $\alpha_i$ is determined, the end vertex of $\alpha_i$ being determined, actually only $|V|$ number of $\alpha_j$ in $TS$ need to be examined. Consequently, the algorithm can be easily implemented with time complexity being $O(|V|^4)$.

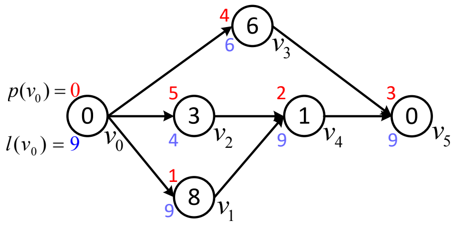


Figure 4 An example illustrating priority assignment.

## 5 Priority Assignment

To illustrate the performance of arbitrary priority assignment supported by our computing method in Section 4, we devise a priority assignment policy without topology constraint, leading to a much smaller bound as shown in Section 7. The guideline is to assign higher priorities to vertices in a longer path as indicated in Section 3.2. We first introduce a concept to identify vertices in a longer path.

▶ **Definition 22** (Vertex Length). *The vertex length of $v$ (denoted as $l(v)$) is defined as*

$$l(v) = max\{len(\lambda) \mid \lambda \in \Pi(G) \wedge v \in \lambda\} \quad (5)$$

Intuitively, vertex length is the longest path length among all the paths which go through the vertex. Vertex length can be computed by a straightforward dynamic programming in polynomial time with respect to the size of the graph (Algorithm 3 in [17]). The fixed priorities of vertices are assigned based on vertex length as follows.

- A vertex with a larger length is assigned a higher priority, formally $p(v_i) < p(v_j)$ if $l(v_i) > l(v_j)$;
- If two vertices have the same length, the vertex with a smaller index in the graph is given a higher priority, formally $p(v_i) < p(v_j)$ if $l(v_i) = l(v_j) \wedge i < j$.

The vertex index does not necessarily follow topological order. Ties can be broken by any other rules. The second rule is introduced to make the priority assignment policy determinate and make the evaluation in Section 7 reproducible.

▶ **Example 23.** For the graph in Figure 2b, the length (the blue number below vertices) of each vertex is labelled in Figure 4, and a priority assignment (the red number above vertices) according to the proposed policy is also illustrated.

In Figure 4, the priority of $v_4$ is higher (smaller priority value means higher priority) than the priority of $v_2$ (note that $v_2$ is an ancestor of $v_4$), which does not comply with topology constraint (i.e., a vertex's priority is not higher than any of its ancestors). In consequence, the proposed priority policy is without topology constraint. As an illustrative specific example, the proposed priority assignment policy indeed relies on some topological characteristics (e.g., the vertex length in Definition 22). However, the claim and the main contribution of this paper is that our computing method is valid for arbitrary priority assignment, thus not limited to topology constraint required in [17].

We note that the proposed priority assignment policy does not strictly dominate the policy in [17], i.e., not always producing a bound smaller than the bound of the policy in [17]. However, the proposed policy is much simpler, and leads to a smaller bound in general cases (actually, only in very rare cases with a larger bound, see Section 7.1).

# 6 Extension to Multi-DAG Systems

The proposed method for computing response time bound for single DAG task can also be applied to multi-DAG sporadic systems. The approach of utilizing intra-task priority to improve the schedulability of multi-DAG systems was introduced in [17]. Although the intra-task priority studied in this work is without topology constraint, thus the computing method and priority assignment policy being completely different, the response time analysis behind the bound in Equation 1 is still the same. Therefore, the approach of [17] can be used directly to extend our method to multi-DAG systems. We briefly introduce it to help understanding the experiments in Section 7.2.

The scheduling algorithm for multi-DAG systems is *global prioritized list scheduling* [17], which has two levels: task level and vertex level. In task level, a priority policy, e.g., early deadline first (EDF) and rate monotonic (RM), is employed to determine the highest-priority ready DAG task; in vertex level, prioritized list scheduling is used. After priorities between tasks and further between vertices are determined, the scheduling behavior is unchanged, which is global, work-conserving, preemptive, and always chooses at most $m$ (the core number) highest-priority eligible vertices for execution.

▶ **Theorem 24** ( [17]). *For a multi-DAG sporadic system with constrained deadlines scheduled by global prioritized list scheduling on a platform with $m$ cores, a bound $R_j$ on the response time of a task $\tau_j$ can be derived by the fixed-point iteration of the following equation, starting with $R_j = len(G_j)$:*

$$R_j = \max_{\lambda \in \Pi(G_j)} \{len(\lambda) + \frac{vol(I(\lambda))}{m}\} + \frac{\sum_{i \neq j} I_j^i(R_j)}{m} \tag{6}$$

*where $I_j^i(R_j)$ is the upper bound of the interference of task $\tau_i$ to $\tau_j$ during an interval of length $R_j$.*

In Equation 6, $I_j^i(R_j)$ is related to task level priority policies and is computed by the method in [25]. The computation of this term for EDF and RM is detailed in Lemma V.2 and Lemma V.1 of [25], respectively. For details of Theorem 24, please refer to [17].

# 7 Performance Evaluation

In this section, the performance of the proposed method is evaluated. During the evaluation, we conduct experiments of both scheduling single-DAG systems and scheduling multi-DAG systems using randomly generated task graphs.

## 7.1 Evaluation of Single-DAG Systems

In this section, the following methods are compared:

- *B-OUR*. The bound in Equation 1 computed by our method with priority assignment policy introduced in Section 5.
- *B-ZHAO*. The bound proposed in [29] with explicit execution order developed alongside its analysis.
- *B-HE*. The bound proposed in [17] with its priority assignment policy.

These three bounds are normalized with respect to *B-HE* as the metric for comparison. So in the figures with respect to normalized bound, *B-HE* is always one. The bounds computed by other priority assignment algorithms [21], [19] are shown dominated by [17], thus not included in the evaluation.

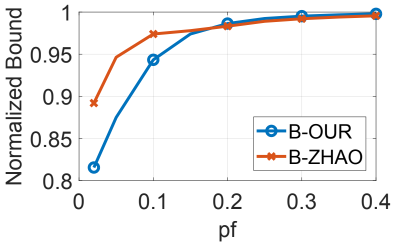


**Figure 5** Normalized bound with different $pf$.

**Table 1** The percentage of inferior cases comparing with *B-HE* and *B-ZHAO*.

| $m$ | *B-HE* | *B-ZHAO* |
|---|---|---|
| 4-10 | 0.11% | 77.69% |
| 11-16 | 0% | 17.38% |
| 17-22 | 0% | 1% |
| 23-32 | 0.01% | 1.44% |

**Task Generation.** The DAG tasks are generated using the Erdos-Renyi method [10], where the number of vertices $|V|$ is randomly chosen in a specified interval. For each pair of vertices, the method generates a random value in [0, 1] and adds the edge to the graph if the generated value is less than a predefined *parallelism factor* $pf$. The larger $pf$, the more sequential the graph is. If the generated graph has multiple source/sink vertices, a dummy source/sink vertex with zero WCET is added to the graph.

For experiments, we first give a default parameter setting, then tune different parameters for evaluation. The default setting is as follows. The vertex number $|V|$ and the WCET of vertices $c_i$ are randomly chosen in [50, 250] and [50, 100] respectively. For each configuration, we randomly generate 1000 DAG tasks to compute the average normalized bound.

**Evaluation with Task Parallelism.** We conduct experiments by changing parallelism factor $pf$ with core number $m = 16$. The results are presented in Figure 5. Since the normalized bound is always smaller than one, our method consistently outperforms *B-HE* on average, especially when the DAG task has high parallelism (when $pf$ is small), which is the typical case as benchmarks (such as [11, 15]) and practical applications generally possess high parallelism. This is because paths of a DAG with higher parallelism suffer more interference. The priority assignment enabled by our computing method can balance such interference among different paths better, thus reducing the response time bound. From experimental results, compared with *B-HE*, the improvement of response time bound is up to 18.1%. For $pf$ less than 0.2, the performance of our method is better than *B-ZHAO* on average, which further shows our method can balance interference among different paths effectively for DAG tasks with high parallelism. When $pf$ becomes larger, the DAG being more sequential, both bounds becomes closer to *B-HE*, and finally approach to the length of the longest path in the graph. This observation is also obtained in [17, 29].

In the following experiments, we randomly choose $pf$ in [0.01, 0.1] to better represent real world applications which generally possess high parallelism as mentioned above.

**Evaluation with Core Number.** The objective of this experiment is to demonstrate how sensitive the evaluated method is to core number. Figure 6 shows that our method always produces smaller bounds than *B-HE* on average and outperforms *B-HE* by up to 13.3% with $m = 12$. With core number being smaller and larger, *B-OUR* is close to *B-HE*. This is because for core number being smaller, both bounds approach $vol(G)$; for core number being larger, both bounds approach $len(G)$.

When core number is small, *B-ZHAO* performs better than our method. This is because, for a small core number, the computing resource for non-critical vertices (vertices not in the longest path) becomes scarce, which results in non-critical vertices having a large impact on the response time bound. The method for *B-ZHAO* can delicately adjust the execution

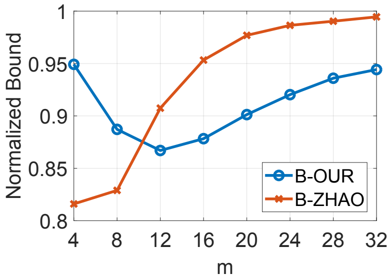


**Figure 6** Normalized bound with different $m$.

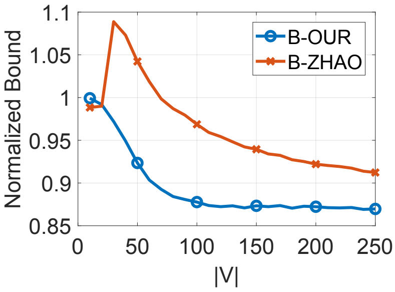


**Figure 7** Normalized bound with different $|V|$.

order of non-critical vertices, which is utilized by its analysis method subsequently, finally leading to a smaller bound. With core number increasing, our method, being able to balance interference among different paths effectively, outperforms *B-ZHAO* (e.g., by up to 7.5% with $m = 20$). Real world applications, generally possessing high parallelism [17], may require executing on computing platforms with a larger core number to meet their deadlines. What's more, nowadays mainstream computing platforms generally have a large number of cores. These facts render our method more useful and effective in practice.

Table 1 reports the percentage of inferior cases (cases where the bound *B-OUR* is larger than *B-HE* or *B-ZHAO*) during experiments of Figure 6. For example, in Table 1, for $m$ in [11, 16], no inferior case with respect to *B-HE* is observed and there are 17.38% cases where bounds computed by our method are larger than that of *B-ZHAO*. Table 1 is consistent with Figure 6: for small core numbers, *B-ZHAO* performs better; with core number increasing, our method becomes more effective. We observe that during experiments of Figure 6, the overall inferior cases are less than 0.04% with respect to *B-HE* and less than 24.38% with respect to *B-ZHAO*. This observation further demonstrates the effectiveness of the proposed method.

In the following experiments, we choose core number $m = 16$ as the representative to evaluate the performance.

**Evaluation with Vertex Number.** This experiment evaluates the sensitivity of the proposed analysis to the vertex number, and results are shown in Figure 7. For $|V| < 30$, the proposed method provides similar results to *B-HE*, which is because the parallelism of the DAG is relatively low when vertex number is small. As analysed above, our method is more effective when parallelism is relatively higher. With vertex number increasing, our method becomes more effective and outperforms *B-HE* more than 10% on average and up to 13.1% and $|V| = 240$. For almost all vertex numbers in this experiment, our method outperforms *B-ZHAO*. We also observe that for small vertex numbers, *B-ZHAO* may produce a bound larger than *B-HE*.

## 7.2 Evaluation of Multi-DAG Systems

This section evaluates the performance of our method for multi-DAG systems with constrained deadlines. All three methods in Section 7.1 extended their bounds for single DAG task to multi-DAG systems. For task level priority policy, *B-OUR* and *B-HE* can be applied to both dynamic priority (e.g., EDF) and static priority (e.g., RM), denoted as *EDF-OUR*, *RM-OUR*, *EDF-HE*, *RM-HE* respectively; *B-ZHAO* can only be applied to static priority, denoted as *RM-ZHAO*. These five methods are compared in this section.

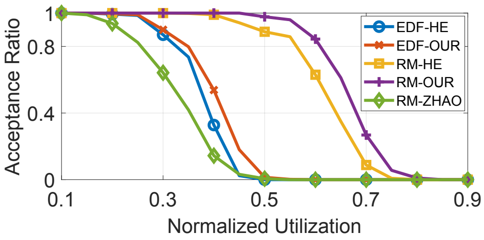


**Figure 8** Acceptance ratio with different normalized utilization ($m = 16$).

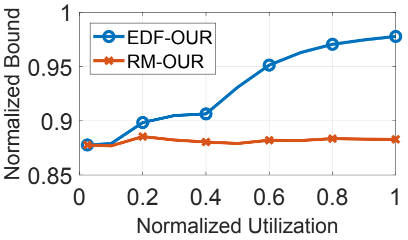


**Figure 9** Normalized bound with different normalized utilization ($m = 16$).

**Task Set Generation.** DAG tasks are generated by the same method as Section 7.1 with $pf$, $|V|$ and $c_i$ in [0.01, 0.1], [50, 250] and [50, 100] respectively. The period $T$ (which is also the deadline in the experiment) of a DAG task is randomly chosen in $[L, 6L]$, where $L$ is the length of the longest path of the task graph. To generate a task set with specific utilization, we randomly generate a DAG task and add it to the task set until the total utilization reaches the required value.

**Evaluation Using Acceptance Ratio.** We first test the schedulability of multi-DAG systems using acceptance ratio to evaluate our method. For configuration, we randomly generate 1000 task sets. From the results reported in Figure 8, the proposed method offers better schedulability than that of the state-of-the-art under all settings, especially when normalized utilization is in [0.4, 0.7]. Compared with methods in [17], the improvement of acceptance ratio for EDF and RM is up to 22.2% and 32.0% respectively. *RM-ZHAO* performs worse than *RM-HE*, the reason of which is explained in the following. First, the scheduling for task set in [29] is not work-conserving. Only when a DAG task finishes its execution completely, it can schedule another DAG task to execute. However, before a DAG task finishes its execution completely, some cores may be idle and available to execute tasks (this behavior is fundamental to its underlying response time analysis), which wastes a lot of computing resources. Second, the $(\alpha, \beta)$-pair analysis for one DAG task proposed in [29] is not incorporated into its analysis for task set, which makes its performance even worse. The acceptance ratio for *RM-ZHAO* reported in our experiments is consistent with the results reported in [29].

**Evaluation Using Normalized Bound.** The normalized bound of a task set is the average value among normalized bounds of its tasks. Even if a task set is deemed to be unschedulable, we still try to iterate until all tasks reach a fixed point to compute the response time bound for all tasks. If a task set cannot reach a fixed point, it will be discarded. As reported in Figure 8, the performance of *RM-ZHAO* is relatively poor, which results in that the response time bound cannot be computed for lots of task sets (in our experiment, the fixed point iteration procedure for computing bounds cannot converge before the iterated bounds reach thirty times of its deadlines). Therefore, *RM-ZHAO* is not included in the result of this experiment. For each configuration, we have at least 1000 task sets to compute the average normalized bound. As shown in Figure 9, since the normalized bound is always smaller than 1, our method completely dominates *B-HE* for both EDF and RM, reducing the response time bound by up to 12.3% for EDF and up to 12.4% for RM. The results are consistent with the evaluation of single-DAG systems.

**Summary.** Experiments in this section show that our computing method and priority assignment can reduce response time bound, improve system schedulability compared to the state-of-the-art by a considerable margin. Specifically, compared to [17], our method reduces response time bound by more than 10% on average and improves schedulability up to 20%. The effectiveness of the method is also supported by the number of DAG tasks with a smaller bound than the state-of-the-art.

## 8 Related Work

He et al. [17] proposed a dynamic programming algorithm to compute response time bound for DAG tasks with intra-task priority assignment, alongside its response time analysis, which is the most relevant work to this paper. Their computing method assumed that priority assignment should comply with topology constraint. For a wide range of priority assignment without this constraint, their algorithm may produce a wrong bound.

The response time analysis for multi-DAG systems has been intensively studied in recent years, with different scheduling strategies including global scheduling [3, 7, 12, 13, 22, 25] and federated scheduling [4–6, 23, 24]. All the above works involve using the response time bound of a single DAG to bound the intra-task interference, which is the focus of this paper. Fonseca et al. [14] proposed a partitioned scheduling for sporadic DAG tasks.

For response time bound of a single DAG task, zhao et al. [29] explored parallelism and dependencies in DAG structure, and proposed a priority assignment policy and response time bounds based on its CPC (concurrent provider and consumer) model. Han et al. [16] studied typed DAG task for heterogeneous multi-core platform. Sun et al. [27] proposed a method to compute the exact worst case response time with exponential time complexity while being efficient for DAG tasks with small number of vertices. Chen et al. [9] proposed a bound for a DAG with conditional branches by simulating the DAG task with a predefined execution order. The bound in [9] was proved to be timing-anomaly free.

For intra-task priority assignment, in real-time community, Voudouris et al. [28] computed response time bound by simulating the timing-anomaly free scheduler. Pathan et al. [26] proposed a method to utilize intra-task priority assignment to improve resource utilization, and used the idea of ready time and response time of vertices to reduce intra-task interference. Besides research work from real-time community, there are plenty of techniques concerning scheduling task graphs on multiprocessor platform with intra-task priority assignment. Their objective is to reduce the response time on average, not the response time bound. Works [19, 21] considered priority assignment for static scheduling algorithms for DAGs. Kwok and Ahmad proposed a static scheduling algorithm for allocating task graphs to fully connected multiprocessor based on the critical path of task graphs [20].

## 9 Conclusion, Limitations and Future Work

Computing response time bound of DAG tasks is one of the most important problems in the real-time community. In this paper, we address a serious constraint of the previous result, and propose a method capable of computing response time bound for DAG tasks with arbitrary intra-task priority assignment. Experiments show that our method can greatly reduce the response time bound. In the future, we plan to formulate the graph interference problem into a formal language, clearly identify the context-free grammar inherently associated with this problem, and utilize the automata theory [18] to compute it within a constant number of iterations (the method of this paper computing within $|V|$ iterations) and further improve efficiency. Another direction is searching for an optimal priority assignment with respect to the bound in Equation 1.

## References

1 Openmp-api-specification-5.0.pdf. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. (Accessed on 03/01/2021).

2 Theodore P Baker and Sanjoy K Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 141–150. IEEE, 2009.

3 Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *2014 26th Euromicro conference on real-time systems*, pages 97–105. IEEE, 2014.

4 Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic dag task systems. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1323–1328. IEEE, 2015.

5 Sanjoy Baruah. Federated scheduling of sporadic dag task systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 179–186. IEEE, 2015.

6 Sanjoy Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *Proceedings of the 12th International Conference on Embedded Software*, pages 1–10. IEEE Press, 2015.

7 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *2013 25th Euromicro conference on real-time systems*, pages 225–233. IEEE, 2013.

8 Alan Burns and Sanjoy Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

9 Peng Chen, Weichen Liu, Xu Jiang, Qingqiang He, and Nan Guan. Timing-anomaly free dynamic scheduling of conditional dag tasks on multi-core systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–19, 2019.

10 Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd international ICST conference on simulation tools and techniques*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

11 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 124–131. IEEE, 2009.

12 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved response time analysis of sporadic dag tasks for global fp scheduling. In *Proceedings of the 25th international conference on real-time networks and systems*, pages 28–37, 2017.

13 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 55(2):387–432, 2019.

14 José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2016.

15 Vladimir Gajinov, Srđan Stipić, Igor Erić, Osman S Unsal, Eduard Ayguadé, and Adrián Cristal. Dash: a benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In *Proceedings of the 11th ACM conference on computing frontiers*, page 4. ACM, 2014.

16 Meiling Han, Nan Guan, Jinghao Sun, Qingqiang He, Qingxu Deng, and Weichen Liu. Response time bounds for typed dag parallel tasks on heterogeneous multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2567–2581, 2019.

17 Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

18 John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

19 H KASAHARA and S NARITA. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE transactions on computers*, 33(11):1023–1029, 1984.

20 Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

21 Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

22 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Outstanding paper award: Analysis of global edf for parallel tasks. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 3–13. IEEE, 2013.

23 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

24 Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-time systems*, 53(5):760–811, 2017.

25 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221. IEEE, 2015.

26 Risat Pathan, Petros Voudouris, and Per Stenström. Scheduling parallel real-time recurrent tasks on multicore platforms. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):915–928, 2017.

27 Jinghao Sun, Feng Li, Nan Guan, Wentao Zhu, Minjie Xiang, Zhishan Guo, and Wang Yi. On computing exact wcrt for dag tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

28 Petros Voudouris, Per Stenström, and Risat Pathan. Timing-anomaly free dynamic scheduling of task-based parallel applications. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 365–376. IEEE, 2017.

29 Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *IEEE Real-Time Systems Symposium*. IEEE, 2020.

# Graceful Degradation in Semi-Clairvoyant Scheduling

**Sanjoy Baruah**
Washington University in Saint Louis, MO, USA

**Pontus Ekberg**
Uppsala University, Sweden

## Abstract

In the Vestal model of mixed-criticality systems, jobs are characterized by multiple different estimates of their actual, but unknown, worst-case execution time (WCET) parameters. Some recent research has focused upon a *semi-clairvoyant* model for mixed-criticality systems in which it is assumed that each job reveals upon arrival which of its WCET parameters it will respect. We study the problem of scheduling such semi-clairvoyant systems to ensure graceful degradation of service to less critical jobs in the event that the systems exhibit high-criticality behavior. We propose multiple different interpretations of graceful degradation in such systems, and derive efficient scheduling algorithms that are capable of ensuring graceful degradation under these different interpretations.

**2012 ACM Subject Classification** Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Real-time schedulability

**Keywords and phrases** Mixed criticality, semi-clairvoyance, graceful degradation

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.9

**Funding** *Sanjoy Baruah*: National Science Foundation Grants CNS-1814739 and CPS-1932530.
*Pontus Ekberg*: Swedish Research Council grant 2018-04446.

## 1 Introduction

A model for mixed-criticality workloads was proposed by Vestal [30] as a means of achieving timing predictability upon modern processors. In this model, individual pieces of real-time code are represented as jobs with associated deadlines that are characterized by multiple worst-case execution time (WCET) parameters. These different WCET parameters represent different estimates, made at differing levels of assurance, of the actual unknown WCET of the code. Each job is also assigned a criticality – in the two-criticality level model considered in this paper (and much of the mixed-criticality scheduling theory literature), these are called HI and LO, denoting greater and lesser criticality respectively. The two WCET parameters are determined for each job, one at a level of assurance consistent with HI criticality and a second at a level of assurance consistent with LO criticality. The correctness criterion in the Vestal mixed-criticality model is that if each job completes execution within a duration not exceeding its LO-criticality WCET estimate then all the jobs should complete execution by their respective deadlines, whereas if some jobs do not complete execution within their LO-criticality WCET estimates (but all jobs would complete execution if allowed to execute for as much as their HI-criticality WCET), then all the HI-criticality jobs should complete execution by their respective deadlines although the LO-criticality jobs may fail to do so.

**Run-time monitoring.** The methodology introduced by Vestal [30] was initially applied for the verification of timing correctness of *criticality-agnostic* scheduling algorithms: algorithms that do not seek to determine during run-time whether or not LO-criticality WCET estimates have been exceeded. It was later observed [12] that if system state were *monitored* during run-time to determine when jobs execute in excess of their LO-criticality WCETs, then

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 9; pp. 9:1–9:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

custom-designed mixed-criticality scheduling algorithms could be developed that explicitly exploit such run-time information. Several such mixed-criticality algorithms were developed – OCBP [10], MC-EDF [29], EDF-VD [5], AMC [7], MC-Fluid [26], etc. – that determine during run-time if and when the system "mode" transitions from LO-criticality mode (no job has executed beyond its LO-criticality WCET) to HI-criticality mode (some job has executed for more than its LO-criticality WCET), and adjusts its scheduling decisions accordingly.

**Forms of Clairvoyance.** The notion of *clairvoyance*, which has previously been used to quantify the effectiveness of on-line algorithms (see, e.g., [25]), forms the basis of the speedup factor metric that is widely used for quantitatively characterizing these mixed-criticality scheduling algorithms. In the context of mixed-criticality scheduling, a clairvoyant algorithm is one that knows prior to run-time whether any job is going to exceed its LO-criticality WCET or not. Generally speaking, a clairvoyant scheduling algorithm is an idealized abstraction against which to compare the performance of actual scheduling algorithms.

More recently, the concept of *semi-clairvoyance* was introduced for mixed-criticality scheduling [1] (also see [14]). A semi-clairvoyant scheduling algorithm is one that knows, at the instant of a job's arrival, whether it will complete execution within its LO-criticality WCET. Unlike [full] clairvoyance, which is a purely conceptual abstraction that is not realisable in practice, it is persuasively argued in both [1] and [14] that semi-clairvoyance is a realistic and practically useful model for certain circumstances. For instance, a system developer may provide two separate implementations of a job: upon arrival, the system determines which implementation it is appropriate to execute given the current circumstances – i.e., the current mode. (E.g., one implementation may be intended for execution under regular conditions, and another for execution under unexpected – i.e., HI criticality – conditions: the HI-criticality implementation including code to perform crisis-mitigation functionalities). As stated in [14], semi-clairvoyance is particularly applicable "when the execution time [...] depends on the state of the system at the time the job arrives, rather than on some internal property that emerges as it executes."

**Graceful degradation and mixed-criticality scheduling.** Despite its very significant impact on the real-time scheduling theory literature, Vestal's mixed-criticality workload model [30] has been criticized [20, 19, 31] for not matching systems developers' expectations in some important aspects. Our focus here is upon one such aspect: that LO-criticality jobs ought to be guaranteed some amount of execution prior to their deadlines even in HI-criticality modes. Modifications to the Vestal model have been proposed (e.g., in [13] and [18]) that allows for the specification of some degraded service for LO-criticality jobs even in HI-criticality system behaviors – we will describe such a model in Section 2. Many algorithms have been proposed (e.g., [8, 23, 24, 22]) that seek to ensure such graceful degradation for systems.

**This research.** In this paper, we study graceful degradation for semi-clairvoyant algorithms upon preemptive uniprocessors – to our knowledge, this is the first piece of research that integrates consideration of these two concepts. We consider three different notions of graceful degradation, characterized here as three different *correctness criteria*, by placing different requirements as to which LO-criticality jobs it is acceptable to provide degraded service to, upon some HI-criticality job's arrival signalling HI-criticality mode:

**CC-1.** all LO-criticality jobs that *have deadlines* after this instant;
**CC-2.** all LO-criticality jobs that *begin execution* after this instant; or
**CC-3.** all LO-criticality jobs that *arrive* after this instant.

Although the three correctness criteria appear very similar, differing only in the treatment accorded to LO-criticality jobs that span a mode-change instant, we will see that the associated schedulability analysis problems are very different. We will show that determining whether a collection of independent jobs can be scheduled correctly upon a preemptive uniprocessor can be done in polynomial time via reduction to a linear program for correctness criterion CC-1, is NP-complete in the strong sense for CC-2, and reduces to a variant of EDF-schedulability analysis for CC-3 that is also solvable in polynomial time.[1]

Correctness criterion CC-1 is consistent with standard interpretations of mixed-criticality scheduling (and most prior work). It can be a sensible criterion when LO-criticality jobs, unlike HI-criticality jobs, do not have multiple available implementations, but their execution times are instead enforced by *budgeting*. The reduced execution time of LO-criticality jobs in HI-criticality mode is then simply achieved by lowering their execution-time budgets. This can be done even to started jobs, and jobs that do not finish within their budgets are suspended or discarded.

However, there are circumstances where the other two criteria are more appropriate. Under the scenario in which multiple implementations of each job are available and the run-time system chooses which to execute based upon current system state, it is reasonable to expect, as in CC-2, that once an implementation is chosen and its execution has commenced, it must be completed.

For systems observing "*commitment upon job arrival*" semantics [16], the choice as to which available implementation of a job to execute must be made upon the job's arrival. It is reasonable to expect that for jobs arriving before the mode-change instant, this choice will favor the LO-criticality implementations – this is what CC-3 requires.

**Contributions.** We have seen above that there are very good reasons for studying all three correctness criteria, which is what we do in this research. Specifically

- We formally define our three correctness criteria in Section 2, within the framework of a workload model integrating graceful degradation and mixed criticality considerations.
- In Section 3 we present a table-based run-time scheduling algorithm for scheduling collections of independent jobs, and a fluid-based algorithm for scheduling implicit-deadline sporadic task systems, under correctness criterion CC-1. Exact schedulability tests that run in polynomial time are presented for both cases.
- In Section 4 we prove that it is NP-hard in the strong sense to determine schedulability of a collection of independent jobs under correctness criterion CC-2. We also provide a mixed integer linear program (MILP) representation of this schedulability problem: solving this MILP allows one to construct a table-based run-time scheduling algorithm.
- In Section 5 we show that EDF is an optimal scheduling algorithm under CC-3 and present exact schedulability tests for scheduling both collections of independent jobs and three-parameter sporadic task systems with bounded utilization under this correctness criterion. These tests run in polynomial and pseudo-polynomial time, respectively.
- We place all these above results within a broader context in Section 6, explaining how they fit together and how they suggest some guidelines for implementation and analysis of semi-clairvoyant systems; while the benefits of these guidelines will be quite evident, we will additionally provide a quantitative evaluation of their cost.

[1] By comparison, schedulability analysis for *non-clairvoyant* scheduling of mixed-criticality jobs under the ordinary mixed-criticality semantics (corresponding to CC-1) without graceful degradation – this is the default mixed-criticality setting seen in most previous work – is NP-hard in the strong sense [4].

## 2 Workload Model

As mentioned in Section 1 we will restrict our attention here to *dual*-criticality systems: systems with two distinct criticality levels denoted LO and HI, that are to execute upon a single preemptive processor. We consider both collections of independent jobs and recurrent (sporadic) tasks.

For **jobs**, an instance is a collection of $n$ dual-criticality jobs $\mathcal{J} = \{J_1, J_2, ..., J_n\}$. Each job $J_i$ is characterized by a tuple of parameters: $J_i = (\chi_i, a_i, [c_i(\text{LO}), c_i(\text{HI})], d_i)$ where $\chi_i \in \{\text{LO}, \text{HI}\}$ and the remaining parameters are non-negative integers, with the interpretation that

- $\chi_i$ denotes the criticality of the job;
- $a_i$ denotes its release time;
- $d_i$ denotes its deadline; and
- $c_i(\text{LO})$ and $c_i(\text{HI})$ denote LO-criticality and HI-criticality specifications of the job's worst-case execution time (WCET) parameter respectively. We require that the WCETs satisfy the constraint that

$$\big(c_i(\text{HI}) \le c_i(\text{LO}),\ \text{if } \chi_i = \text{LO}\big) \ \text{ and } \ \big(c_i(\text{HI}) \ge c_i(\text{LO}),\ \text{if } \chi_i = \text{HI}\big)$$

(I.e., a LO-criticality job receives *degraded* service whereas a HI-criticality job receives *enhanced* service in HI-criticality mode.)

For **sporadic tasks**, an instance (or system) $\tau$ is a collection of $n$ tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each $\tau_i$ is characterized by the parameters $(\chi_i, [C_i(\text{LO}), C_i(\text{HI})], D_i, T_i)$, where $\chi_i \in \{\text{LO}, \text{HI}\}$ denotes its criticality, $C_i(\text{LO})$ and $C_i(\text{HI})$ its LO and HI criticality WCETs, $D_i \in \mathbb{N}$ its relative deadline parameter, and $T_i \in \mathbb{N}$ its period. Analogously to jobs, we require that the WCETs satisfy the constraint

$$\big(C_i(\text{HI}) \le C_i(\text{LO}),\ \text{if } \chi_i = \text{LO}\big) \ \text{ and } \ \big(C_i(\text{HI}) \ge C_i(\text{LO}),\ \text{if } \chi_i = \text{HI}\big)$$

Some additional notation: Let $\tau^{(\text{LO})}$ and $\tau^{(\text{HI})}$ denote the subsets of tasks $\tau_i$ with $\chi_i = \text{LO}$ and $\chi_i = \text{HI}$, respectively. Let $U_i(\text{LO}) = C_i(\text{LO})/T_i$ and $U_i(\text{HI}) = C_i(\text{HI})/T_i$. Finally, let $U_{\text{LO}} = \sum_{\tau_i \in \tau} U_i(\text{LO})$ and $U_{\text{HI}} = \sum_{\tau_i \in \tau} U_i(\text{HI})$.

**Correctness criteria.** Any task system or collection of jobs is assumed to begin execution in LO-criticality mode, with each job requiring an amount of execution that is no greater than its LO-criticality WCET. Any HI-criticality job may signal a transition to HI-criticality mode upon its arrival. If this happens at some time-instant $t_{\text{switch}}$, then each HI-criticality job that arrives at or after $t_{\text{switch}}$ may require up to its (potentially larger) HI-criticality WCET. Likewise, each LO-criticality job with deadline at or before $t_{\text{switch}}$ requires up to its LO-criticality WCET, while each LO-criticality job that arrives at or after $t_{\text{switch}}$ can only require up to its (potentially smaller) HI-criticality WCET. As mentioned in Section 1, we consider three different notions of correctness for LO-criticality jobs that are *active* at time point $t_{\text{switch}}$, and we will see that this choice has significant consequences for how to schedule and analyse such systems. These three different correctness criteria are defined as follows.

**CC-1.** Any LO-criticality job that has its deadline after time $t_{\text{switch}}$ may only require its smaller HI-criticality WCET.

**CC-2.** Any LO-criticality job that is active and has already started executing at time $t_{\text{switch}}$ is permitted to require its larger LO-criticality WCET.

**CC-3.** Any LO-criticality job that is active at time $t_{\text{switch}}$ is permitted to require its larger LO-criticality WCET.

Here we can note that if we restrict all LO-criticality jobs/tasks to have zero-valued WCETs at HI-criticality (i.e., $c_i(\text{HI})$ and $C_i(\text{HI})$ equal zero for all LO-criticality jobs/tasks), then correctness criterion CC-1 reduces to the model that was studied in previous work on semi-clairvoyant scheduling [1, 14].

## 3 Correctness Criterion CC-1

In this section we devise algorithms for scheduling dual-criticality instances of independent jobs (Section 3.1), and implicit-deadline sporadic task systems (Section 3.2) under the requirement that all LO-criticality jobs *completing* after the arrival of a HI-criticality job that signals HI-criticality mode receive an amount of execution at least equal to their HI-criticality WCETs. The following example illustrates some of the challenges in determining whether such an instance can be correctly scheduled under semi-clairvoyant scheduling.

▶ **Example 1.** Consider an instance comprising the following three jobs:

| | $\chi_i$ | $a_i$ | $c_i(\text{LO})$ | $c_i(\text{HI})$ | $d_i$ |
|---|---|---|---|---|---|
| $J_1$ | LO | 0 | 1 | 0 | 2 |
| $J_2$ | LO | 0 | 2 | 1 | 3 |
| $J_3$ | HI | 1 | 0 | 2 | 3 |

First, we point out that this instance is clairvoyant schedulable:

- in LO-criticality mode one could execute $J_1$ over the interval $[0, 1]$, and $J_2$ over $[1, 3]$;
- in HI-criticality mode one could execute $J_2$ over $[0, 1]$, and $J_3$ over $[1, 3]$.

Observe that a different job is scheduled over $[0, 1]$ in the two modes. However under semi-clairvoyant scheduling the mode only becomes known at time-instant 1 (i.e., upon $J_3$'s arrival). An EDF schedule would choose $J_1$ over the interval $[0, 1]$; if job $J_3$ were to signal HI-criticality mode upon arrival, that would require that an execution amount equal to $c_2(\text{HI}) + c_3(\text{HI}) = (1 + 2) = \mathbf{3}$ units be completed over the next two time units. Hence EDF is not able to schedule this instance correctly; we leave it to the reader to verify that the following on-line scheduling strategy is correct:

```
Execute $J_2$ over $[0, 1]$
**if** $J_3$ signals LO-criticality mode upon its arrival
    execute $J_1$ over $[1, 2]$, and $J_2$ over $[2, 3]$
**else** // (i.e., $J_3$ signals HI-criticality mode upon its arrival)
    execute $J_3$ over $[1, 3]$
```

◢

In the remainder of this section we describe how such strategies may be determined for collections of jobs and implicit-deadline task systems. Our results for CC-1 are closely inspired by those of [1] (which effectively targets CC-1 without graceful degradation), though in order to enable graceful degradation we take a very different approach to the formulation of the linear program that we will see next.

### 3.1 Jobs

We partition the time-line by the release-dates and the deadlines (the $a_i$ and $d_i$ parameters) of the jobs. I.e., the time-line from the first release date to the last deadline is divided into subintervals by dividing it at every $a_i$ and $d_i$. Let $I_1, I_2, \ldots$ denote these subintervals and note that there are at most $2n - 1$ of them. Let $t_1, t_2, \ldots$ denote the distinct time instants at which HI-criticality jobs arrive, in order (i.e., $t_k < t_{k+1}$). There are at most $n$ such time instants $t_k$.

In the following we let $i$ range over jobs $J_i$, let $j$ range over subintervals $I_j$, and let $k$ range over the time points $t_k$ where a HI-criticality job arrives.

**Variables.** Our LP uses the following **variables**:

1. The variable $x_{ij}$ represents the amount of execution assigned to the $i$'th job $J_i$ in the $j$'th interval $I_j$, in a schedule in which no HI-criticality job signals the transition to HI-criticality mode. There are $O(n^2)$ such variables.
2. For each $k$, the variable $y_{ij}^{(k)}$ represents the amount of execution assigned to job $J_i$ in the interval $I_j$, in a schedule in which HI-criticality mode is signalled at time-instant $t_k$. There are $O(n^3)$ such variables – $O(n^2)$ for each value of $k$.

For the example instance of Example 1 there are $3 \times 3 = 9$ $x_{ij}$ variables, and the same number of $y_{ij}^{(1)}$ variables.

**Constructing Scheduling Tables.** A solution to our LP will assign values to these variables. We will use these assigned values to construct several scheduling tables prior to run-time:

- Scheduling table $S_o$ will schedule job $J_i$ for a duration $x_{ij}$ during the interval $I_j$.
- For each $k$, there will be a scheduling table $S_k$ that schedules job $J_i$ for a duration $y_{ij}^{(k)}$ during the interval $I_j$.

Our run-time scheduling strategy is then to start out making scheduling decisions according to $S_o$, and to switch to $S_k$ if HI-criticality mode is first signalled by the arrival of a HI-criticality job at $t_k$. The Constraints in Eq. 1 below enforce that $y_{ij}^{(k)} = x_{ij}$ for all intervals $I_j$ before $t_k$; hence in the event that the arrival of a HI-criticality job at time $t_k$ signals HI-criticality mode we are effectively following scheduling table $S_k$ from the very beginning.

▶ **Example 2.** For the 3-job example instance in Example 1, there are three subintervals $I_1 = [0, 1]$, $I_2 = [1, 2]$, and $I_3 = [2, 3]$. Since there is only one HI-criticality job, we have $t_1 = 1$ as the only instant at which HI-criticality jobs arrive. Below is a possible assignment of values for $x_{ij}$ and $y_{ij}^{(1)}$ for this example instance (these happen to be the $x_{ij}$ and $y_{ij}^{(1)}$ values corresponding to the correct scheduling strategy described in Example 1):

| | $j = 1$ | $j = 2$ | $j = 3$ |
|---|---|---|---|
| $i = 1$ | $x_{11} = 0, y_{11}^{(1)} = 0$ | $x_{12} = 1, y_{12}^{(1)} = 0$ | $x_{13} = 0, y_{13}^{(1)} = 0$ |
| $i = 2$ | $x_{21} = 1, y_{21}^{(1)} = 1$ | $x_{22} = 0, y_{22}^{(1)} = 0$ | $x_{23} = 1, y_{23}^{(1)} = 0$ |
| $i = 3$ | $x_{31} = 0, y_{31}^{(1)} = 0$ | $x_{32} = 0, y_{32}^{(1)} = 1$ | $x_{33} = 0, y_{33}^{(1)} = 1$ |

The following scheduling tables are constructed from these $x_{i,j}$ and $y_{i,j}$ values:

1. $S_o$ schedules $J_2$ over $I_1 = [0, 1]$, $J_1$ over $I_2 = [1, 2]$, and $J_2$ over $I_3 = [2, 3]$.
2. $S_1$ schedules $J_2$ over $I_1$, and $J_3$ over both $I_2$ and $I_3$.

During run-time we start out scheduling according to $S_o$; if job $J_3$ signals HI-criticality mode upon arrival at time-instant 1 then we subsequently switch to schedule $S_1$. ⌟

**Constraints.** We will now describe the **constraints** added to our LP in order to ensure that the variables defined above have their intended interpretations.

- (Non-clairvoyance.) For each $k$, for all $i$, and for all $j$ such that the interval $I_j$ completes no later than time-instant $t_k$, we have

$$y_{ij}^{(k)} = x_{ij} \tag{1}$$

  There are $O(n^3)$ such constraints.
- (Correctness in LO criticality.) For each $i$

$$\sum_{I_j \subseteq [r_i, d_i]} x_{ij} \geq c_i(\text{LO}) \tag{2}$$

There are $O(n)$ such constraints – one per job.

- (Correctness in HI criticality.) For each $k$ (HI-criticality signalled at $t_k$)
  - For each $i$ for which $\chi_i = \text{HI}$

$$\sum_{I_j \subseteq [r_i, d_i]} y_{ij}^{(k)} \geq \begin{cases} c_i(\text{LO}) & \text{if } r_i < t_k \\ c_i(\text{HI}) & \text{if } r_i \geq t_k \end{cases} \tag{3}$$

    There are $O(n^2)$ such constraints.

  - For each $i$ for which $\chi_i = \text{LO}$

$$\sum_{I_j \subseteq [r_i, d_i]} y_{ij}^{(k)} \geq \begin{cases} c_i(\text{LO}) & \text{if } d_i \leq t_k \\ c_i(\text{HI}) & \text{if } d_i > t_k \end{cases} \tag{4}$$

    There are $O(n^2)$ such constraints.

  Notice the difference between Expression 3 and Expression 4: the first case in Expression 3 applies to all jobs that *arrive* before $t_k$; in Expression 4, to all jobs that *have deadlines* no later than $t_k$.

- (Adequate computing capacity to construct scheduling table $S_o$.) For each $j$

$$\sum_i x_{ij} \leq |I_j|, \tag{5}$$

  where $|I|$ denotes the length of an interval $I$. There are $O(n)$ such constraints.

- (Adequate computing capacity to construct scheduling table $S_k$.) For each $k$, for each $j$

$$\sum_i y_{ij}^{(k)} \leq |I_j| \tag{6}$$

  There are $O(n^2)$ such constraints.

## 3.2 Tasks

We now give an optimal algorithm for scheduling systems of *implicit-deadline* sporadic tasks[2] – task systems in which the relative deadline parameter $D_i$ of each task $\tau_i$ is equal to its period parameter $T_i$. Our algorithm is based on the fluid scheduling paradigm. Such algorithms are allowed to assign individual tasks a fraction $\leq 1$ of a processor (rather than an entire processor, or none) at each instant in time. The MC-Fluid non-clairvoyant scheduling algorithm [26, 9] was designed for scheduling dual-criticality implicit-deadline sporadic task systems upon identical multiprocessor platforms. Prior to run-time, MC-Fluid computes LO-criticality and HI-criticality *execution rates* $\theta_i(\text{LO})$ and $\theta_i(\text{HI})$ for each task $\tau_i \in \tau$. Each task $\tau_i$ is initially scheduled at a rate $\theta_i(\text{LO})$; if any job does not complete despite having executed for its LO-criticality WCET, all LO-criticality tasks are immediately discarded and each HI-criticality task $\tau_i$ henceforth executes at a rate $\theta_i(\text{HI})$. An algorithm for computing suitable values for the $\theta_i(\text{LO})$ and $\theta_i(\text{HI})$ parameters is presented in [26], and a somewhat simpler algorithm subsequently derived in [9], and shown to be speedup-optimal[3], with speedup factor $\frac{4}{3}$.

[2] We do not yet have an algorithm for scheduling task systems in which the $D_i$ and $T_i$ parameters of individual tasks may differ – to our knowledge, there are no non-trivial speedup-competitive prior results known for semi-clairvoyant scheduling of task systems that are not implicit-deadline. In Section 6 we will describe how a correct but non-optimal algorithm may be obtained for scheduling such systems.

[3] The reader is referred to [25, 15] for in-depth discussions about speedup factors.

The non-clairvoyant MC-Fluid algorithm is easily modified to form an optimal semi-clairvoyant algorithm for dual criticality implicit-deadline tasks upon uniprocessors. Observe first that for an implicit-deadline sporadic task system $\tau$ to be schedulable by any scheduler (including a clairvoyant one), it is necessary that the following condition hold:

$$U_{\text{LO}} \leq 1 \text{ and } U_{\text{HI}} \leq 1 \tag{7}$$

The schedulability test associated with our optimal semi-clairvoyant scheduling algorithm is straightforward: any task system $\tau$ satisfying the condition above will be correctly scheduled by our algorithm. The algorithm assigns the $\theta_i(\text{LO})$ and $\theta_i(\text{HI})$ execution rates to each task $\tau_i$ as follows:

$$\Big(\theta_i(\text{LO}) = U_i(\text{LO})\Big) \text{ and } \Big(\theta_i(\text{HI}) = U_i(\text{HI})\Big)$$

(We point out that therefore $\theta_i(\text{HI}) \leq \theta_i(\text{LO})$ for LO-criticality tasks while $\theta_i(\text{HI}) \geq \theta_i(\text{LO})$ for HI-criticality tasks.) It initially executes each task $\tau_i$ at a rate $\theta_i(\text{LO})$; if any HI-criticality job signals a transition to HI-criticality mode upon arrival, the algorithm subsequently executes each task $\tau_i$ at a rate $\theta_i(\text{HI})$. It is evident that this algorithm is feasible upon a uniprocessor since the rates of all the tasks both before and after such a transition sum to $\leq 1$.

**Proof of Correctness.** We show that any task system $\tau$ satisfying the necessary schedulability conditions in Eq. 7 is scheduled correctly as per correctness criterion CC-1.

It is evident that all tasks execute correctly in all LO-criticality behaviors (since each job of each task $\tau_i$ receives a total execution $\theta_i(\text{LO}) \times T_i = U_i(\text{LO}) \times T_i = C_i(\text{LO})$ units of execution). Consider now some HI-criticality behavior, and let $t_{\text{switch}}$ denote the instant at which HI-criticality behavior is signalled. It is evident that any job that has both its arrival time and its deadline $\leq t_{\text{switch}}$, as well as any job that has both its arrival time and its deadline $\geq t_{\text{switch}}$, receives adequate execution. It remains to consider jobs that arrive before, but have deadline after, time-instant $t_{\text{switch}}$.

**HI-criticality tasks.** Under semi-clairvoyant scheduling, all such HI-criticality jobs, having arrived before HI-criticality mode was signalled, are guaranteed to complete upon having executed for no more than their LO-criticality WCETs. Since $\theta_i(\text{LO}) \leq \theta_i(\text{HI})$ for any HI-criticality task $\tau_i$, any job of such a $\tau_i$ clearly receives at least $\theta_i(\text{LO}) \times T_i = C_i(\text{LO})$ units of execution by its deadline.

**LO-criticality tasks.** Symmetrically to the case above, under correctness criterion CC-1 all such LO-criticality jobs are to be assigned an amount of execution not exceeding their HI-criticality WCETs. Since $\theta_i(\text{LO}) \geq \theta_i(\text{HI})$ for any LO-criticality task $\tau_i$, any job of such a $\tau_i$ receives at least $\theta_i(\text{HI}) \times T_i = C_i(\text{HI})$ units of execution by its deadline.

## 4 Correctness Criterion CC-2

LO-criticality jobs that begin executing before a mode-transition has been signalled are required to execute for their (larger) LO-criticality WCETs under correctness criterion CC-2. It turns out that establishing the schedulability of systems under this correctness criterion is a computationally harder problem than under correctness criterion CC-1: while we saw above (Section 3.1) that schedulability analysis of collections of jobs can be done in polynomial time under CC-1, we now show that this is NP-complete in the strong sense under CC-2.

▶ **Theorem 3.** *Under correctness criterion CC-2, it is NP-hard in the strong sense to determine whether an instance of jobs can be correctly scheduled upon a preemptive uniprocessor.*

**Proof.** We will give a reduction from the 3-Partition problem, which is well known to be NP-complete in the strong sense [21]. An instance of 3-Partition is a set $S = \{s_1, s_2, \ldots, s_{3m}\}$ of $3m$ positive integers, such that $\sum_{s_i \in S} s_i = mk$ for some integer $k$. We are asked whether $S$ can be partitioned into $m$ disjoint subsets, such that each subset sums to $k$.

Let an instance $S = \{s_1, s_2, \ldots, s_{3m}\}$ of 3-Partition be given. We create the following set $\mathcal{J}$ of jobs.

$$J_i = (\text{LO}, 0, [2s_i, s_i], 2mk), \qquad \text{for } 1 \le i \le 3m$$

$$J_{3m+j} = (\text{HI}, 2jk, [0, k], (2j+1)k), \qquad \text{for } 1 \le j < m.$$

It is clear that this reduction can be carried out in polynomial time and that the values of the produced numerical parameters are polynomially related to those given. In the following we show that $\mathcal{J}$ is schedulable under correctness criterion CC-2 if and only if $S$ can be partitioned into subsets $S_0, S_1, \ldots, S_{m-1}$ with the same sum. We consider the two directions separately. Let $\mathcal{J}_{\text{LO}}$ and $\mathcal{J}_{\text{HI}}$ denote the sets of LO- and HI-criticality jobs in $\mathcal{J}$, respectively.

$S$ can be partitioned $\Rightarrow \mathcal{J}$ is schedulable: Assume $S$ can be partitioned into $S_0, S_1, \ldots, S_{m-1}$ such that each partition sums to $k$. Let $\mathcal{J}_j$ denote the set of jobs $J_i \in \mathcal{J}_{\text{LO}}$ such that $s_i \in S_j$, for $0 \le j < m$. That is, $J_i \in \mathcal{J}_j$ if and only if $s_i \in S_j$.

The following scheduling strategy will be successful: We allocate time interval $[0, 2k)$ for the jobs in $\mathcal{J}_0$ where they are executed in any order. Then for all $j$ such that $1 \le j < m$ we allocate time interval $[2jk, 2(j+1)k)$ for first executing HI-criticality job $J_{3m+j}$ and then the jobs in $\mathcal{J}_j$ in any order. If the jobs that are allocated to a time interval have finished early, then we idle the processor until the start of the next allocated time interval.

We note that the total execution time of the jobs in $\mathcal{J}_0$ is at most $\sum_{J_i \in \mathcal{J}_0} c_i(\text{LO}) = \sum_{s_i \in S_0} 2s_i = 2k$, and therefore they can all finish in their allocated time interval $[0, 2k)$.

We then note that for $1 \le j < m$, if neither $J_{3m+j}$ nor any previous HI-criticality job has signaled HI-criticality behavior, then the LO-criticality jobs in $\mathcal{J}_j$ have the entire time interval $[2jk, 2(j+1)k)$ to execute in, which is enough since $\sum_{J_i \in \mathcal{J}_j} c_i(\text{LO}) = 2k$. If, on the other hand, $J_{3m+j}$ or some previous HI-criticality job has signaled HI-criticality behavior, then the jobs in $\mathcal{J}_j$ might only have the time interval $[(2j+1)k, 2(j+1)k)$ to execute in, but this is still enough as they then only need to execute for up to $\sum_{J_i \in \mathcal{J}_j} c_i(\text{HI}) = k$.

The HI-criticality tasks $J_{3m+j}$, for $1 \le j < m$, all execute first in their allocated time intervals. Those time intervals start at the arrival time of the corresponding HI-criticality job, so the HI-criticality jobs will always finish no later than their deadlines.

$S$ cannot be partitioned $\Rightarrow \mathcal{J}$ is unschedulable: Assume that $S$ cannot be partitioned into $m$ subsets of equal sum. We will show that no matter what scheduling decisions are taken, there will always exist some runtime behaviors that lead to a deadline miss. In the following we consider only behaviors where each job $J_i$ requires an execution time of either exactly $c_i(\text{LO})$ or $c_i(\text{HI})$. We then note that no job has a deadline later than $2mk$ and that $\sum_{J_i \in \mathcal{J}_{\text{LO}}} c_i(\text{LO}) = 2mk$. Therefore, no idle time can possibly be allowed in a successful schedule as long as HI-criticality behavior has not been signaled. In time interval $[0, 2k)$, LO-criticality jobs must then be scheduled for the entire $2k$ time units. Let $\mathcal{J}_{\text{LO}}^{\text{start}}$ be the jobs that begin execution in $[0, 2k)$, and let $x = \sum_{J_i \in \mathcal{J}_{\text{LO}}^{\text{start}}} c_i(\text{LO})$. We consider two cases.

**Case 1 ($x > 2k$)** Suppose the HI-criticality job $J_{3m+1}$ that arrives at time $2k$ signals HI-criticality behavior, and that this and all following HI-criticality jobs require their HI-criticality WCETs. The total execution time of the HI-criticality jobs is then $(m-1)k$, and the total time left over for the LO-criticality jobs until their deadline at $2mk$ is $2mk - 2k - (m-1)k = (m-1)k$. However, the LO-criticality jobs that started execution before time $2k$ require their larger LO-criticality WCET. The total remaining execution time requirement of the LO-criticality jobs is then

$$\begin{aligned}\sum_{J_i \in \mathcal{J}_{\text{LO}} \setminus \mathcal{J}_{\text{LO}}^{\text{start}}} c_i(\text{HI}) + \sum_{J_i \in \mathcal{J}_{\text{LO}}^{\text{start}}} c_i(\text{LO}) - 2k &= \sum_{J_i \in \mathcal{J}_{\text{LO}}} c_i(\text{HI}) + \sum_{J_i \in \mathcal{J}_{\text{LO}}^{\text{start}}} c_i(\text{HI}) - 2k \\ &= mk + \frac{x}{2} - 2k \\ &> (m-1)k,\end{aligned}$$

and they can not all finish before their deadline.

**Case 2 ($x = 2k$)** Since $\sum_{J_i \in \mathcal{J}_{\text{LO}}^{\text{start}}} c_i(\text{LO}) = 2k$, the jobs in $\mathcal{J}_{\text{LO}}^{\text{start}}$ correspond to a subset of $S$ that sums to $k$. In this case, let the HI-criticality job $J_{3m+1}$ that arrives at time $2k$ require zero execution time and therefore *not* signal HI-criticality behavior.

If the second case holds, we simply note that the LO-criticality jobs that were executed corresponds to a subset of $S$ that sums to $k$ and repeat the same argument, but starting from time $2k$ instead. If again the second case holds, we note that the rest of $S$ again has a subset that sums to $k$ and repeat the argument from time $4k$ and so on. Since by assumption $S$ cannot be partitioned completely into subsets that each sum to $k$, any scheduler must eventually either idle the processor or behave according to Case 1 above, both of which can then lead to a deadline miss. ◀

**An MILP for schedulability analysis under CC-2.** Theorem 3 above establishes that we cannot analyze schedulability of collections of jobs under CC-2 even in pseudo-polynomial time (assuming P≠NP). This means that we cannot solve it with a polynomially-sized LP, but below we adapt the LP obtained in Section 3.1 for correctness criterion CC-1 to make it applicable to schedulability analysis under correctness criterion CC-2 by introducing some additional *binary integer variables*, which in effect turns the LP into a mixed-integer linear program (MILP). Since it is known [11] that determining whether an MILP has a feasible solution is in NP, the existence of this MILP also serves to show that schedulability analysis of collections of jobs under correctness criterion CC-2 is in the complexity class NP. This fact, in conjunction with Theorem 3, establishes that the problem is NP-complete.

Observe that the difference between correctness criteria CC-1 and CC-2 is in the execution time that LO-criticality jobs may require when some HI-criticality job signals HI-criticality behavior. This was captured by the constraints in Eq. 4 for CC-1; reproduced below:
For each $i$ for which $\chi_i = \text{LO}$

$$\sum_{I_j \subseteq [r_i, d_i]} y_{ij}^{(k)} \geq \begin{cases} c_i(\text{LO}) & \text{if } d_i \leq t_k \\ c_i(\text{HI}) & \text{if } d_i > t_k \end{cases}$$

We replace the constraints in Eq. 4 by the following set of constraints, keeping the other constraints of the LP in Section 3.1 as they are.

For each LO-criticality job $J_i$ and time instant $t_k$ where some HI-criticality job arrives, let $b_i^{(k)}$ be a new binary (i.e., 0–1 valued) integer variable. The intended interpretation is that job $J_i$ has started execution before $t_k$ in schedule $S_o$ if and only if $b_i^{(k)} = 1$. Instead of the constraints in Eq. 4 we add the following constraints.
For each $k$ (HI-criticality signalled at $t_k$) and each $i$ for which $\chi_i = \text{LO}$:

- If $d_i \leq t_k$ (i.e., the entire job must be scheduled by time-instant $t_k$),

$$\sum_{I_j \subseteq [r_i, d_i]} y_{ij}^{(k)} \geq c_i(\text{LO}) \tag{8}$$

- If $r_i > t_k$ (i.e., the entire job must be scheduled after time-instant $t_k$),

$$\sum_{I_j \subseteq [r_i, d_i]} y_{ij}^{(k)} \geq c_i(\textsc{hi}) \tag{9}$$

- Otherwise (i.e., the job spans $t_k$),

$$\sum_{I_j \subseteq [r_i, d_i]} y_{ij}^{(k)} \geq c_i(\textsc{lo}) \times b_i^{(k)} \tag{10}$$

$$\sum_{I_j \subseteq [t_k, d_i]} y_{ij}^{(k)} \geq c_i(\textsc{hi}) \times (1 - b_i^{(k)}) \tag{11}$$

$$\sum_{I_j \subseteq [r_i, t_k]} y_{ij}^{(k)} \leq M \times b_i^{(k)}, \tag{12}$$

where $M$ is some large enough positive constant (e.g., $M = \max_i(c_i(\textsc{lo}))$ can be used). We note that if $b_i^{(k)} = 1$, then the constraint in Eq. 10 "forces" job $J_i$ to be allocated at least $c_i(\textsc{lo})$ units of execution, while the constraints in Eqs. 11 and 12 are trivially satisfied. If instead $b_i^{(k)} = 0$, then the constraint in Eq. 10 is always satisfied, while the constraint in Eq. 11 forces $J_i$ to be allocated at least $c_i(\textsc{hi})$ units of execution after $t_k$, and the constraint in Eq. 12 forces $J_i$ to not have begun execution before $t_k$.

## 5 Correctness Criterion CC-3

Under correctness criterion CC-3, it is required that all LO-criticality jobs arriving before mode-change is signalled receive an amount of service equal to their (larger) LO-criticality WCETs. It is easily seen that unlike with regards to correctness criteria CC-1 and CC-2, under correctness criterion CC-3 *the WCET of each job is known upon the job's arrival* regardless of whether some future job will signal HI-criticality mode upon arrival or not. And it follows from the optimality property of the Earliest Deadline First scheduling algorithm (EDF) upon preemptive uniprocessor platforms [27, 17] that any collection of such jobs that can be scheduled to all complete by their deadlines is scheduled to all complete by their deadlines by EDF. Therefore, under correctness criterion CC-3 *EDF is an optimal run-time algorithm*. This is in contrast to correctness criteria CC-1 and CC-2, for neither of which do we have a general, efficient, run-time scheduling algorithm. In the remainder of this section we derive efficient schedulability-analysis for the EDF scheduling of dual-criticality instances of independent jobs (Section 5.1), and three-parameter sporadic task systems (Section 5.2) under this correctness criterion.

### 5.1 Jobs

In Sections 3.1 and 4, we had solved a linear program and an MILP respectively in order to construct scheduling tables for the run-time scheduling of collections of jobs subject to correctness criteria CC-1 and CC-2. Such an approach is not necessary for CC-3: as observed above, we know that EDF is an optimal algorithm for scheduling instances subject to correctness criterion CC-3. An associated schedulability test is easily obtained: simply simulate the EDF scheduling of the instance multiple times, once assuming LO-criticality behavior and once each under the assumption that each individual HI-criticality job is the one that signals transition to HI-criticality behavior.[4] For an instance with $n$ jobs, $O(n)$ such simulations of EDF need to be performed; each can be done in $O(n \log n)$ time [28], yielding an overall complexity of $O(n^2 \log n)$ for the schedulability test.

## 5.2 Tasks

As stated above, EDF is an optimal run-time scheduling algorithm under correctness criterion CC-3. We now derive an exact EDF schedulability test for 3-parameter sporadic task systems under this correctness criterion; our test holds for “arbitrary-deadline” systems – i.e., systems in which tasks may have relative deadlines smaller than, equal to, or larger than their periods. We will show that our schedulability test has pseudo-polynomial running time upon systems in which $\max(U_{\text{LO}}, U_{\text{HI}})$ is a priori bounded by some constant $c < 1$.

**Demand bound function.** Let $\text{DBF}_i(t, s)$ denote the *demand bound function* [2] (see [3, Chapter 10.3] for a text-book description) of task $\tau_i$ in an interval of length $t$, where HI-criticality mode is first signalled $s$ time units into the interval (possibly by some other task). That is, the function $\text{DBF}_i(t, s)$ bounds the maximum sum of execution times of jobs from $\tau_i$ that have both release times and deadlines within any such interval.

Let $t$ and $s$ be given. We make the following observations (illustrated in Figure 1).

- For a HI-criticality task $\tau_i$, the execution demand is maximized when as many jobs as possible fit into the interval, and as many of those as possible are released at or after the signalling of HI-criticality mode, and therefore can have the larger WCET $C_i(\text{HI})$. This corresponds to a scenario where one job from $\tau_i$ has its deadline at the end of the interval, and the previous jobs are each released as late as possible.
- For a LO-criticality task $\tau_i$, the execution demand is instead maximized when the maximum number of jobs from $\tau_i$ fit into the interval, but as many as possible are released before the time instant where HI-criticality mode is signalled, and therefore can have the larger WCET $C_i(\text{LO})$. This corresponds to a scenario where one job is released at the start of the interval and subsequent jobs as early as possible.

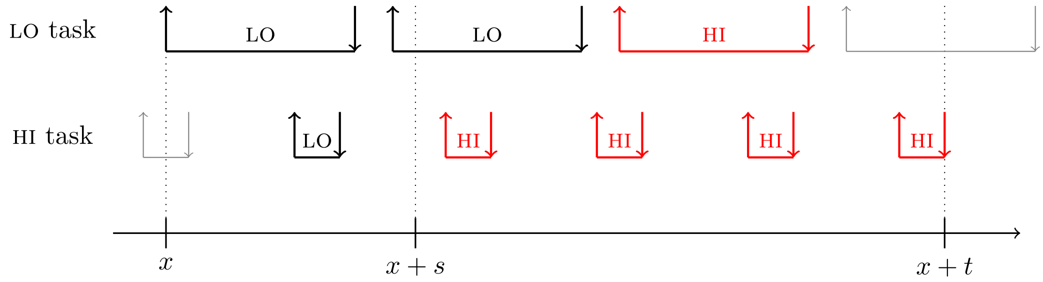


**Figure 1** An interval of length $t$ with HI-criticality mode revealed $s$ time units into the interval. The LO-criticality task maximizes execution demand within the interval by fitting two jobs with WCET $C(\text{LO})$ and one job with WCET $C(\text{HI})$. The HI-criticality task maximizes demand by fitting four jobs with WCET $C(\text{HI})$ and one with WCET $C(\text{LO})$.

We use the above observations to express $\text{DBF}_i(t, s)$. Let $\psi_i(t)$ denote the maximum number of jobs of $\tau_i$ that both arrive in, and have their deadlines within, any contiguous

[4] It follows from the *sustainability* property of EDF [6] that each such simulation can be done assuming that each job executes to exactly its WCET.

interval of duration $t$; it is known [2] that

$$\psi_i(t) = \max\left(\left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1, 0\right).$$

For a HI-criticality task $\tau_i$, a total of $\psi_i(t)$ jobs can fit inside the interval, and a total of $\psi_i(t-s)$ of those jobs can have an execution time requirement of $C_i(\text{HI})$. We therefore have

$$\text{DBF}_i(t,s) \;=\; \psi_i(t) \times C_i(\text{LO}) + \psi_i(t-s) \times (C_i(\text{HI}) - C_i(\text{LO})).$$

For a LO-criticality task $\tau_i$ the number of jobs that can fit in the interval is also at most $\psi_i(t)$. No more than $\lfloor s/T_i \rfloor + 1$ of those jobs can be released before[5] the instant when HI-criticality behavior is first signaled, and therefore have the larger execution time requirement $C_i(\text{LO})$. We have

$$\text{DBF}_i(t,s) \;=\; \psi_i(t) \times C_i(\text{HI}) + \min\left(\psi_i(t), \left\lfloor \frac{s}{T_i} \right\rfloor + 1\right) \times (C_i(\text{LO}) - C_i(\text{HI})).$$

Putting the above together, we have the following expression to bound the maximum total execution time demand of jobs from task $\tau_i$ in an interval of size $t$, where HI-criticality behavior is first revealed $s$ time units into the interval.

$$\text{DBF}_i(t,s) = \begin{cases} \psi_i(t) \times C_i(\text{LO}) + \psi_i(t-s) \times (C_i(\text{HI}) - C_i(\text{LO})), & \text{if } \chi_i = \text{HI} \\ \psi_i(t) \times C_i(\text{HI}) + \min\big(\psi_i(t), \lfloor \frac{s}{T_i} \rfloor + 1\big) \times (C_i(\text{LO}) - C_i(\text{HI})), & \text{if } \chi_i = \text{LO} \end{cases} \tag{13}$$

## The Schedulability Test

We assume here that $\max(U_{\text{LO}}, U_{\text{HI}}) < 1$. Before establishing the schedulability test we present three lemmas. Let $B$ and $\mathcal{S}(t)$ be defined for any task system $\tau$ as follows.

$$B \;=\; \frac{\sum_{\tau_i \in \tau} C_i(\chi_i)}{1 - \max(U_{\text{LO}}, U_{\text{HI}})},$$
$$\mathcal{S}(t) \;=\; \bigcup_{\tau_i \in \tau^{(\text{HI})}} \{t - kT_i - D_i \mid 0 \le k < \psi_i(t)\} \cup \{t\}.$$

As we will see in the following, $B$ is the upper bound for the values of $t$ that we need to consider when using $\text{DBF}_i(t,s)$ for a schedulability test, and $\mathcal{S}(t)$ is the set of values for $s$ that needs to be considered for each $t$. In our first lemma we show that the demand bound function for a task set is maximized at some value of $s \in \mathcal{S}(t)$, which corresponds to a release of a job from a HI-criticality task when its jobs are aligned as in Figure 1.

▶ **Lemma 4.** *Let $t$ and $s$ be given, where $t > 0$ and $s \in [0,t]$. Then there exists $s' \in \mathcal{S}(t)$ such that $\sum_{\tau_i \in \tau} \text{DBF}_i(t,s) \;\le\; \sum_{\tau_i \in \tau} \text{DBF}_i(t,s')$.*

**Proof.** Let $s'$ be the smallest $s' \in \mathcal{S}(t)$ such that $s' \ge s$.

For all $\tau_i \in \tau^{(\text{HI})}$, the set of values $s'$ where $\psi_i(t-s')$ is discontinuous in the interval $[0,t]$ is a subset of $\mathcal{S}(t)$. As $\psi_i$ is a right-continuous step function, we must have $\psi_i(t-s) = \psi_i(t-s')$. Hence, $\text{DBF}_i(t,s) = \text{DBF}_i(t,s')$ if $\chi_i = \text{HI}$.

[5] In fact, this captures the maximum number of jobs that can be released before *or at* the time point where HI-criticality is signaled. It seems as if we should replace $\lfloor s/T_i \rfloor + 1$ by $\lceil s/T_i \rceil$, but we will see later that this is in fact the suitable formulation in order to use the demand bound function for an efficient schedulability test that is both sufficient and necessary. Specifically, this formulation is required for Lemma 6, but we will see in Theorem 7 that it does not detract from the exactness of the test.

If $\tau_i \in \tau^{(\text{LO})}$, then $\text{DBF}_i(t,s)$ in non-decreasing in $s$. Therefore $\text{DBF}_i(t,s) \le \text{DBF}_i(t,s')$ if $\chi_i = \text{LO}$, which completes the proof. ◀

Our second lemma puts an upper bound on the values of $t$ that need to be considered.

▶ **Lemma 5.** *If $t \ge B$, then $\sum_{\tau_i \in \tau} \text{DBF}_i(t,s) \le t$ for all $s \in [0,t]$.*

**Proof.** Take any $\tau_i \in \tau$. We first note that $\psi_i(x) \le x/T_i + 1$ and then consider two cases. Case 1 ($\chi_i = \text{HI}$): We must have

$$\begin{aligned}
\text{DBF}_i(t,s) &\le \left(\frac{t}{T_i} + 1\right) \times C_i(\text{LO}) + \left(\frac{t-s}{T_i} + 1\right) \times (C_i(\text{HI}) - C_i(\text{LO})) \\
&= \frac{C_i(\text{LO})}{T_i} \times t + C_i(\text{LO}) + \frac{C_i(\text{HI}) - C_i(\text{LO})}{T_i} \times (t-s) + C_i(\text{HI}) - C_i(\text{LO}) \\
&= U_i(\text{HI}) \times (t-s) + U_i(\text{LO}) \times s + C_i(\text{HI}).
\end{aligned}$$

Case 2 ($\chi_i = \text{LO}$): Similarly, we have

$$\begin{aligned}
\text{DBF}_i(t,s) &\le \left(\frac{t}{T_i} + 1\right) \times C_i(\text{HI}) + \left(\frac{s}{T_i} + 1\right) \times (C_i(\text{LO}) - C_i(\text{HI})) \\
&= \frac{C_i(\text{HI})}{T_i} \times t + C_i(\text{HI}) + \frac{C_i(\text{LO}) - C_i(\text{HI})}{T_i} \times s + C_i(\text{LO}) - C_i(\text{HI}) \\
&= U_i(\text{HI}) \times (t-s) + U_i(\text{LO}) \times s + C_i(\text{LO}).
\end{aligned}$$

In both cases we then have

$$\text{DBF}_i(t,s) \le U_i(\text{HI}) \times (t-s) + U_i(\text{LO}) \times s + C_i(\chi_i)$$

and therefore

$$\begin{aligned}
\sum_{\tau_i \in \tau} \text{DBF}_i(t,s) &\le U_{\text{HI}} \times (t-s) + U_{\text{LO}} \times s + \sum_{\tau_i \in \tau} C_i(\chi_i) \\
&\le \max(U_{\text{LO}}, U_{\text{HI}}) \times t + \sum_{\tau_i \in \tau} C_i(\chi_i).
\end{aligned}$$

But if $t \ge B$, then

$$\sum_{\tau_i \in \tau} \text{DBF}_i(t,s) \le \max(U_{\text{LO}}, U_{\text{HI}}) \times t + \sum_{\tau_i \in \tau} C_i(\chi_i) \le t,$$

which concludes the proof. ◀

Our third lemma puts the above two together to limit the values of both $t$ and $s$ that we must consider.

▶ **Lemma 6.** *If $\sum_{\tau_i \in \tau} \text{DBF}_i(t,s) > t$ for some $t > 0$ and $s \in [0,t]$, then there exists $t' \in \{0, 1, \ldots, \lfloor B \rfloor\}$ and $s' \in \mathcal{S}(t')$ such that $\sum_{\tau_i \in \tau} \text{DBF}_i(t',s') > t'$.*

**Proof.** Assume that $\sum_{\tau_i \in \tau} \text{DBF}_i(t,s) > t$ for some $t > 0$ and $s \in [0,t]$. By Lemma 4 there exists $s' \in S(t)$ such that $\sum_{\tau_i \in \tau} \text{DBF}_i(t,s') \ge \sum_{\tau_i \in \tau} \text{DBF}_i(t,s)$.

Let $t' = \lfloor t \rfloor$ and $s'' = \lfloor s' \rfloor$. We note then that for any task $\tau_i$, we have $\psi_i(t) = \psi_i(t')$ as $\psi_i$ is a right-continuous step function that only changes value at integers. Further, we note that by definition of $\mathcal{S}(t)$ it must be the case that the fractional parts of $t$ and $s'$

are the same. We must then have $t - s' = t' - s''$ and $\psi_i(t - s') = \psi_i(t' - s'')$. Also, we note that as $T_i$ is integer we must have $\lfloor s/T_i \rfloor = \lfloor s'/T_i \rfloor$. From Eq. 13 it follows then that $\sum_{\tau_i \in \tau} \text{DBF}_i(t, s') = \sum_{\tau_i \in \tau} \text{DBF}_i(t', s'')$.

Finally, by Lemma 4 there must exist $s''' \in \mathcal{S}(t')$ such that $\sum_{\tau_i \in \tau} \text{DBF}_i(t', s'') \leq \sum_{\tau_i \in \tau} \text{DBF}_i(t', s''')$. Putting the above together we have

$$\begin{aligned}
\sum_{\tau_i \in \tau} \text{DBF}_i(t', s''') &\geq \sum_{\tau_i \in \tau} \text{DBF}_i(t', s'') \\
&= \sum_{\tau_i \in \tau} \text{DBF}_i(t, s') \\
&\geq \sum_{\tau_i \in \tau} \text{DBF}_i(t, s) \\
&> t \\
&\geq t'
\end{aligned}$$

By Lemma 5 we must have $t < B$ and therefore $t' \in \{0, 1, \ldots, \lfloor B \rfloor\}$. As $s''' \in \mathcal{S}(t')$, the lemma follows. ◀

We can now establish the schedulability test.

▶ **Theorem 7.** *Let $\tau$ be a task set of arbitrary-deadlines sporadic mixed-criticality tasks with $\max(U_{\text{LO}}, U_{\text{HI}}) < 1$. The task set $\tau$ is schedulable by EDF under correctness criterion CC-3 on a single preemptive processor if and only if*

$$\forall t \in \{0, 1, 2, \ldots, \lfloor B \rfloor\}, \forall s \in \mathcal{S}(t) : \quad \sum_{\tau_i \in \tau} \text{DBF}_i(t, s) \leq t.$$

**Proof.** We separately prove the necessity and sufficiency of the schedulability test.

Test fails $\Rightarrow$ $\tau$ is unschedulable: Assume there exists $t \in \{1, 2, \ldots, \lfloor B \rfloor\}$ and $s \in \mathcal{S}(t)$ such that $\sum_{\tau_i \in \tau} \text{DBF}_i(t, s) > t$. Let $\tau$ release jobs in an interval of length $t + \epsilon$, for some $0 < \epsilon < 1$, such that all LO-criticality tasks release a job at the start of the interval and then subsequent jobs as early as possible. Let HI-criticality tasks instead release jobs such that one job has a deadline at the end of the interval, and previous jobs are released as late as possible in the interval.

By definition of $\mathcal{S}(t)$, at least one HI-criticality task must then release a job exactly $s + \epsilon$ time units into the interval. Let that job be the first to signal HI-criticality behavior, and let all other jobs require their largest allowed execution time.

Any HI-criticality task $\tau_i$ then releases $\psi_i(t+\epsilon)$ jobs with both release times and deadlines within the interval, of which $\psi_i(t + \epsilon - (s + \epsilon))$ are released at or after the time point where HI-criticality behavior is signaled, for a total execution time requirement of

$$\begin{aligned}
& \psi_i(t + \epsilon) \times C_i(\text{LO}) + \psi_i(t + \epsilon - (s + \epsilon)) \times (C_i(\text{HI}) - C_i(\text{LO})) \\
= \quad & \psi_i(t) \times C_i(\text{LO}) + \psi_i(t - s) \times (C_i(\text{HI}) - C_i(\text{LO})) \\
= \quad & \text{DBF}_i(t, s),
\end{aligned}$$

where $\psi_i(t + \epsilon) = \psi(t)$ since $t$ is integer and $\psi_i$ only changes value at integers.

Similarly, any LO-criticality task $\tau_i$ will release $\psi_i(t + \epsilon)$ jobs in total, of which at most $\lceil (s + \epsilon)/T_i \rceil$ are released *before* the time point where HI-criticality behavior is signaled, for a total execution time requirement of

$$\begin{aligned}
&\psi_i(t+\epsilon)\times C_i(\textsc{hi})+\min\Big(\psi_i(t+\epsilon),\Big\lceil\frac{s+\epsilon}{T_i}\Big\rceil\Big)\times(C_i(\textsc{lo})-C_i(\textsc{hi}))\\
=\;&\psi_i(t)\times C_i(\textsc{hi})+\min\Big(\psi_i(t),\Big\lfloor\frac{s}{T_i}\Big\rfloor+1\Big)\times(C_i(\textsc{lo})-C_i(\textsc{hi}))\\
=\;&\textsc{dbf}_i(t,s),
\end{aligned}$$

where $\lceil(s+\epsilon)/T_i\rceil=\lfloor(s+\epsilon)/T_i\rfloor+1=\lfloor s/T_i\rfloor+1$ since both $s$ and $T_i$ are integer.

The total workload of jobs with both release time and deadline within the interval of size $t+\epsilon$ is then $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,s)$. Since $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,s)$ is integer-valued and since $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,s)>t$ by assumption, we must also have $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,s)>t+\epsilon$. It follows that the total workload that must be scheduled inside the interval is greater than the length of the interval, hence it is impossible to meet all deadlines on a single processor.

$\underline{\tau\text{ is unschedulable}\Rightarrow\text{test fails:}}$ Assume that $\tau$ is unschedulable by EDF and let $t_2$ be the time point of a deadline miss. Let $t_1$ be the earliest time point before $t_2$ such that there exists at least one active job with deadline no later than $t_2$ at any time in the interval $[t_1,t_2]$. By definition of $t_1$, there are no active jobs in $[t_1,t_2]$ with deadline latest at $t_2$ that are also released earlier than $t_1$. It follows that EDF schedules jobs that both arrive no earlier than $t_1$ and have deadline no later than $t_2$ during the entirety of $[t_1,t_2]$. Still one of those jobs misses its deadline at $t_2$, so the total workload of those jobs must exceed $t_2-t_1$.

We let $t=t_2-t_1$ and consider three cases.

- HI-*criticality behavior is signaled before* $t_1$*:* The total workload of all jobs scheduled by EDF in $[t_1,t_2]$ can be no more than $\sum_{\tau_i\in\tau}\psi_i(t)\times C_i(\textsc{hi})$. By Eq. 13 we have

$$\sum_{\tau_i\in\tau}\psi_i(t)\times C_i(\textsc{hi})\le\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,0).$$

- HI-*criticality behavior has not been signaled by* $t_2$*:* The total workload of all jobs scheduled by EDF in $[t_1,t_2]$ can be no more than $\sum_{\tau_i\in\tau}\psi_i(t)\times C_i(\textsc{lo})$. Using $\psi_i(t)\le\lfloor t/T_i\rfloor+1$ and Eq. 13 we get

$$\sum_{\tau_i\in\tau}\psi_i(t)\times C_i(\textsc{lo})=\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,t).$$

- HI-*criticality behavior is first signaled in* $[t_1,t_2]$*:* Let $t_{\text{signal}}$ be the time point where HI-criticality behavior is first signaled. The total workload of all jobs scheduled by EDF in $[t_1,t_2]$ can be no more than $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,t_{\text{signal}}-t_1)$.

In all three cases, the total workload of the jobs scheduled by EDF in $[t_1,t_2]$ can be no more than $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,s)$ for some $s\in[0,t]$. Since the total workload of those jobs must exceed $t$, we then have $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t,s)>t$ for some $s\in[0,t]$.

By Lemma 6, there must then exist $t'\in\{0,1,\ldots,\lfloor B\rfloor\}$ and $s'\in\mathcal{S}(t')$ such that $\sum_{\tau_i\in\tau}\textsc{dbf}_i(t',s')>t'$. This demonstrates the sufficiency of the test and concludes the proof. ◀

▶ **Corollary 8.** *The schedulability test described in Theorem 7 can be implemented to run in pseudo-polynomial time if* $\max(U_{\text{LO}},U_{\text{HI}})\le c$ *for some constant* $c<1$*.*

**Proof.** If $\max(U_{\text{LO}},U_{\text{HI}})\le c$, then $B$ is clearly pseudo-polynomially bounded. It follows that $\{0,1,\ldots,\lfloor B\rfloor\}$ is of pseudo-polynomial size, and so is $\mathcal{S}(t)$ for any $t\in\{0,1,\ldots,\lfloor B\rfloor\}$. ◀

## 6 Comparison and Recommendations

The results in Sections 3–5 above establish that CC-3 is the most tractable of our three correctness criteria both from the run-time complexity perspective and in the sense that we have the most positive results regarding sporadic task systems about this criterion. We point out that *correctness criterion CC-3 is a stronger constraint than correctness criterion CC-2* – any schedule for an instance that satisfies correctness criterion CC-3 also satisfies correctness criterion CC-2 for that instance (this follows from the observation that CC-3 guarantees any LO-criticality job spanning a mode-transition instant its larger WCET, while correctness criterion CC-2 only requires this for those LO-criticality jobs that have already begun execution prior to the mode transition). In a similar vein *correctness criterion CC-2 is a stronger constraint than correctness criterion CC-1* since CC-2 guarantees some LO-criticality jobs spanning a mode-transition instant (those that began execution prior to the transition) their larger WCET, while correctness criterion CC-1 does not require this for any LO-criticality job. It therefore follows that a schedule for an instance satisfying correctness criterion CC-3 also satisfies correctness criteria CC-2 and CC-1: *correctness criterion CC-3 is a conservative over-approximation of correctness criteria CC-2 and CC-1.* Based on this observation and the additional tractability of CC-3 compared to CC-1 and CC-2, we recommend that when graceful degradation is the goal **correctness criterion CC-3 be considered the default correctness criterion for semi-clairvoyant scheduling**, and furthermore that **EDF be considered the default preferred run-time semi-clairvoyant scheduling algorithm**.

**Quantifying the cost.** We now quantify the cost of our recommendation: how much faster does a processor need to be in order to ensure that an instance that can be correctly scheduled under either of the weaker correctness criteria CC-1 or CC-2 can also be correctly scheduled under the more conservative correctness criterion CC-3? We formalize this metric as the *criteria loss*:

▶ **Definition 9** (Criteria Loss). *For two different correctness criteria CC-$x$ and CC-$y$ for $x, y \in \{1, 2, 3\}, (x \neq y)$, the criteria loss of CC-$x$ compared to CC-$y$ is the smallest number $\ell$ such that any collection of jobs $\mathcal{J}$ that is schedulable under correctness criterion CC-$y$ on a unit-speed processor is also schedulable on a speed-$\ell$ processor under correctness criterion CC-$x$.*

We start with an upper bound for the criteria loss of CC-3 compared to the others.

▶ **Lemma 10.** *The criteria loss of CC-3 compared to CC-1 or CC-2 is no greater than 2.*

**Proof.** First, we note that a necessary condition for schedulability of a collection of jobs $\mathcal{J}$ with any algorithm under CC-1 or CC-2 is that it should meet all deadlines in the two cases where either (i) every job $J_i \in \mathcal{J}$ executes for exactly $c_i(\text{LO})$ time units (i.e., HI-criticality mode is never signaled) or (ii) every job $J_i \in \mathcal{J}$ executes for exactly $c_i(\text{HI})$ time units (i.e., HI-criticality mode is signaled at the first HI-criticality job arrival, and each LO-criticality job completes upon having executed for exactly $c_i(\text{HI})$ time units, even if it would be allowed to execute for $c_i(\text{LO})$ time units).

Second, due to the sustainability of EDF [6], we note that a sufficient condition for EDF to successfully schedule $\mathcal{J}$ under CC-3 is that it could meet all deadlines if (iii) every job $J_i \in \mathcal{J}$ executes for exactly $\max(c_i(\text{LO}), c_i(\text{HI}))$ time units. It follows directly from standard analysis of EDF on non-mixed criticality jobs that it will succeed with (iii) on a speed-2 processor if any algorithm can succeed with both (i) and (ii) on a unit-speed processor. ◀

Next we see that this bound is tight compared to CC-2.

▶ **Lemma 11.** *The criteria loss of CC-3 compared to CC-2 is at least 2.*

**Proof.** Consider the collection $\mathcal{J} = \{J_1 = (\textsc{lo}, 0, [k-1, 0], k), J_2 = (\textsc{hi}, 1, [0, k-1], k)\}$ of two jobs. Clearly $\mathcal{J}$ is schedulable under CC-2: simply idle the processor until the arrival of $J_2$ at time instant 1, after which there will be at most $(k-1)$ work to be done no matter if $J_2$ signals HI-criticality behavior or not.

Under CC-3, $J_1$, having arrived prior to $J_2$, can require up to $c_1(\textsc{lo}) = k-1$ units of execution. If $J_2$ then signals HI-criticality behavior, the total workload over $[0, k]$ may be as high as $2(k-1)$. As $k \to \infty$, we see that this would require a speed-2 processor. ◀

Lemmas 10 and 11 together yield the following theorem which completely characterizes the worst-case penalty of over-approximating CC-1 or CC-2 by CC-3.

▶ **Theorem 12.** *The criteria loss of CC-3 compared to either CC-1 or CC-2 is exactly 2.*

**Proof.** From Lemma 10 we know that the criteria loss of CC-3 compared to CC-1 or CC-2 is at most 2. From Lemma 11 we know that the criteria loss of CC-3 compared to CC-2 is at least 2. Since CC-2 is a conservative over-approximation of CC-1, the criteria loss of CC-3 compared to CC-1 must be at least 2 as well. ◀

For the sake of completeness we also present bounds on the criteria loss of CC-2 compared to CC-1.

▶ **Lemma 13.** *The criteria loss of CC-2 compared to CC-1 is in $[\varphi, 2]$, where $\varphi$ is the golden ratio $\varphi = (1+\sqrt{5})/2 \approx 1.618$.*

**Proof.** First, observe that the upper bound of 2 clearly holds here as well. For simplicity of presentation in deriving the lower bound we use real-valued job parameters in this proof, with the observation that we can approximate those to an arbitrary level of precision with rational parameters. Rational parameters can in turn be changed to integer parameters by scaling everything with the least common multiple of the denominators without affecting the schedulability of the jobs.

Let $x = (3-\sqrt{5})/2$ and consider the collection of jobs $\mathcal{J} = \{J_1, J_2\}$, where

$$J_1 = (\textsc{lo}, 0, [1, 0], 1),$$
$$J_2 = (\textsc{hi}, x, [0, 1-x], 1).$$

Note that $\mathcal{J}$ is schedulable under CC-1 since we can simply schedule $J_1$ in $[0, x)$ and then see whether $J_2$ signals HI-criticality behavior when it arrives. If $J_2$ does not signal HI-criticality behavior we continue executing $J_1$ until it finishes, otherwise we execute $J_2$ until it finishes since $J_1$ already has received more than $c_1(\textsc{hi})$ execution time.

Under CC-2 we must make the choice of whether to start executing $J_1$ before the arrival of $J_2$. If we do start executing $J_1$ immediately and $J_2$ later signals HI-criticality behavior, then we need to finish a total of $1 + (1-x)$ units of work over $[0, 1]$, and we need a speed-$\ell$ processor where

$$\ell \geq 2 - x = \frac{1+\sqrt{5}}{2} = \varphi.$$

If we instead decide to idle the processor until the arrival of $J_2$ and $J_2$ arrives without signaling HI-criticality behavior, then we need to finish $J_1$'s entire execution time $c_1(\text{LO}) = 1$ in $[x, 1]$. For this we need a speed-$\ell$ processor where

$$\ell \geq \frac{1}{1-x} = \frac{2}{\sqrt{5}-1} = \frac{1+\sqrt{5}}{2} = \varphi,$$

which completes the proof. ◀

## 7 Context and Conclusions

Since the mixed-criticality model was introduced by Vestal [30], several extensions and variations have been proposed. Criticism of the original model has made it clear that some form of *graceful degradation* often is necessary for mixed-criticality scheduling to be used in practice. In this paper we have combined graceful degradation with *semi-clairvoyant scheduling*, an interesting new take on how and when information becomes available at runtime to a scheduler, and studied this under three different *correctness criteria* that we labeled CC-1, CC-2 and CC-3. Although the differences between the correctness criteria appear minor – they differ only in the treatment of LO-criticality jobs that are active when a HI-criticality job signals HI-criticality behavior – we have seen that they require wildly differing solutions. The difference in the complexity of the associated schedulability problems is also stark: schedulability for a collection of jobs is solvable in $O(n^2 \log n)$ time for CC-3, but is NP-complete in the strong sense for CC-2.

There is no single correctness criterion which is the correct one in all situations: each is a reasonable model for some types of systems. However, as CC-3 is a safe over-approximation of the other criteria it looks particularly useful as a default model. This is especially true considering that it leads to easy scheduling (plain EDF is an optimal scheduler) and that it is easy to analyze (in polynomial time for jobs, in pseudo-polynomial time for arbitrary deadline tasks if utilization is bounded).

While we have studied these problems with the added generalization of graceful degradation, it should be noted that the correctness criteria – and the results of this paper – are equally valid without graceful degradation. This is represented by simply having $c_i(\text{HI}) = 0$ or $C_i(\text{HI}) = 0$ for all LO-criticality jobs or tasks.

We also note that the correctness criteria can apply equally to systems without semi-clairvoyance: in ordinary (non-clairvoyant) mixed-criticality scheduling we can still have different correctness criteria for the LO-criticality jobs that are active when it is first discovered that the system is exhibiting HI-criticality behavior (i.e., when a job has executed for its LO-criticality WCET without signaling completion). In such systems, CC-1 would correspond to the standard semantics as studied in most previous work, but it is not necessarily the most appropriate one, or the one that is easiest to work with.

## References

1 Kunal Agrawal, Sanjoy Baruah, and Alan Burns. Semi-clairvoyance in mixed-criticality scheduling. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 458–468, December 2019. doi:10.1109/RTSS46320.2019.00047.

2 S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press. doi:10.1109/REAL.1990.128746.

3 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems.* Springer Publishing Company, Incorporated, 2015. doi:10.1007/978-3-319-08696-5.

4 Sanjoy 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*, 2012. doi:10.1109/TC.2011.142.

5 Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM*, 62(2):14:1–14:33, 2015. doi:10.1145/2699435.

6 Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 159–168, Rio de Janeiro, December 2006. IEEE Computer Society Press. doi:10.1109/RTSS.2006.47.

7 Sanjoy Baruah, Alan Burns, and Robert Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press. doi:10.1109/RTSS.2011.12.

8 Sanjoy Baruah, Alan Burns, and Zhishan Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *Proceedings of the 2016 28th EuroMicro Conference on Real-Time Systems*, ECRTS '16, Toulouse (France), 2016. IEEE Computer Society Press. doi:10.1109/ECRTS.2016.12.

9 Sanjoy Baruah, Arvind Easwaran, and Zhishan Guo. MC-Fluid: simplified and optimally quantified. In *Real-Time Systems Symposium (RTSS), 2015 IEEE*, December 2015. doi:10.1109/RTSS.2015.38.

10 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. doi:10.1109/RTAS.2010.10.

11 I. Borosh and L. Treybig. Bounds on positive integral solutions of linear diophantine equations. *Proceedings of the American Mathematical Society*, 55:299–304, 1976. doi:10.1090/S0002-9939-1976-0396605-3.

12 A. Burns and S. Baruah. Timing faults and mixed criticality systems. In Jones and Lloyd, editors, *Dependable and Historic Computing*, volume LNCS 6875, pages 147–166. Springer, 2011. doi:10.1007/978-3-642-24541-1_12.

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 Ian Davis. Schedulability analysis for adaptive mixed criticality systems with arbitrary deadlines and semi-clairvoyance. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2020. doi:10.1109/RTSS49844.2020.00013.

15 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.

16 Lin Chen, Franziska Eberle, Nicole Megow, Kevin Schewior, and Cliff Stein. A general framework for handling commitment in online throughput maximization. In *Integer Programming and Combinatorial Optimization (IPCO)*, pages 141–154, Cham, 2019. Springer International Publishing.

17 Michael Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

18 Pontus Ekberg and Wang Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems*, 50(1):48–86, 2014. doi:10.1007/s11241-013-9187-z.

19 Rolf Ernst and Marco Di Natale. Mixed criticality systems - A history of misconceptions? *IEEE Design & Test*, 33(5):65–74, 2016. doi:10.1109/MDAT.2016.2594790.

20 Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 139–148, New York, NY, USA, 2015. ACM. doi:10.1145/2834848.2834869.

21 Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.

22 G. Giannopoulou, P Huang, R Ahmed, D Bartolini, and L Thiele. Isolation scheduling on multicores: model and scheduling approaches. *Real-Time Systems: The International Journal of Time-Critical Computing*, 53:614–667, 2017. doi:10.1007/s11241-017-9277-4.

23 Xiaozhe Gu and Arvind Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 47–56, 2016. doi:10.1109/RTSS.2016.014.

24 Z. Guo, K. Yang, S. Vaidhun, S. Arefin, S. K. Das, and H. Xiong. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 373–383, 2018. doi:10.1109/RTSS.2018.00052.

25 B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 214–223, Los Alamitos, 1995. IEEE Computer Society Press. doi:10.1109/SFCS.1995.492478.

26 Jaewoo Lee, Kieu-My Phan, Xiaozhe Gu, Jiyeon Lee, A. Easwaran, Insik Shin, and Insup Lee. MC-Fluid: Fluid model-based mixed-criticality scheduling on multiprocessors. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 41–52, December 2014. doi:10.1109/RTSS.2014.32.

27 C. Liu and J. 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.

28 A. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, Washington D.C., May 1988.

29 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. doi:10.1109/ECRTS.2013.20.

30 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press. doi:10.1109/RTSS.2007.47.

31 Reinhard Wilhelm. Mixed feelings about mixed criticality (invited paper). In Florian Brandner, editor, *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis*, pages 1:1–1:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/OASIcs.WCET.2018.1.

# Hard Real-Time Stationary GANG-Scheduling

**Niklas Ueter**
Department of Computer Science, TU Dortmund University, Germany

**Mario Günzel**
Department of Computer Science, TU Dortmund University, Germany

**Georg von der Brüggen**
Max Planck Institute for Software Systems, Kaiserslautern, Germany

**Jian-Jia Chen**
Department of Computer Science, TU Dortmund University, Germany

## Abstract

The scheduling of parallel real-time tasks enables the efficient utilization of modern multiprocessor platforms for systems with real-time constrains. In this situation, the gang task model, in which each parallel sub-job has to be executed simultaneously, has shown significant performance benefits due to reduced context switches and more efficient intra-task synchronization.

In this paper, we provide the first schedulability analysis for sporadic constrained-deadline gang task systems and propose a novel stationary gang scheduling algorithm. We show that the schedulability problem of gang task sets can be reduced to the uniprocessor self-suspension schedulability problem. Furthermore, we provide a class of partitioning algorithms to find a stationary gang assignment and show that it bounds the worst-case interference of each task. To demonstrate the effectiveness of our proposed approach, we evaluate it for implicit-deadline systems using randomized task sets under different settings, showing that our approach outperforms the state-of-the-art.

**2012 ACM Subject Classification** Computing methodologies → Concurrent algorithms; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time operating systems

**Keywords and phrases** Real-Time Systems, Gang Scheduling, Parallel Computing, Scheduling Algorithms

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.10

**Funding** This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (Project no. 398602212). This result is part of two projects (PropRT and TOROS) that have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreements No. 865170 and No. 803111).

## 1 Introduction

In hard real-time systems, it is mandatory to verify the temporal behavior of the application, e.g., the compliance to deadline constraints, by means of timing analysis. Due to the high computational demands of modern real-time systems, multiprocessor platforms are increasingly utilized since they potentially allow parallel tasks to be executed efficiently. In parallel task scheduling, inter- and intra-task parallelism has to be considered in the timing analysis, where inter-task parallelism refers to the co-scheduling of different tasks and intra-task parallelism refers to parallel execution of a single task. In the context of task models for parallel computing, fork/join models [26], synchronous parallel task models, and DAG (directed-acyclic graph) based task models [4,5,10,11,18,19,28] have been proposed and analyzed with respect to real-time constraints.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 10; pp. 10:1–10:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The scheduling algorithms for parallel tasks can be classified into three models: *rigid*, *moldable*, and *malleable* tasks. A parallel task is called *rigid* if the number of processors assigned to it is specified externally to the scheduler a priori and does not change throughout its execution; *moldable* if the number of processors assigned to it is determined by the scheduler and does not change throughout its execution; and *malleable* if the number of processors assigned to it can be changed by the scheduler during its execution. Such classifications can be found in the literature of multiprocessor scheduling and real-time systems such as [20].

In the gang task model, a set of threads is grouped together into a so called *gang* with the additional constraint that all threads of a gang must be co-scheduled at the same time on available processors. It has been demonstrated that gang-based parallel computing can improve the performance in many cases [17, 23]. Even more, Wasly et al. [32] provided experimental evidence of negative effects of non-gang scheduling with respect to the number of context-switches and increased thread execution time due to blocking when threads are not executed together. Moreover, the authors argue that by scheduling all threads of a task in-parallel, the communication time can be easily accounted for, given that the inter-processor interconnect provides real-time bounds. Due to its practicability, the gang model is supported by many parallel computing standards, e.g., MPI, OpenMP, Open ACC, or GPU computing.

One advantage of the rigid gang model is that the interference caused by shared resource and intra-task parallelism can potentially be quantified better, thus reducing the worst-case execution time of the gang. Within a gang, co-scheduling of memory accesses and computation is possible, which can also potentially reduce the worst-case execution time of the gang. Specifically, one strict view of this is the RT-Gang model by Ali and Yun [1], in which all processors are allocated to a gang at the same time.

The computational complexity of the rigid gang scheduling problem was studied back in 1980s. Specifically, it has been shown that finding the optimal schedule for the rigid gang scheduling problem is *NP*-hard in the strong sense even when all the tasks have the same period and the same deadline [25]. Even simpler cases, like three machines [9] or unit execution time per task [22] are also shown to be NP-hard in the strong sense.

To schedule a set of *ordinary* periodic [27] or sporadic [29] real-time tasks on a multiprocessor platform, three paradigms have been widely adopted: partitioned, global, and semi-partitioned multiprocessor scheduling. A comprehensive survey can be found in [15]. For the rigid gang scheduling problem, the three scheduling paradigms are slightly modified and called *stationary*, *global*, and *semi-stationary* (rigid) gang scheduling. The *stationary gang* scheduling paradigm statically assigns a gang task to a set of processors, in which the cardinality of the set is equal to the gang size of the task. After this assignment is done, a gang task is only eligible to be executed on stationary processors assigned to it. The *semi-stationary* scheduling paradigm allows a gang task to execute on any subset of processors within a given set of processors that is larger than the gang size itself. That is, it allows a job of the gang task to migrate from one subset of processors to another sub set of the given processors at any time. The *global* rigid gang scheduler allows a gang task to migrate to any available set of processors as long as the gang size constraints are met.

Note that when the gang size is 1 for each task (i.e., tasks are not executed in parallel and are ordinary periodic or sporadic tasks), the stationary, global, and semi-stationary gang scheduling paradigms correspond to the partitioned, global, and semi-partitioned multiprocessor scheduling paradigms, respectively.

In real-time systems, rigid gang scheduling has been mostly studied under global earliest-deadline-first (EDF) scheduling, in which the set of processors used by a gang task is not fixed and can be dynamically relocated at runtime, e.g., [16,24,30]. Specifically, in [24], the authors extended Baruah's [3] multiprocessor global EDF analysis for ordinary sporadic real-time tasks to deal with global EDF gang scheduling, which has been disproved by Richard et al. [30]. The only valid analysis for global EDF gang scheduling is from Dong and Liu [16] and restricted to implicit-deadline sporadic real-time rigid gang task systems. They provide two utilization-based analyses, one optimized and one approximated.

Goossens and Richard [20] studied fixed-priority scheduling for the rigid gang scheduling problem for implicit-deadline periodic real-time task systems. They presented two algorithms, one based on linear programming and another based on a heuristic algorithm, providing exact and sufficient schedulability tests. Moreover algorithms based on deadline partitioning (DP-Fair) for periodic gang systems have been proposed. However the many preemptions of DP-Fair make this algorithm impractical and the complexity of the proposed algorithms is high especially for a large number of processors. The authors themselves discuss the problems to extend their algorithms to sporadic job arrival sequences due to its non-determinism.

For classical multiprocessor scheduling, is has been recently shown that global static-priority scheduling [31] and global EDF as well as global FIFO scheduling [8] are dominated by partitioned scheduling under state-of-the-art efficient sufficient schedulability tests, e.g., [6,21]. The main reason is due to the inherited pessimism in those tests, which all stem from the work by Baker [2]. Hence, they all use carry-in interference to compensate the lack of a critical instant theorem and divide the higher-priority interference by the number of processors, i.e., they have a multiplicative factor of $1/M$ in the corresponding analyses. We note that the factor $1/M$ also appears in the schedulability tests in [16].

**Contributions:** In this paper we explore *stationary gang scheduling* for a set of sporadic real-time tasks with constrained deadlines (i.e., the relative deadline of a task is no more than its minimum inter-arrival time) on a homogeneous symmetric multiprocessor system consisting of $M$ processors. We develop the corresponding schedulability analyses for fixed-priority scheduling and a heuristic algorithm for stationary gang assignments.

The contributions of this paper are as follows:

- We present schedulability tests for stationary gang assignments for constrained-deadline sporadic real-time tasks in Section 3. To the best of our knowledge, this is the first schedulability analysis that is capable of verifying the schedulability of sporadic constrained-deadline gang task systems, whilst the analysis in [16] is limited to implicit-deadline sporadic real-time rigid gang task systems and the algorithm in [20] is limited to implicit-deadline periodic tasks. Our success is due to the observation of self-suspension behavior in Section 3.2 and the recent improvement of optimizations and analyses for dynamic self-suspension task behavior [12,13].
- We propose a class of partitioning algorithms based on the concept of consecutive stationary gang assignment in Section 4. Furthermore, we show that consecutive stationary gang assigments yield beneficial theoretical properties that can be used to upper-bound the worst-case interference suffered by any task according to the ratio of gang sizes of two tasks.
- In Section 5, we compare our algorithm to the state-of-the-art schedulability analysis for global EDF by Dong and Liu [16] by evaluation synthetically generated sporadic real-time task systems with implicit deadlines. The evaluation results show that our algorithm outperforms the algorithm by Dong and Liu [16]. Furthermore, we conducted evaluations for constrained-deadline task systems and observe reasonable schedulability.

## 2 System Model and Stationary GANG Scheduling

In this paper we consider a symmetric multiprocessor (SMP) system composed of $M$ identical processors and analyze the response-times of a gang task set with constrained deadlines using our proposed stationary gang scheduler.

We consider a set $\mathbb{T} = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$ of $n$ constrained-deadline gang tasks to be scheduled on a set $\mathbb{P} = \{P_0, P_1, \dots, P_{M-1}\}$ of $M$ identical processors using fixed-priority rigid gang schedulers under the additional constraint of stationary gang assignments. Each task has a fixed-priority that is inherited by each instantiated job. We use $\pi_i$ to denote the priority of task $\tau_i$ and say $\tau_j$ has higher priority than $\tau_i$ if and only if $\pi_j > \pi_i$. We assume that no two tasks have the same priority, i.e., there are sufficient priority levels. Moreover, each task is assigned and restricted to a subset of processor, namely its stationary gang assignment, to execute on. This subset does not change in time, i.e., it is rigid. Throughout this section, we will assume that a stationary gang assignment is given for each task and revisit the problem to generate provably good stationary assignments in Section 4.

▶ **Definition 1.** *A sporadic constrained-deadline gang task $\tau_i$ is defined by $(C_i, E_i, D_i, T_i)$ and releases an infinite number of task instances, called jobs. Each job of a task releases a gang of $E_i$ sub-jobs with worst-case execution time $C_i$, that have to be executed in parallel. That is, either all $E_i$ sub-jobs are scheduled simultaneously or none is. Hence, a total workload of $E_i \cdot C_i$ has to be executed in the time interval between job release and deadline. The period $T_i$ denotes the minimal inter-arrival time of two jobs of $\tau_i$ and each task has a relative deadline $D_i \leq T_i$. Moreover, the utilization of a gang task is given by $U_i = E_i \cdot C_i / T_i$.*

This means that when a job of $\tau_i$ is released at time $t$, the subsequent job of $\tau_i$ must be released not earlier than at time $t + T_i$. Furthermore, to fulfill its timing constraints, this job must be able to finish its execution not later than its absolute deadline at time $t + D_i$. The response time of a job of $\tau_i$ is its finishing time minus its release time, and the worst-case response time $R_i$ of task $\tau_i$ under a given scheduling policy is the maximum response time of any job of $\tau_i$ for any job arrival sequence possible according to the parameters of tasks in $\mathbb{T}$.

We now define stationary gang assignment and the related schedules.

▶ **Definition 2.** *A stationary gang assignment $A_i \subseteq \{P_0, P_1, \dots, P_{M-1}\}$ of a gang task $\tau_i$ is a subset of processors of size $|A_i| = E_i \leq M$, that are assigned to execute jobs of task $\tau_i$.*

In order to formalize the properties of a fixed-priority stationary gang scheduler, we first formalize the definition of an arbitrary schedule.

▶ **Definition 3.** *A schedule $\sigma_{P_q} : \mathbb{R} \mapsto \mathbb{T} \cup \{\perp\}$ for a processor $P_q$ with $q \in \{0, \dots, M-1\}$ is a mapping from the continuous time domain to the task that is executed at time $t$ or to $\perp$ if the processor idles, i.e.,*

$$\sigma_{P_q} : \mathbb{R} \mapsto \mathbb{T} \cup \{\perp\}, \quad \sigma_{P_q}(t) = \begin{cases} \tau_i & \text{\textit{if task} } \tau_i \text{ \textit{is executed on} } P_q \text{ \textit{at time} } t \\ \perp & \text{\textit{if} } P_q \text{ \textit{is idle at time} } t \end{cases} \tag{1}$$

Despite that a realistic schedule does not perform context switch arbitrarily, e.g., due to granularity determined by the system tick duration, our analysis can in general be applied in the continuous time domain. A stationary gang schedule is described as follows.

▶ **Definition 4.** *A schedule for a multiprocessor system satisfies the stationary gang property if for each task $\tau_i$ and its stationary gang assignment $A_i$, the following property holds:*

$$\bigwedge_{P_q \in A_i} (\sigma_{P_q}(t) = \tau_i) \quad \text{\textit{if and only if} } \tau_i \text{ \textit{is scheduled at time} } t \tag{2}$$

Whenever we argue about schedules that satisfy the stationary gang property, we for example write $\sigma_{A_i}(t) = \tau_i$, if task $\tau_i$ is scheduled on all the processors in $P_q \in A_i$ at time $t$. Throughout this paper, we say that a gang task $\tau_i$ is *active* at time $t$ if a job of $\tau_i$ is released and not yet finished. The stationary gang scheduler then schedules all active tasks $\tau_i$ that are the highest-priority tasks with respect to all other tasks that use processors in $A_i$.

▶ **Example 5.** Consider the stationary gang schedule illustrated in Figure 1 with two tasks $\tau_k$ and $\tau_i$ with the stationary gang assignments $A_k = \{P_2, P_3\}$ and $A_i = \{P_1, P_2, P_3\}$. Moreover, let $\pi_k < \pi_i$. Therefore $\tau_i$ that releases a job at time 1 preempts task $\tau_k$. Whenever $\tau_i$ is preempted on $A_i$, $\tau_k$ is the highest-priority task amongst all tasks that compete for processors $P_2$ and $P_3$ and is thus scheduled. ◀

## 3 Schedulability Test for Stationary Gang Scheduling

This section presents the schedulability test for stationary gang scheduling, provided that each task $\tau_i$ has a predefined stationary gang assignment $A_i$. How to achieve good stationary gang assignments is discussed in Section 4. Throughout this section, our analysis focuses on the analysis to validate whether task $\tau_k$ can meet its deadline constraint, provided that the tasks with higher priorities than $\tau_k$ are validated beforehand. Hence, the validation of schedulability iterates from the highest-priority task to the lowest-priority task in $\mathbb{T}$.

Towards this, we present methods to analyze the contention between a higher-priority task $\tau_i$ and the task $\tau_k$ under analysis in Section 3.1. Specifically, our result shows that $\tau_i$ can be considered as a self-suspending task under certain circumstances. Due to this observation that some higher-priority tasks can be transformed into self-suspending tasks, we employ existing suspension-aware schedulability analysis and present our schedulability test for stationary gang scheduling in Section 3.2.

### 3.1 Contention Analysis

The preemptive fixed-priority stationary gang scheduler always schedules the active task $\tau_k$ that has the highest priority with respect to all other tasks that use processors in $A_k$.

▶ **Definition 6.** *The contention domain $\delta(A_k)$ of a set of processors $A_k$ is defined as*

$$\delta(A_k) := \{\tau_\ell \in \mathbb{T} \,|\, A_k \cap A_\ell \neq \emptyset\} \tag{3}$$

Based on this behavior, we can formalize the condition for a higher-priority task $\tau_i$ to be able to interfere with a task $\tau_k$ ($\pi_i > \pi_k$) as follows

$$\tau_i \text{ interferes with } \tau_k \iff \tau_i \in \delta(A_k) \tag{4}$$

This is simply due to the fact that a task $\tau_i$ is able to preempt another task $\tau_k$ if and only if it starts to be executed on a processor $P_q$ on which $\tau_k$ is assigned. In such a case, $P_q \in A_k$ and $P_q \in A_i$ and in conclusion $P_q \in A_k \cap A_i$, i.e., $\tau_i \in \delta(A_k)$.

As a consequence, the schedulability of gang task $\tau_k$ can be reduced to the schedulability of a single job with worst-case execution time $C_k$ that is subjected to the maximum interference by jobs of tasks in $\delta(A_k)$. In the remainder of this subsection, we show that the interfering behavior of task $\tau_i$ in $\delta(A_k)$ can be over approximated by the interference behaviour of a corresponding sequential task with dynamic self-suspension behavior, where the suspension-time depends on the stationary gang assignments of the interfering tasks.

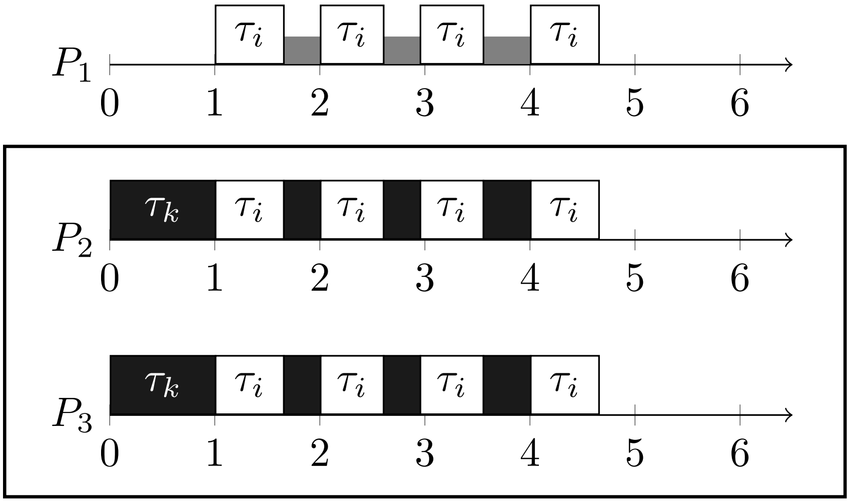


**Figure 1** An illustration of the suspension behavior of task $\tau_k$ from the point of view of the task $\tau_i$. The gray boxes denote interference due to other higher-priority tasks on processor $P_1$.

▶ **Definition 7** (Dynamic Self-Suspension [13, 14]). *A task is said to have dynamic self-suspension behavior if an active task can transition from a ready state into a suspended state, in which the task is exempted from the scheduling decisions, and resume into a ready state at any time. The cumulative amount of time that an active task $\tau_i$ can spend in a suspended state is upper-bounded by a parameter $S_i$.* ◀

The link between stationary gang schedules and dynamic self-suspension behavior can be illustrated in the following example.

▶ **Example 8.** Assume an arbitrary fixed-priority gang schedule for two tasks $\tau_k, \tau_i$ with $\pi_k < \pi_i$ and a stationary gang assignments $A_k = \{P_2, P_3\}$ and $A_i = \{P_1, P_2, P_3\}$ as shown in Figure 1. We analyze the execution of task $\tau_k$ solely from the perspective of the processors specified in $A_k$, i.e., $P_2$ and $P_3$. Due to the arrival of the higher-priority task $\tau_i$ at time $t = 1$, $\tau_k$ is preempted. However, execution on a processor not in $A_k$ interferes with the execution of $\tau_i$. Whenever $\tau_i$ is preempted by some interfering tasks on $P_1$ (denoted by the gray boxes), $\tau_k$ is scheduled on its assigned processors as described in the definition of the stationary gang scheduling paradigm. Hence, if we only analyze the execution of $\tau_k$ with respect to its assigned processors, then transparent preemption of $\tau_i$ equates to self-suspending behavior that needs to be accounted for in the response-time analysis of $\tau_k$. ◀

In the following, we formalize and explain how these task model substitutions can be safely obtained. Before moving into the formal proof, we present the conditions that hold for our scheduling policy.

▶ **Definition 9.** *A task $\tau_j$ is executed at time $t$ if and only if*

1. *Task $\tau_j$ is active at time $t$.*
2. *There exists no task $\tau_\ell \in \delta(A_j)$ with higher priority, i.e., $\pi_\ell > \pi_j$, such that $\tau_\ell$ is executed at time $t$.*

For further clarification, assume that we are interested in the response-time of task $\tau_k$ and thus analyze the interference caused by higher-priority tasks that use some processors in $A_k$. Assume that $\tau_i$ is active and has the highest priority among all active jobs that use some processors in $A_k$ at time $t$, but is interfered by a higher-priority task $\tau_\ell \in \delta(A_i)$ (e.g., the grey boxes in Figure 1). From the perspective of task $\tau_k$ this job is *self-suspended* and is resumed when the interfering task $\tau_\ell$ releases the processor. More specifically, we provide the following definition.

▶ **Definition 10.** *A task $\tau_i \in \delta(A_k)$ is in a suspended state at time $t$ with respect to a task $\tau_k$ under analysis if and only if*

1. *Task $\tau_i$ is active at time $t$.*
2. *Task $\tau_i$ has the highest priority among all active tasks on the processors in $A_k$, i.e., $\pi_i \geq \max\{\pi_j \mid \tau_j \in \delta(A_k) \text{ active at } t\}$*
3. *Task $\tau_i$ is not executed.*

We use the following definition to collect the set of tasks that may interfere with a higher-priority task $\tau_i \in \delta(A_k)$ but not interfere with the task $\tau_k$ under analysis.

▶ **Definition 11** (Self-Suspension Inducing Tasks). *The set of tasks that can induce self-suspending behavior of $\tau_i$ when analyzing task $\tau_k$ is denoted by*

$$V_{i,k} = \{\tau_\ell \in \delta(A_i) \mid \tau_\ell \notin \delta(A_k) \text{ and } \pi_\ell > \pi_i\} \tag{5}$$

We now validate that only these tasks induce self-suspending behavior for $\tau_i$.

▶ **Lemma 12.** *Suppose that task $\tau_i$ is in a suspended state at time $t$ with respect to a task $\tau_k$ under analysis, then at least one task in $V_{i,k}$ is executed at time $t$.*

**Proof.** By Definition 10, (i) $\tau_i$ is active at time $t$, (ii) $\pi_i \geq \max\{\pi_j \mid \tau_j \in \delta(A_k) \text{ active at } t\}$, and (iii) $\tau_i$ is not executed. Due to Definition 9 and since (i) and (iii) hold, there exists some task $\tau_\ell \in \delta(A_i)$ with $\pi_\ell > \pi_i$ that is executed at time $t$. It remains to show that $\tau_\ell \notin \delta(A_k)$. Assume that $\tau_\ell \in \delta(A_k)$, then from (ii) follows that $\pi_i > \pi_\ell$ which contradicts $\pi_\ell > \pi_i$. ◀

Now, we can provide a safe upper bound of the self-suspension time if $V_{i,k}$ is not empty.

▶ **Theorem 13.** *Suppose that $\pi_i > \pi_k$ and $R_i \leq D_i \leq T_i$, where $R_i$ is an upper bound on the worst-case response time of task $\tau_i$, which was already verified beforehand. The amount of time $S_{i,k}$ that a job of an active task $\tau_i$ self-suspends with respect to $\tau_k$ is at most*

$$S_{i,k} \leq \min\left\{R_i - C_i, \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{R_i}{T_j} \right\rceil\right) \cdot C_j\right\} \tag{6}$$

**Proof.** Suppose that a job of $\tau_i$ is released at time $t_i$ and finished at time at $t_i + \Delta$. By the assumption, $\Delta \leq R_i$. Let $f(t)$ be 1 if tasks $\tau_k$ and $\tau_i$ are both active at time $t$ and $\pi_i > \pi_k$ but task $\tau_k$ is executed at time $t$; otherwise $f(t)$ is 0. Therefore, the amount of time that $f(t)$ is set to 1 is the amount of time $S_{i,k}$ that the job of task $\tau_i$ self-suspends instead of preempting $\tau_k$. Therefore, $S_{i,k}$ can be calculated by integrating the function $f(t)$ from $t_i$ to $t_i + \Delta$, i.e., $S_{i,k} = \int_{t_i}^{t_i+\Delta} f(t)dt$.

Suppose the amount of time that the job of $\tau_i$ suspends during $t_i$ and $t_i + \Delta$ is $> R_i - C_i$ for contradiction. This implies that the job of $\tau_i$ has only completed $\Delta - R_i + C_i$ amount of computation. This violates the assumption that $R_i$ is the worst-case response time of $\tau_i$.

In addition, the suspension behavior of $\tau_i$ is in fact induced by the tasks in $V_{i,k}$ when analyzing task $\tau_k$. By Lemma 12, we know that such interference can only come from tasks in $V_{i,k}$. Since $R_j \leq T_j$ for every task $\tau_j$ with $\pi_j > \pi_k$, we know that the amount of time that a task $\tau_j$ is executed from $t_i$ to $t_i + \Delta$ is at most $\left(1 + \left\lceil \frac{\Delta}{T_j} \right\rceil C_j\right)$. This can be proved by showing that the jobs of $\tau_j$ that are executed in the interval $[t_i, t_i + \Delta)$ are (i) at most only one job released prior to $t_i$, and (ii) the amount of jobs that we get by releasing jobs after $t_1$ as soon as possible.[1]

Summing all tasks in $V_{i,k}$ together, we have

$$S_{i,k} = \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{\Delta}{T_j} \right\rceil C_j\right) \leq \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{R_i}{T_j} \right\rceil C_j\right)$$

where the inequality is due to the assumption that $\Delta \leq R_i$.

Putting these two safe conditions together, we reach the conclusion. ◀

Note that the estimation in Theorem 13 may not be precise as it counts the higher-priority interference of $\tau_j \in \delta(A_i)$ and $\tau_j \in \delta(A_k)$ as the suspension time of $\tau_i$ as well. This is in fact standard higher-priority interference as in uniprocessor systems.

The following corollary is a direct implication from Theorem 13.

▶ **Corollary 14.** *If $V_{i,k}$ is empty, then task $\tau_i$ does not have any self-suspension behavior, i.e., $S_{i,k} = 0$ when analyzing task $\tau_k$.*

**Proof.** This is because the right-hand side of Equation (6) is 0 under this condition. ◀

## 3.2 Schedulability Analysis

After analyzing the link between the stationary gang scheduling problem and the dynamic self-suspension problem, we now construct a worst-case response time analysis and schedulability analysis for task $\tau_k$. We provide such a bound based on suspension-aware analyses on uniprocessor systems.

On the basis of Theorem 13 and Corollary 14, we can safely upper-bound the interference of task $\tau_k$. We first collect the higher-priority tasks that interfere with $\tau_k$ in $\Psi_k$, i.e., $\Psi_k = \{\tau_i \,|\, \tau_i \in \delta(A_k) \wedge \pi_i > \pi_k\}$. For every task in $\Psi_k$, we transform it to an equivalent dynamic self-suspension task as follows:

▶ **Definition 15.** *Let a sporadic gang task $\tau_i \in \Psi_k$ be transformed to the corresponding self-suspending task $\tau_i^{sus} = (C_i, D_i, T_i, S_{i,k})$ with the same $C_i$, $D_i$, and $T_i$ as for $\tau_i$, where*

$$\begin{cases} S_{i,k} = \min\left\{R_i - C_i, \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{R_i}{T_j} \right\rceil\right) \cdot C_j\right\} & \textit{if } V_{i,k} \neq \emptyset \\ S_{i,k} = 0 & \textit{otherwise} \end{cases} \tag{7}$$

*and $V_{i,k}$ is defined as in Definition 11.*

The set $\Psi_k^{sus}$ is the set of all transformed tasks, i.e., $\Psi_k^{sus} = \cup_{\tau_i \in \Psi_k} \tau_i^{sus}$.

▶ **Theorem 16.** *Suppose that all higher-priority tasks $\tau_0, \tau_1, \ldots \tau_{k-1}$ with given stationary gang assignments $A_0, A_1, \ldots, A_{k-1}$ are already verified to be schedulable. A sporadic constrained-deadline gang task $\tau_k$ with stationary gang assignment $A_k$ is schedulable by the fixed-priority stationary gang scheduling algorithm if the worst-case response time of executing $C_k$ time units (without suspending task $\tau_k$) is at most $D_k \leq T_k$ under the interference of $\Psi_k^{sus}$ on $\underline{\text{one processor}}$ under the same priority assignment.*

[1] This is typically done with the concept of *carry-in* jobs. Since $R_i \leq T_i$, there is at most one carry-in job of $\tau_j$ released before $t_i$.

**Proof.** Suppose that a job of task $\tau_k$ is released at time $t_k$ and there is no other job of $\tau_k$ active at time $t_k$. From $t_k$, the schedule $\sigma$ either executes $\tau_k$ or higher-priority jobs on the processors in $A_k$. Therefore, the job of $\tau_k$ is either executed or interfered by higher-priority tasks in $\delta(A_k)$. Hence, only tasks in $\Psi_k$ have interference with $\tau_k$. The equivalence of the self-suspension behavior is due to Theorem 13. Therefore, the proof is complete. ◀

We adopt the current sound state-of-the-art self-suspension aware uniprocessor schedulability analyses by Chen et al. [12] for gang-scheduling, hence the correctness follows directly from the related proofs in [12] and Theorem 16.

▶ **Corollary 17.** *By the statement in Theorem 16, a gang sporadic task $\tau_k$ is schedulable by the stationary gang scheduling algorithm if*

$$\exists 0 < t \leq D_k,\ \ C_k + \sum_{\tau_i^{sus} \in \Psi_k^{sus}} \min\{C_i, S_{i,k}\} + \sum_{\tau_i^{sus} \in \Psi_k^{sus}} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \leq t \tag{8}$$

▶ **Corollary 18.** *By the statement in Theorem 16, a gang sporadic task $\tau_k$ is schedulable by the stationary gang scheduling algorithm if*

$$\exists 0 < t \leq D_k,\ \ C_k + \sum_{\tau_i^{sus} \in \Psi_k^{sus}} \left\lceil \frac{t + R_i - C_i}{T_i} \right\rceil \cdot C_i \leq t \tag{9}$$

▶ **Corollary 19.** *Suppose that there are $k$ tasks in $\Psi_k^{sus}$, indexed from the highest priority to the lowest priority, i.e., $\tau_0^{sus}$ is the highest-priority task in $\Psi_k^{sus}$. By the statement in Theorem 16, a gang sporadic task $\tau_k$ is schedulable by the stationary gang scheduling if there is a vector $\vec{x} = (x_0, x_1, \ldots, x_{k-1})$ with $x_i \in \{0, 1\}$ such that*

$$\exists 0 < t \leq D_k,\ \ C_k + \sum_{\tau_i^{sus} \in \Psi_k^{sus}} \left\lceil \frac{t + Q_i(\underline{x}) + (1 - x_i)(R_i - C_i)}{T_i} \right\rceil \leq t \tag{10}$$

*where $Q_i(\vec{x}) = \sum_{j=i}^{k-1} S_{j,k} \cdot x_j$.*

The provided schedulability analyses in Corollary 17, Corollary 18, and Corollary 19 can be evaluated using fixed-point iteration techniques. More precisely, let $W_k(t)$ denote the left-hand sides of the inequalities in the above corollaries and $\epsilon > 0$, then we verify all test-points $t_0 = W_k(\epsilon)$, $t_1 = W_k(t_0), \ldots,\ t_n = W_k(t_{n-1})$ until convergence is reached or $t_n > D_k$. Due to the fact that the above equations are step-functions and can thus only change at discontinuity points of $W_k(t)$, the amount of test-points is at most $k \cdot D_k / \min_{i<k}\{T_i\}$ resulting in pseudo-polynomial time-complexity. In the remainder of this paper, we only use $\mathcal{O}(kD_k)$ for time-complexity, since the scaling of the deadline does not change the asymptotic complexity.

As reported in [12], neither of the schedulability analyses in Corollary 17 and Corollary 18 dominate each other analytically and are incomparable. The authors also showed that the test in Corollary 19 dominates those in Corollary 17 (i.e., Lemma 17 in [12]) and Corollary 18 (i.e., Lemma 16 in [12]). To efficiently find a vector $\vec{x}$ for Corollary 19, they suggest to use three vectors, one is based on a linear approximation, one sets all elements of $\vec{x}$ to 0, and one sets the $x_i$ in $\vec{x}$ to 1 if $S_{i,k} \leq C_i$, and 0 otherwise. Specifically, in the case when the entries in $\vec{x}$ are all 0, Equation (19) is the same as Equation (18). In our evaluations we use Corollary 19 with the above three vectors and choose the best one, i.e., a task is determined schedulable if it is schedulable for at least one of the three vectors.

# 4 GANG Assignment Algorithm

Since finding optimal schedules for the rigid gang scheduling problem is NP-hard in the strong sense even in the simplest settings, we seek for approximation algorithms to solve the gang assignment problem.

In fixed-priority stationary gang scheduling, next to priority assignments, the gang assignments determine the schedulability of the task set $\mathbb{T}$. A key problem in finding stationary gang assignments is the dependency of gang assignments and the resulting interference behaviour of higher-priority tasks. In general, each task $\tau_k$ under consideration can have $\binom{M}{E_k}$ many distinct gang assignments in terms of gang to processor mappings. However, for any given gang assignment of all higher-priority tasks, there may exist subsets of these distinct gang assignments, in which the interference of all higher-priority tasks of $\tau_k$ is equivalent. A trivial example is the gang assignment of the first task, in which all gang assignments are equivalent, since there is no interfering tasks. In that case, all distinct gang assignments belong to the same equivalence class and any representative can be chosen for the gang assignment. However, finding all equivalence classes results in an exhaustive exploration of all possible solutions, which is computationally expensive especially for larger task sets.

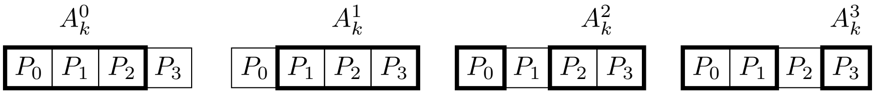


**Figure 2** Consecutive stationary gang assignments $A_k^0, A_k^1, A_k^2, A_k^3$ of a gang task $\tau_k$ with $E_k = 3$ on a system using 4 processors are generated by a sliding window.

We intend to identify a class of computationally feasible gang assignment algorithms that allow to formulate worst-case performance guarantees with respect to any optimal rigid gang scheduling algorithm. In order to get worst-case performance guarantees, it is mandatory to find (preferably small) upper bounds of interference caused by higher-priority tasks. Thus, instead of arbitrary gang assignments, we restrict ourselves to consecutive stationary gang assignments that allow to bound interference. We note however that other gang assignments can be explored starting from the consecutive assignments. By this, the approximation properties can be kept whilst improving the schedulability using any heuristic.

▶ **Definition 20.** *A consecutive stationary gang assignment $A_k^\ell$, $\ell \in \{0, 1, \ldots, M-1\}$ for a gang task $\tau_k$ in a system of $M$ processors is a set of consecutive processor indices*

$$\ell \bmod M,\ (\ell+1) \bmod M,\ \ldots,\ (\ell+E_k-1) \bmod M \tag{11}$$

*where $|A_k^\ell| = E_k \leq M$.*

An example of consecutive stationary gang assignments of a task $\tau_k$ with $E_k = 3$ on a platform of 4 processors is illustrated in Figure 2. Intuitively, the consecutive stationary gang assignments are generated by a sliding window of length 3.

Another restriction in our algorithm is to devise gang assignments in priority-order under the premise that all higher-priority tasks are verified to be schedulable. By this restriction, we only have to determine the interference behaviour of each higher-priority task that only depends on the gang assignment $A_k$ of task $\tau_k$.

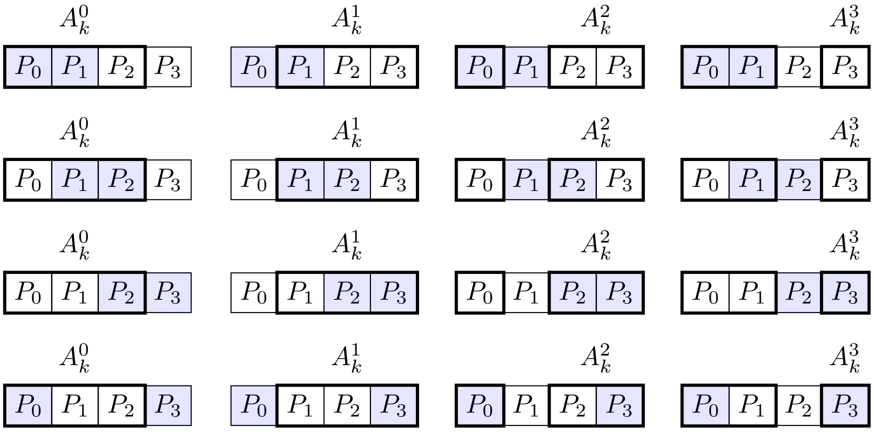


**Figure 3** Enumeration of all consecutive stationary gang assignments of a task $\tau_k$ (black window) under the condition of a given consecutive stationary gang assignment of a higher-priority task (light blue window).

The above restrictions yield the following two important theoretical properties:

1. There are always $M$ different consecutive stationary gang assignments for each task.
2. Out of these $M$ assignments, we are able to find upper-bounds for the number of consecutive stationary gang assignments of $\tau_k$, in which higher-priority tasks have self-suspension behaviour and non self-suspension behaviour respectively. That means, we are able to argue that in at most $x$ out of the $M$ consecutive stationary gang assignments, a higher-priority task $\tau_i$ has self-suspension behaviour irrespective of the actual consecutive assignment of $\tau_i$.

In Figure 3, each column illustrates the $M$ consecutive stationary gang assignments of $\tau_k$, that is subject to assignment and analysis, given a consecutive stationary gang assignment of a higher-priority task. Each row shows a different assignment of a higher-priority task $\tau_i$ indicated by the light blue window. According to the discussion in Section 3, $\tau_i$ interferes with $\tau_k$ if and only if $A_k \cap A_i$, i.e., whenever the windows in Figure 3 intersect. If for a column (consecutive assignment of $\tau_k$) there exists at least one row (consecutive assignment of $\tau_i$) in which both windows intersect then $\tau_i$ interferes with $\tau_k$ for the consecutive assignment under consideration. In the provided example, all consecutive assignments suffer interference from $\tau_i$. In Lemma 21 we prove that there are at most $E_k + E_i - 1$ out of the $M$ consecutive stationary gang assignments of $\tau_k$, in which $\tau_i$ interferes with $\tau_k$.

Moreover, guided by the observation that if $A_k \subseteq A_i$ then $\tau_i$ can not have self-suspension behaviour with respect to $\tau_k$ under analysis, we can lower-bound the number of consecutive stationary gang assignments of $\tau_k$ in which $\tau_i$ can not exhibit self-suspension behaviour. For better illustration of this observation, assume that $\tau_i$ has self-suspension behavior with respect to $\tau_k$ then there exists a task $\tau_\ell$ with higher priority than $\tau_i$ (and subsequently higher priority than $\tau_k$) such that $A_\ell \cap A_i \neq \emptyset$ and $A_\ell \cap A_k = \emptyset$. This however implies that $A_k \not\subseteq A_i$ and contradicts the assumption. This can only happen if $E_i \geq E_k$ and if so then $E_i - E_k$ many of the $M$ consecutive stationary assignments satisfy this property. In the next two lemmas, we formally prove the intuition described above.

▶ **Lemma 21.** *Given a task $\tau_k$ under analysis, each higher-priority task $\tau_i$ causes interference, i.e., $\tau_i \in \delta(A_k)$, in at most $E_i + E_k - 1$ of the $M$-many consecutive stationary gang assignments.*

**Proof.** Let the consecutive stationary gang assignments for some higher-priority tasks $i < k$ be given by the following processor indices:

$$j \bmod M,\ (j+1) \bmod M,\ \ldots,\ (j+E_i-1) \bmod M \tag{12}$$

where $j \in \{0, 1, \ldots, M-1\}$ is already given (fixed). Furthermore, let

$$\ell + h \bmod M,\ (\ell+h+1) \bmod M,\ \ldots,\ (\ell+h+E_k-1) \bmod M \tag{13}$$

denote the processor indices of a consecutive stationary gang assignment of task $\tau_k$ after the $h$-iteration for some arbitrary initial $\ell \in \{0, 1, \ldots, M-1\}$ (we only need this to show that this works for an arbitrary initial position and can be set to 0 for comprehension). Then let $h'$ denote the first iteration such that $(\ell + h' + E_k - 1) \bmod M \equiv j - 1 \bmod M$ (we shift the window of $A_k$ to the border of window of $A_i$, i.e., the two consecutive stationary gang assignments intersect in the next iteration for the first time. Therefore,

$$(\ell + h') \bmod M \equiv (j - E_k) \bmod M.$$

We have to iterate further $z$ allocations unil the index of the first processor in the allocation of $\tau_k$, i.e., $(\ell + h' + z) \bmod M \equiv (j + E_i - 1) \bmod M$ coincides with the index of the last processor in the assignment of task $\tau_i$. More formally, we seek to find the smallest $z > 0$ such that:

$$\begin{aligned} (\ell + h' + z) \bmod M &\equiv (j + E_i - 1) \bmod M \\ ((\ell + h') \bmod M) + (z \bmod M) &\equiv (j + E_i - 1) \bmod M \\ (j - E_k + z) \bmod M &\equiv (j + E_i - 1) \bmod M \end{aligned}$$

which implies that $z = E_i + E_k - 1$, i.e., $z$ consecutive stationary gang assignments yield an intersection of both tasks. ◀

We can furthermore bound the interference for self-suspending tasks as follows:

▶ **Lemma 22.** *For task $\tau_k$ (under analysis), there are at most $\min\{2E_k - 1, E_i + E_k - 1\}$ many consecutive stationary gang assignments, in which a higher-priority task $\tau_i$ has self-suspension behavior with respect to task $\tau_k$.*

**Proof.** From Lemma 21, we know that at most $E_k + E_i - 1$ many consecutive stationary gang assigments cause an intersection of consecutive stationary gang assignments of task $\tau_i$ and task $\tau_k$. We hence substract $\max\{E_i - E_k, 0\}$, namely the number of consecutive stationary gang assignments in which self-suspension behavior of $\tau_i$ is impossible, from the above. Clearly, in the case that $E_k \leq E_i$ we have $(E_k + E_i - 1) - E_i + E_k = 2E_k - 1$. Since $2E_k - 1 \leq E_i + E_k - 1$ implies that $E_k \leq E_i$ we can write it as $\min\{2E_k - 1, E_i + E_k - 1\}$. ◀

For the rest of this paper, we used deadline-monotonic priority assignment and index the tasks such that $D_1 \leq D_2 \leq \cdots \leq D_n$, in which $\tau_i$ has a higher priority than $\tau_k$ if $i < k$. Due to the additional restrictions described above, it is possible to prove interference bounds and in consequence approximation guarantees in terms of schedulability for any stationary gang assignment algorithm that uses the following algorithm as a basis.

We first sort the tasks according to the relative deadlines. Starting from the highest-priority task, we consider each of the possible stationary gang assignment candidates $A_k^0, A_k^1, \ldots, A_k^{M-1}$ and check whether it is feasible to assign task $\tau_k$ to the consecutive gang assignment. It starts from $\ell = 0, 1, \ldots, M-1$. If the consecutive gang assignment candidate $A_k^\ell$ is feasible, we assign the gang task to the consecutive gang assignment; otherwise, we move to the next candidate. If none of the $M$ possible consecutive gang assignments is feasible, this assignment step fails and the algorithm returns failure.

In Theorem 16, we assume that all stationary gang assignments $A_i$ are given for all tasks with higher priority than $\tau_k$. Based on the information, we need to calculate $V_{i,k}$, $\delta(A_k)$, and $\delta(A_i)$ before using Theorem 16.

To facilitate efficient implementation, we use a matrix representation to indicate whether task $\tau_i$ is assigned on processor $P_j$. Let $\rho$ be a $n \times M$ matrix in which

$$\rho(i,j) := \begin{cases} True & P_j \in A_i \\ False & P_j \notin A_i \end{cases}. \tag{14}$$

Given the stationary gang assignment matrix $\rho$, the algorithm constructs the interference matrix

$$\Gamma(i,j) := \begin{cases} True & A_i \cap A_j \neq \emptyset \\ False & \text{otherwise} \end{cases} \tag{15}$$

by the boolean matrix multiplication $\rho \cdot \rho^T$, where $\rho^T$ is the transpose matrix of $\rho$. That is, the multiplication operation of two elements is replaced with the *logical and* operation and the addition operation of two elements is replaced with a *logical or* operation. More precisely, each entry in the interference matrix is computed as follows:

$$\Gamma(i,j) = \bigvee_{m=0}^{M-1} \rho(i,m) \wedge \rho(j,m)$$

which is true only if task $\tau_i$ and task $\tau_j$ share at least one processor in their stationary gang assignments. The asymptotic time-complexity for the matrix multiplication is given by $\mathcal{O}(n^2 M)$. The space complexity is given by $\mathcal{O}(nM)$.

The transformation of the higher-priority tasks in $\mathbb{T}$ into $\Psi_k$, which is later needed to construct $\Psi_k^{sus}$, can be done by the following operation:

$$\begin{cases} \tau_i \in \Psi_k & \text{if } \bigvee_{\ell=1}^{i-1} \Gamma(\ell,i) \wedge \overline{\Gamma(\ell,k)} \\ \tau_i \notin \Psi_k & \text{otherwise} \end{cases} \tag{16}$$

We now analyze the time-complexity of Algorithm 1. Line 4 requires $\mathcal{O}(i)$ for each task $\tau_i$ and therefore $\mathcal{O}(k^2)$ for one iteration. Line 5 requires to calculate the right-hand side of Equation (6), which can be done in $\mathcal{O}(1)$ if we only take $R_i - C_i$ or $\mathcal{O}(i)$ if both terms are evaluated in Equation (6) for a task $\tau_i \in \Psi_k$. Therefore, Line 5 in one iteration requires $\mathcal{O}(k^2)$. The schedulability test in Line 6 from Corollaries 17, 18 and 19 is $\mathcal{O}(kD_k)$. Line 7 is $\mathcal{O}(M)$. Since the loop can run up to $\mathcal{O}(nM)$ iterations, the time complexity is $\mathcal{O}(nM^2 + n^3 M D_n)$.

## 5 Evaluation

In this section, we present evaluations with synthetically generated gang task sets to evaluate our proposed algorithm (denoted as *OUR-DM* here) against the current state-of-the art by Dong and Liu [16] for sporadic implicit-deadline gang task systems under global EDF. Specifically, we compare to the optimized schedulability test in [16], denoted as *DONG-OPT*, based on the acceptance ratio, i.e., the number of schedulable task sets compared to the number of tested task sets.

**Algorithm 1** Deadline-Monotonic Stationary GANG Schedulability Analysis and Assignment.

```
Sort task set $\mathbb{T}$ such that $D_i \le D_j$ for $i < j$ (ties are broken arbitrarily);
for $k$ in $\{1, 2, \ldots, n\}$ do {Loop tasks.}
  for $\ell \in \{0, 1, \ldots, M-1\}$ do {Loop candidates.}
    Generate $\Psi_k$ given the candidate $A_k^\ell$ from Def. 20;
    Transform $\Psi_k$ to $\Psi_k^{sus}$ using Def. 15;
    if $(C_k, D_k, T_k) \cup \Psi_k^{sus}$ is schedulable according to any self-suspension aware uni-
    processor schedulability test (from Cor. 17, 18 and 19) then
      Assign $A_k \leftarrow A_k^\ell$;
      break;
  return No feasible stationary gang assignments found;
return Feasible stationary gang assignment $A_i$ for each task $\tau_i$;
```

We also evaluate our algorithm for sporadic constrained-deadline gang task systems under different settings of gang sizes, but without comparison due to the absence of research results for constrained-deadline gang tasks. In these experiments, we seek to explore how much the imposed constraints in terms of stationary gang assignments and fixed-priority scheduling algorithms impact the schedulability of the tested task sets.

## 5.1 Experimental Setup

We generate synthetic task sets of sporadic gang tasks with implicit- and constrained-deadlines in the following way. To generate the task sets, we use the UUniFast algorithm [7] to draw $n$ samples of $x_i = E_i \cdot C_i/MT_i$ uniform at random where $x_i \in (0, 1]$ such that $\sum_{i=1}^{n} x_i = x$ for $x \in \{0.05, 0.1, 0.15, \ldots, 1\}$. Moreover, the periods $T_i$ are drawn from a log-uniform distribution in the range of $[10, 100]$ ms.

The generated task sets are classified by the range of admissible gang sizes into *light*, *moderate*, and *heavy*. We differentiate two different settings for these gang sizes:

1. **Setting I** - with variable gang sizes: In the first setting, each *light* gang task can have a gang size in $[1, M/8]$, a *moderate* task can gang size in $[1, M/4]$, and a *heavy* task can have gang size in $[M/8, M/2]$.
2. **Setting II** - with fixed gang sizes: In this setting, a fixed gang size number is assigned to each task of a category. Namely, each *light* task has gang size $M/8$, each *moderate* task has gang size $M/4$ and each *heavy* task has a gang size $3M/8$.

We avoid the generation of too heavy tasks, since in these cases the scheduling problem is degraded to uniprocessor scheduling.[2] With respect to constrained-deadlines, we only demonstrate our proposed algorithm by a case of variable gang sizes (Setting I) in Figure 7 and a case of fixed gang sizes (Setting II) in Figure 8.

[2] Dong and Liu [16] also performed their evaluations for gang size in $[5M/8, M]$ for all tasks. This configuration is not considered here as this setup implies that there is no possibility to concurrently execute two gang tasks in parallel due to the imposed gang size. The problem becomes equivalent to uniprocessor scheduling by viewing all processors as one virtual group. In this case, preemptive EDF is the optimal solution and the classical timing analysis for uniprocessor EDF scheduling can be applied.

## 5.2 Evaluation Results

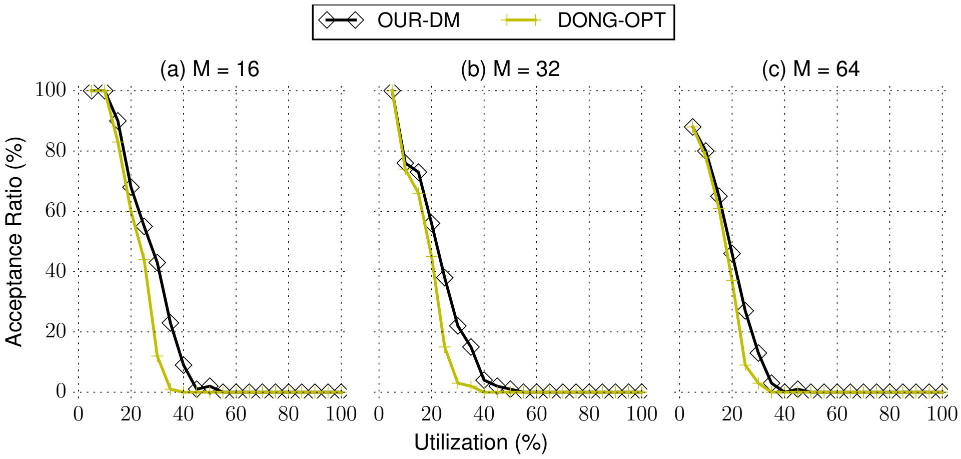


**Figure 4** Acceptance ratio for *light* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according *Setting II*.

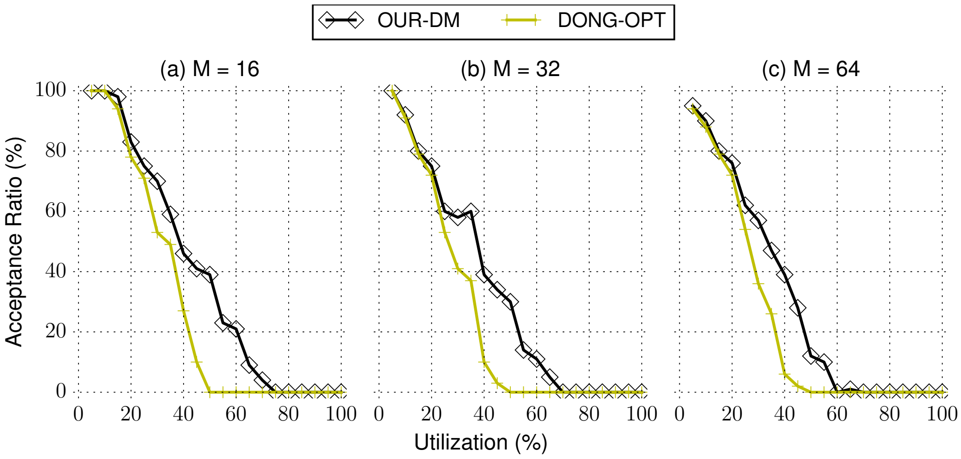


**Figure 5** Acceptance ratio for *moderate* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according to *Setting I*.

### 5.2.1 Evaluation results for implicit-deadline task sets

For sporadic implicit-deadline gang task systems, we compare our algorithm ($OUR\text{-}DM$) with the approach by Dong and Liu [16] ($DONG\text{-}OPT$) under the setting with variable gang sizes, in which each configuration is evaluated with 100 task sets and 20 tasks for each task set. In all conducted experiments shown in Figures 4, 5, and 6, our algorithm *OUR-DM* outperforms *DONG-OPT* for all evaluated scenarios under the setting with variable gang sizes. The most significant improvement of *OUR-DM* compared with *DONG-OPT* is demonstrated for the *moderate* task set in Figure 5 where up to 40% can be achieved for 50% normalized utilization. The smallest improvement can be observed for *heavy* gang task sets, where *OUR-DM* slightly outperforms *DONG-OPT*. This is due to the fact that the heavier the task sets are, the more similar the schedulability is to the uniprocessor schedulability problem. This also implies

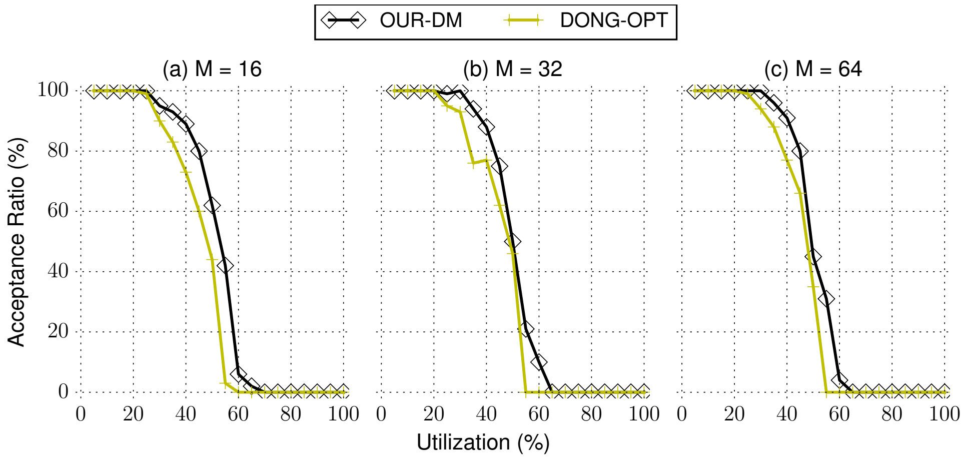


**Figure 6** Acceptance ratio for *heavy* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according to *Setting I*.

that the stationary gang scheduling has less choices for gang assignments. Since EDF is an optimal uniprocessor schedulability, the trouble to deal with the heavy gang task sets comes from the adopted schedulability tests. For *OUR-DM*, we have to consider more tasks in $\Psi_k$ and for *DONG-OPT* their analysis becomes less pessimistic as the multiplicative of $1/M$ in their analysis decreases.

### 5.2.2 Evaluation results for constrained-deadline task sets

For constrained-deadlines, we show our schedulability test for light, moderate, and heavy task sets for gang sizes compliant to Setting I in Figure 7 and gang sizes compliant to the Setting II described in Figure 8, in which each configuration is tested with 100 task sets and 20 tasks per task set. The behavior of Setting I is almost similar to the results in Figures 4, 5, and 6 but with lower acceptance ratios.

For constrained-deadlines with fixed numbers of gang sizes as explained in Setting II, a similar trend can be observed. However, moderate as well as heavy task sets almost show the same acceptance ratio and the acceptance ratio of light tasks also increases. This further supports the assumption, that the increased number of tasks with self-suspension behaviour decreases the overall schedulability. This is explained by the fact that it is less likely to have self-suspension behaviour of interfering tasks if all tasks have the same gang size.

## 5.3 Summary of Evaluation Results

In summary, the evaluations demonstrate, that the restriction of fixed-priority stationary gang scheduling does not significantly sacrifice the schedulability of sporadic implicit-deadline rigid gang task systems, in comparison to the state-of-the-art. In contrast, the schedulability could be improved slightly without even considering performance benefits of implementations in real systems, e.g., reduced context switches and migrations.

# 6 Conclusion and Future Work

In this paper we propose a specialization of the rigid gang scheduling problem for hard real-time systems. We present how this problem can be analyzed and reduced to the uniprocessor self-suspension problem and the schedulability analyses thereof. We show how to derive

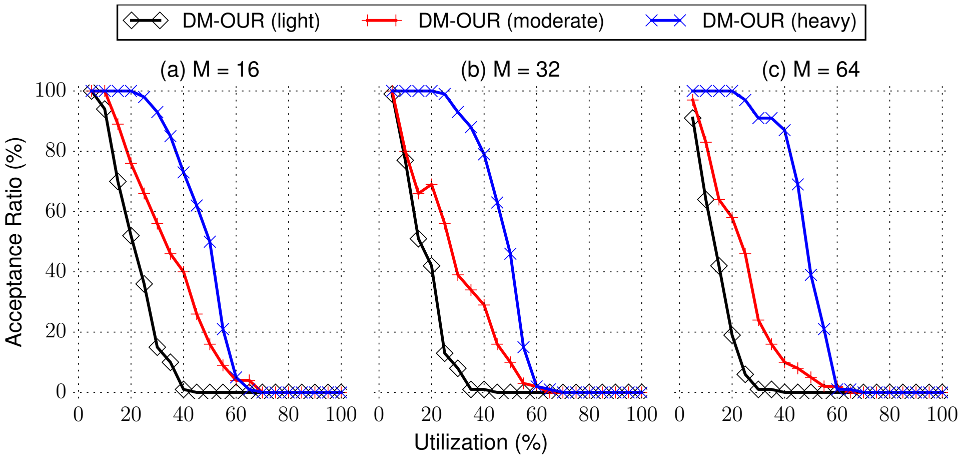


**Figure 7** Acceptance ratio for *light* sporadic constrained-deadline gang task sets according to *Setting I*. The deadline is chosen randomly between 70% – 100% of the minimum inter-arrival time.

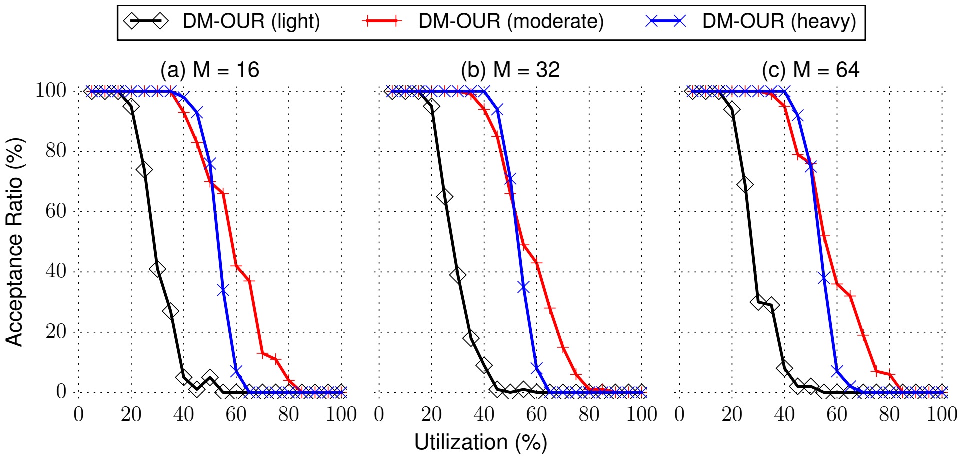


**Figure 8** Acceptance ratio for *light*, *moderate*, *heavy* sporadic constrained-deadline gang task sets according to *Setting I*. The deadline is chosen randomly between 70% – 100% of the minimum inter-arrival time.

stationary gang assignments for deadline-monotonic gang scheduling that yields worst-case interference bounds proportional to parameters defined by the ratios of the gang sizes of tasks in the task set.

This paper is limited to constrained-deadline task systems, as there is no result known for schedulability analyses for arbitrary-deadline dynamic self-suspending task systems. The concept in this paper can be extended to EDF by adopting the proper schedulability tests and suspension analysis. As future work, we plan to implement a fixed-priority stationary gang scheduler in real-time operating systems and evaluate if there are significant benefits in terms of scheduling overheads and investigate potential benefits with respect to improved worst-case execution time.

## References

1 W. Ali and H. Yun. RT-Gang: Real-time gang scheduling framework for safety-critical systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 143–155, 2019. doi:10.1109/RTAS.2019.00020.

2 Theodore P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 120–129, 2003. doi:10.1109/REAL.2003.1253260.

3 Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128, 2007.

4 Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 1323–1328, 2015. URL: http://dl.acm.org/citation.cfm?id=2757121.

5 Sanjoy Baruah. Federated scheduling of sporadic DAG task systems. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 179–186, 2015. doi:10.1109/IPDPS.2015.33.

6 Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2007. doi:10.1109/RTSS.2007.31.

7 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.

8 Alessandro Biondi and Youcheng Sun. On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. *Real Time Syst.*, 54(3):515–536, 2018. doi:10.1007/s11241-018-9303-1.

9 J. Błażewicz, P. Dell' Olmo, M. Drozdowski, and M.G. Speranza. Corrigendum to: Scheduling multiprocessor tasks on three dedicated processors. *Inf. Process. Lett.*, 49(5):269–270, 1994.

10 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *ECRTS*, pages 225–233, 2013.

11 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *RTSS*, pages 421–433. IEEE Computer Society, 2018.

12 Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 61–71, 2016.

13 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real Time Syst.*, 55(1):144–207, 2019. doi:10.1007/s11241-018-9316-9.

14 Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Cong Liu. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2017, Hsinchu, Taiwan, August 16-18, 2017*, pages 1–10. IEEE Computer Society, 2017. doi:10.1109/RTCSA.2017.8046321.

15 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.

16 Zheng Dong and Cong Liu. Analysis techniques for supporting hard real-time sporadic gang task systems. In *IEEE Real-Time Systems Symposium, RTSS*, pages 128–138, 2017. doi:10.1109/RTSS.2017.00019.

17 Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

18 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017. doi:10.1145/3139258.3139288.

19 José Carlos Fonseca, Geoffrey Nelissen, Vincent Nélis, and Luís Miguel Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *SIES*, pages 290–299. IEEE, 2016.

20 Joël Goossens and Pascal Richard. Optimal scheduling of periodic gang tasks. *LITES*, 3(1):04:1–04:18, 2016. doi:10.4230/LITES-v003-i001-a004.

21 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.

22 J.A. Hoogeveen, S.L. van de Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Appl. Math.*, 55(3):259–272, 1994.

23 Morris A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, 1997.

24 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *IEEE Real-Time Systems Symposium, RTSS*, pages 459–468, 2009. doi:10.1109/RTSS.2009.42.

25 M. Kubale. The complexity of scheduling independent two-processor tasks on dedicated processors. *Inf. Process. Lett.*, 24(3):141–147, 1987.

26 Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 259–268, 2010.

27 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.

28 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, 2015.

29 Aloysius K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

30 Pascal Richard, Joël Goossens, and Shinpei Kato. Comments on "gang EDF schedulability analysis". *CoRR*, http://arxiv.org/abs/1705.05798, 2017. URL: http://arxiv.org/abs/1705.05798.

31 Youcheng Sun and Marco Di Natale. Assessing the pessimism of current multicore global fixed-priority schedulability analysis. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC*, pages 575–583. ACM, 2018.

32 Saud Wasly and Rodolfo Pellizzoni. Bundled scheduling of parallel real-time tasks. In *RTAS*, pages 130–142. IEEE, 2019.

# Tight Tardiness Bounds for Pseudo-Harmonic Tasks Under Global-EDF-Like Schedulers

Shareef Ahmed ✉ 🏠
University of North Carolina at Chapel Hill, NC, USA

James H. Anderson ✉
University of North Carolina at Chapel Hill, NC, USA

## Abstract

The global earliest-deadline-first (GEDF) scheduler and its variants are soft-real-time (SRT) optimal for periodic/sporadic tasks, meaning they provide bounded tardiness so long as the underlying platform is not over-utilized. Although their SRT-optimality has long been known, tight tardiness bounds for these schedulers have remained elusive. In this paper, a tardiness bound, that does not depend on the processor or task count, is derived for pseudo-harmonic periodic tasks, which are commonly used in practice, under global-EDF-like (GEL) schedulers. This class of schedulers includes both GEDF and first-in-first-out (FIFO). This bound is shown to be generally tight via an example. Furthermore, it is shown that exact tardiness bounds for GEL-scheduled pseudo-harmonic periodic tasks can be computed in pseudo-polynomial time.

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

**Keywords and phrases** soft real-time systems, multicore, tardiness bounds

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.11

**Funding** Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

## 1 Introduction

The rise of multicore platforms has generated much interest in global schedulers such as the global earliest-deadline-first (GEDF) scheduler. Although the preemptive uniprocessor earliest-deadline-first (EDF) scheduler is *hard real-time (HRT) optimal*, meaning it can schedule any task system that does not over-utilize the underlying platform without any deadline misses, preemptive GEDF[1] is not HRT-optimal [10]. Despite this, GEDF and many of its variants guarantee bounded tardiness on different types of multiprocessor platforms for any task system that does not over-utilize the platform [9, 20, 26, 28], making them *soft real-time (SRT) optimal*. The significance of GEDF's SRT-optimality is reflected by references to it in the documentation of `SCHED_DEADLINE` [13], Linux's GEDF implementation.

Unfortunately, all known tardiness bounds for GEDF and its variants increase with respect to the number of processors. Moreover, experimental evaluations have shown that these bounds tend to become looser as the processor count increases [27]. This causes the corresponding SRT guarantees to be of questionable utility on large platforms and may even increase system cost. For example, the ill effects of tardiness can be "hidden" by buffering [11, 16] and the needed buffers must be sized based upon established tardiness bounds. While HRT-optimal schedulers can ameliorate these problems by eliminating all tardiness, they come at the expense of large overheads [2, 4, 5, 23]. Hence, a tardiness bound that does not scale with the number of tasks or processors under practical global schedulers like GEDF would be desirable; the derivation of such a bound has remained an open problem since the first work on SRT-optimality in 2005 [8].

[1] All schedulers mentioned herein are assumed to be preemptive unless noted otherwise.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 11; pp. 11:1–11:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we close this problem for an important category of task systems, namely *pseudo-harmonic periodic task systems*, where every period divides the maximum period. In particular, we establish a tight tardiness bound for pseudo-harmonic periodic tasks under *global-EDF-like* (GEL) schedulers on identical multiprocessor platforms. Our tardiness bound does not depend on the processor or task count, but scales with respect to the task parameters, e.g, periods. The class of GEL schedulers includes not only GEDF, but first-in-first-out (FIFO) and various other related schedulers. Pseudo-harmonic tasks are common in automotive applications [17]. Moreover, the class of pseudo-harmonic task systems contains *harmonic task systems*, where every period is an integer multiple of each smaller period. Harmonic task systems are common in different application domains such as avionics, robotics, control applications, etc. [3, 6, 14, 21, 24]. To our knowledge, we are the first to establish a tardiness bound that is tight in general for a class of task systems of practical interest under a job-level fixed-priority global scheduler. Our work was inspired by prior seminal work on the periodic behavior of GEDF schedules for HRT periodic systems [7, 15, 22].

**Prior work.** The SRT-optimality of GEDF on identical multiprocessor platforms was first shown by Devi and Anderson [9]. A tighter tardiness bound under GEDF can be obtained by *compliant vector analysis* (CVA), proposed by Erickson et. al [11,12]. The current best-known GEDF tardiness bound, the *harmonic bound*, was given by Valente [27]. *Window-constrained* schedulers, a class of schedulers containing all GEL schedulers, were proven to be SRT-optimal by Leontyev and Anderson [20]. Recent works have established the SRT-optimality of both GEDF on uniform heterogeneous multiprocessor platforms, and window-constrained schedulers on identical multiprocessor platforms with arbitrary affinity masks [25, 28].

**Contributions.** Our contributions are four-fold. First, we give a tardiness bound that is independent of the task or processor count for pseudo-harmonic periodic tasks under GEL schedulers. In a GEL scheduler, each task has a task-level fixed parameter called its *relative priority point*, which is used to assign a *priority point (PP)* to each of its jobs: the PP of a job is determined by adding its task's relative PP to the job's release time. The priority of a job is determined by its PP, with earlier PPs denoting higher priority. For example, under GEDF (resp., FIFO), a job's PP is given by its deadline (resp., release time). Additionally, we show that our bound can be exploited to ensure tardiness bounds that do not depend on the processor count for pseudo-harmonic sporadic tasks by using periodic servers scheduled by a GEL scheduler. Second, we show the general tightness of our bound by an example. Third, we give an upper bound on the length of the interval that needs to be simulated to derive an exact tardiness bound of any task in a pseudo-harmonic periodic task system. Using this, we show how to determine exact tardiness bounds in pseudo-polynomial time. To our knowledge, this is the first work on GEL schedulers that shows how to bound tardiness exactly. Fourth, we compare both of our bounds with each other and prior bounds by simulation experiments.

**Organization.** In the rest of this paper, we give necessary background information (Sec. 2), derive a tight tardiness bound for GEL schedulers (Sec. 3), show how to determine exact tardiness bounds in pseudo-polynomial time via schedule simulation (Sec. 4), discuss our experimental results (Sec. 5), and conclude (Sec. 6).

## 2 Preliminaries

We consider a task system $\tau$ consisting of $n$ implicit-deadline periodic *tasks* $\tau_1, \tau_2, \ldots, \tau_n$ to be scheduled on $m$ identical processors. Each task $\tau_i$ releases a potentially infinite sequence of *jobs* $\tau_{i,1}, \tau_{i,2}, \ldots$. The *period* of task $\tau_i$, denoted by $T_i$, is the separation time between two consecutive job releases by it. The largest period among all tasks is denoted by $T_{max}$. A task system is called *sporadic* when the separation time between consecutive jobs of each task $\tau_i$ can be more than $T_i$. The *worst-case execution cost* of $\tau_i$ is denoted by $C_i$. The *offset* of a periodic task $\tau_i$, denoted by $\Phi_i$, is the release time of $\tau_{i,1}$. The *relative deadline* of $\tau_i$ is $D_i = T_i$. For brevity, we denote a periodic task $\tau_i$ by $(\Phi_i, C_i, T_i)$.

The *release time*, *absolute deadline*, *completion time*, and *execution cost* of job $\tau_{i,k}$ are denoted by $r_{i,k}, d_{i,k}, f_{i,k}$, and $C_{i,k}$, respectively. The jobs of each task are sequential, i.e., $\tau_{i,k+1}$ cannot start execution before $\tau_{i,k}$ completes. The *tardiness* of a job $\tau_{i,k}$ is defined as $\max\{0, f_{i,k} - d_{i,k}\}$. The tardiness of task $\tau_i$ is the maximum tardiness among any of its jobs.

The *utilization* of $\tau_i$ is $u_i = C_i/T_i$. The *utilization* of the task system $\tau$ is $U = \sum_{i=1}^{n} u_i$. We require $u_i \le 1.0$ and $U \le m$ to hold; both are necessary for bounded tardiness [9]. The *hyperperiod* $H$ is the least common multiple of all periods. The periods are *pseudo-harmonic* when each period divides $T_{max}$, i.e., $H = T_{max}$ holds.

The relative PP of a task $\tau_i$ is denoted by $Y_i$. We assume $Y_i \ge 0$ holds for each task $\tau_i$. The maximum and minimum relative PP among all tasks in $\tau$ are denoted by $Y_{max}$ and $Y_{min}$, respectively. The *priority point (PP)* of a job $\tau_{i,k}$, denoted by $y_{i,k}$, is defined as

$$y_{i,k} = r_{i,k} + Y_i. \tag{1}$$

If $y_{i,k} < y_{j,\ell}$, then job $\tau_{i,k}$ has higher priority than job $\tau_{j,\ell}$. We assume ties to be broken arbitrarily but consistently by task index.

We assume time to be discrete and a unit of time to be 1.0. All scheduling decisions are taken at integer points in time. We also assume all task parameters to be integers. Therefore, when a task $\tau_i$ executes during an unit interval $[t-1, t)$, it means $\tau_i$ continuously executes during $[t-1, t)$. A job completes execution at $t$ if it executes for the last time during $[t-1, t)$. A job completes execution before $t$ if it completes at or before $t-1$. (It can be shown that the tardiness bound presented in Sec. 3 also holds when time is continuous.) The following definitions closely follow from material in [1, 9, 28].

▶ **Definition 1.** *A job $\tau_{i,k}$ is* active *at time $t$ in a schedule $\mathcal{S}$ if $r_{i,k} \le t < d_{i,k}$.*

▶ **Definition 2.** *A job $\tau_{i,k}$ is* pending *at time $t$ in a schedule $\mathcal{S}$ if $r_{i,k} \le t$ and $\tau_{i,k}$ has not completed execution at or before $t$ in $S$.*

**Allocation.** The cumulative processor capacity allocated to a task $\tau_i$ (resp., task system $\tau$) in a schedule $\mathcal{S}$ over an interval $[t, t')$ is denoted by $\mathsf{A}_i(t, t', \mathcal{S})$ (resp., $\mathsf{A}(t, t', \mathcal{S})$). Thus, $\mathsf{A}(t, t', \mathcal{S}) = \sum_{\tau_i \in \tau} \mathsf{A}_i(t, t', \mathcal{S})$.

**Ideal schedule.** Let $\widehat{\pi}_1, \widehat{\pi}_2, \ldots, \widehat{\pi}_n$ be $n$ processors with speeds $u_1, u_2, \ldots, u_n$, respectively. In an *ideal schedule* $\mathcal{I}$, each task $\tau_i$ is partitioned to execute on processor $\widehat{\pi}_i$. Each job starts execution as soon as it is released and completes execution by its deadline in $\mathcal{I}$. For task $\tau_i$ (resp., task system $\tau$), $\mathsf{A}_i(t, t', \mathcal{I}) \le u_i(t' - t)$ (resp., $\mathsf{A}(t, t, \mathcal{I}) \le U(t' - t)$) holds. If $\tau_i$ is periodic and each job executes for its worst case $C_i$, then $\mathsf{A}_i(t, t', \mathcal{I}) = u_i(t' - t)$ where $t, t' \ge \Phi_i$.

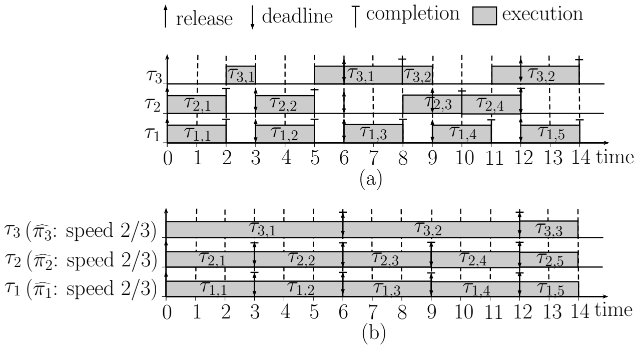


**Figure 1** (a) A GEDF schedule and (b) an ideal schedule of the task system in Ex. 3.

**lag and LAG.** The lag of a task $\tau_i$ at time $t$ in a schedule $\mathcal{S}$ is defined as

$$\mathsf{lag}_i(t, \mathcal{S}) = \mathsf{A}_i(0, t, \mathcal{I}) - \mathsf{A}_i(0, t, \mathcal{S}). \tag{2}$$

Since $\mathsf{lag}_i(0, \mathcal{S}) = 0$, for $t' \geq t$ we have

$$\mathsf{lag}_i(t', \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S}) + \mathsf{A}_i(t, t', \mathcal{I}) - \mathsf{A}_i(t, t', \mathcal{S}). \tag{3}$$

The LAG of a task system $\tau$ in a schedule $\mathcal{S}$ at time $t$ is defined as

$$\mathsf{LAG}(t, \mathcal{S}) = \sum_{\tau_i \in \tau} \mathsf{lag}_i(t, \mathcal{S}) = \mathsf{A}(0, t, \mathcal{I}) - \mathsf{A}(0, t, \mathcal{S}). \tag{4}$$

Since $\mathsf{LAG}(0, \mathcal{S}) = 0$, for $t' \geq t$ we have

$$\mathsf{LAG}(t', \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S}) + \mathsf{A}(t, t', \mathcal{I}) - \mathsf{A}(t, t', \mathcal{S}). \tag{5}$$

▶ **Example 3.** Consider a periodic task system $\tau$ with tasks $\tau_1 = (0, 2, 3)$, $\tau_2 = (0, 2, 3)$, and $\tau_3 = (0, 4, 6)$. A GEDF schedule $\mathcal{S}$ and an ideal schedule $\mathcal{I}$ of the task system is shown in insets (a) and (b) of Fig. 1, respectively. $\tau_1$'s allocation over interval $[0, 5)$ in $\mathcal{S}$ and $\mathcal{I}$ are 4.0 and 10/3 execution units, respectively. $\tau_1$'s lag in $\mathcal{S}$ at time 5 is $\mathsf{lag}_1(5, \mathcal{S}) = 10/3 - 4 = -2/3$. The LAG of the task system $\tau$ at time 5 is 1.0. ◀

## 3 Tardiness Bound

In this section, we derive a tardiness bound for pseudo-harmonic periodic task systems under a GEL scheduler. We assume $n > m$ as each job meets its deadline otherwise. We initially assume the following, which we relax later.

**(B)** Each job of any task $\tau_i$ executes for its worst-case execution cost $C_i$.

We consider a GEL schedule $\mathcal{S}$ of $\tau$ satisfying (B) to derive our tardiness bound. We derive our tardiness bound (Theorem 28) by giving an upper bound on per-task lag (Lemma 27) using a lag-*monotonicity* property (Lemma 17). Informally, the lag-monotonicity property states that no task $\tau_i$ receives more allocation in $\mathcal{S}$ than $\mathcal{I}$, i.e., lag does not decrease, over any interval of length $T_{max}$ beginning at or after $\Phi_i$. We first establish the lag-monotonicity property using a series of properties of lag proved in Sec. 3.1. We then use the lag-monotonicity property to derive our tardiness bound in Sec. 3.2.

## 3.1 lag Properties

We begin by proving some properties of $\mathsf{lag}$. All properties specified here also hold for non-pseudo-harmonic periodic task systems satisfying (B) with $T_{max}$ replaced by $H$ in the properties that reference $T_{max}$. Since the $\mathsf{lag}$-monotonicity property compares $\mathsf{lag}$ values between two time instants, we first establish several properties concering such comparisons between a pair of $\mathsf{lag}$ values (Lemmas 11–15) based on the simpler properties of $\mathsf{lag}$ (Lemmas 4–10). Readers familiar with the concept of $\mathsf{lag}$ may skip the proofs of Lemmas 4–10.

▶ **Lemma 4.** *For any task $\tau_i$ and interval $[t, t')$ with $t \geq \Phi_i$, the following hold.*
*(a) If $\tau_i$ continuously executes during $[t, t')$ in $\mathcal{S}$, then $\mathsf{lag}_i(t', \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S}) + (t' - t)(u_i - 1)$.*
*(b) If $\tau_i$ does not execute during $[t, t')$ in $\mathcal{S}$, then $\mathsf{lag}_i(t', \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S}) + (t' - t)u_i$.*

**Proof.** Since $t \geq \Phi_i$, by the definition of $\mathcal{I}$, we have $\mathsf{A}_i(t, t', \mathcal{I}) = (t' - t)u_i$.
(a) Since $\tau_i$ continuously executes throughout $[t, t')$ in $\mathcal{S}$, $\mathsf{A}_i(t, t', \mathcal{S}) = (t' - t)$ holds. Substituting $\mathsf{A}_i(t, t', \mathcal{I})$ and $\mathsf{A}_i(t, t', \mathcal{S})$ in (3), we have $\mathsf{lag}_i(t', \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S}) + (t' - t)u_i - (t' - t) = \mathsf{lag}_i(t, \mathcal{S}) + (t' - t)(u_i - 1)$.
(b) Since $\tau_i$ does not execute during $[t, t')$ in $\mathcal{S}$, we have $\mathsf{A}_i(t, t', \mathcal{S}) = 0$. Substituting $\mathsf{A}_i(t, t', \mathcal{I})$ and $\mathsf{A}_i(t, t', \mathcal{S})$ in (3), we have $\mathsf{lag}_i(t', \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S}) + (t' - t)u_i - 0 = \mathsf{lag}_i(t, \mathcal{S}) + (t' - t)u_i$. ◀

▶ **Lemma 5** ([28]). *If $\mathsf{lag}_i(t, \mathcal{S}) > 0$, then $\tau_i$ has a pending job at $t$ in $\mathcal{S}$.*

The following lemma states that the $\mathsf{lag}$ of any task $\tau_i$ is non-negative in $\mathcal{S}$ at any time instant $t$ when it releases a job. Intuitively, all of $\tau_i$'s jobs released before $t$ complete execution in $\mathcal{I}$ by time $t$, and thus, $\tau_i$ cannot receive more allocation in $\mathcal{S}$ than $\mathcal{I}$.

▶ **Lemma 6.** *For any task $\tau_i$ and non-negative integer $c$, $\mathsf{lag}_i(\Phi_i + cT_i, \mathcal{S}) \geq 0$.*

**Proof.** If $c = 0$, then the lemma trivially holds. Assume that there is a task $\tau_i$ and an integer $c \geq 1$ such that $\mathsf{lag}_i(\Phi_i + cT_i, \mathcal{S}) < 0$ holds. Then, by (2), $\mathsf{A}_i(0, \Phi_i + cT_i, \mathcal{S}) > \mathsf{A}_i(0, \Phi_i + cT_i, \mathcal{I})$ holds. Since $\tau_i$ releases periodically, $\Phi_i + cT_i$ is the deadline (resp., release time) of $\tau_{i,c-1}$ (resp., $\tau_{i,c}$). By the definition of $\mathcal{I}$, all jobs of $\tau_i$ released before $\Phi_i + cT_i$ complete execution by time $\Phi_i + cT_i$ in $\mathcal{I}$. Since no job can execute before its release, $\mathsf{A}_i(0, \Phi_i + cT_i, \mathcal{S})$ cannot be larger than $\mathsf{A}_i(0, \Phi_i + cT_i, \mathcal{I})$, a contradiction. ◀

Lemmas 7–10 give relationships among a task $\tau_i$'s $\mathsf{lag}$ at time $t$, its utilization, and the deadline or release time of a job of $\tau_i$. We prove these lemmas by expressing $\tau_i$'s allocation in $\mathcal{S}$ by time $t$ in terms of $\tau_i$'s utilization and the deadline or release time of a job of $\tau_i$.

▶ **Lemma 7.** *If $\tau_i$ has no pending job at time $t \geq \Phi_i$ in $\mathcal{S}$ and $\tau_{i,k}$ is the active job of $\tau_i$ at $t$, then $\mathsf{lag}_i(t, \mathcal{S}) = (t - d_{i,k})u_i$.*

**Proof.** Since $\tau_{i,k}$ completes execution at or before $t$, all jobs of $\tau_i$ released at or before $r_{i,k}$ complete execution at or before $t$. Since $t < d_{i,k}$, no jobs released after $r_{i,k}$ execute before $t$. Hence, $\mathsf{A}_i(0, t, \mathcal{S}) = \sum_{j=1}^{k} C_i = \sum_{j=1}^{k} T_i u_i = \sum_{j=1}^{k} (r_{i,j+1} - r_{i,j})u_i = (r_{i,k+1} - r_{i,1})u_i = (d_{i,k} - \Phi_i)u_i$. By the definition of $\mathcal{I}$, we have $\mathsf{A}_i(0, t, \mathcal{I}) = (t - \Phi_i)u_i$. Substituting $\mathsf{A}_i(0, t, \mathcal{I})$ and $\mathsf{A}_i(0, t, \mathcal{S})$ in (2), we have $\mathsf{lag}_i(t, \mathcal{S}) = (t - \Phi_i)u_i - (d_{i,k} - \Phi_i)u_i = (t - d_{i,k})u_i$. ◀

For the task set in Ex. 3 and its $\mathsf{GEDF}$ schedule in Fig. 1(a), $\tau_1$'s active job at time 2 is $\tau_{1,1}$ and it has no pending job at time 2 in $\mathcal{S}$. The $\mathsf{lag}$ of $\tau_1$ at time 2 in $\mathcal{S}$ is $\mathsf{lag}_1(2, \mathcal{S}) = (2 - 3)\frac{2}{3} = -\frac{2}{3}$.

▶ **Lemma 8.** *If $\tau_{i,k}$ completes execution at or before $t \geq \Phi_i$ in $\mathcal{S}$, then $\mathsf{lag}_i(t, \mathcal{S}) \leq (t - d_{i,k})u_i$.*

**Proof.** Since $\tau_{i,k}$ completes execution at or before $t$, all jobs of $\tau_i$ released at or before $r_{i,k}$ complete execution at or before $t$. Hence, $\mathsf{A}_i(0,t,\mathcal{S}) \geq \sum_{j=1}^{k} C_i = \sum_{j=1}^{k} T_i u_i = \sum_{j=1}^{k}(r_{i,j+1} - r_{i,j})u_i = (r_{i,k+1} - r_{i,1})u_i = (d_{i,k} - \Phi_i)u_i$. By the definition of $\mathcal{I}$, we have $\mathsf{A}_i(0,t,\mathcal{I}) = (t - \Phi_i)u_i$. Substituting $\mathsf{A}_i(0,t,\mathcal{I})$ and $\mathsf{A}_i(0,t,\mathcal{S})$ in (2), we have $\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{A}_i(0,t,\mathcal{I}) - \mathsf{A}_i(0,t,\mathcal{S}) \leq (t - \Phi_i)u_i - (d_{i,k} - \Phi_i)u_i = (t - d_{i,k})u_i$. ◀

▶ **Lemma 9.** *If $\tau_i$ has a pending job $\tau_{i,k}$ at $t \geq \Phi_i$ in $\mathcal{S}$, then $\mathsf{lag}_i(t,\mathcal{S}) > (t - d_{i,k})u_i$.*

**Proof.** Since $\tau_{i,k}$ is pending at time $t$, we have $\mathsf{A}_i(0,t,\mathcal{S}) < \sum_{j=1}^{k} C_i = \sum_{j=1}^{k} T_i u_i = \sum_{j=1}^{k}(r_{i,j+1} - r_{i,j})u_i = (r_{i,k+1} - r_{i,1})u_i = (d_{i,k} - \Phi_i)u_i$. By the definition of $\mathcal{I}$, $\mathsf{A}_i(0,t,\mathcal{I}) = (t - \Phi_i)u_i$ holds. Substituting $\mathsf{A}_i(0,t,\mathcal{I})$ and $\mathsf{A}_i(0,t,\mathcal{S})$ in (2), we have $\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{A}_i(0,t,\mathcal{I}) - \mathsf{A}_i(0,t,\mathcal{S}) > (t - \Phi_i)u_i - (d_{i,k} - \Phi_i)u_i = (t - d_{i,k})u_i$. ◀

▶ **Lemma 10.** *If $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t \geq \Phi_i$ in $\mathcal{S}$, then $\mathsf{lag}_i(t,\mathcal{S}) \leq (t - r_{i,k})u_i$.*

**Proof.** Since $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t$, all jobs of $\tau_i$ prior to $\tau_{i,k}$ complete execution at or before $t$. Thus, $\mathsf{A}_i(0,t,\mathcal{S}) \geq \sum_{j=1}^{k-1} C_i = \sum_{j=1}^{k-1} T_i u_i = \sum_{j=1}^{k-1}(r_{i,j+1} - r_{i,j})u_i = (r_{i,k} - r_{i,1})u_i = (r_{i,k} - \Phi_i)u_i$. By the definition of $\mathcal{I}$, $\mathsf{A}_i(0,t,\mathcal{I}) = (t - \Phi_i)u_i$ holds. Substituting $\mathsf{A}_i(0,t,\mathcal{I})$ and $\mathsf{A}_i(0,t,\mathcal{S})$ in (2), we have $\mathsf{lag}_i(t,\mathcal{S}) \leq (t - \Phi_i)u_i - (r_{i,k} - \Phi_i)u_i = (t - r_{i,k})u_i$. ◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\tau_{3,1}$ is $\tau_3$'s earliest pending job at at time 4. $\tau_3$'s lag at time 4 is $4 \times 2/3 - 1 = 5/3$. By Lemma 9, $\mathsf{lag}_3(4,\mathcal{S}) = 5/3 > (4-6) \times 2/3 = -4/3$. By Lemma 10, $\mathsf{lag}_3(4,\mathcal{S}) = 5/3 \leq (4-0) \times 2/3 = 8/3$.

Using Lemmas 7–10, we now prove Lemmas 11–14, which pertain to the relationship between the lag of a task $\tau_i$ at a pair of time instants that are separated by an integer multiple of $\tau_i$'s period. For any integer $c$ and any pair of time instants $t, t + cT_i \geq \Phi_i$, the active jobs of $\tau_i$ at $t$ and $t + cT_i$ receive the same allocation in $\mathcal{I}$ by time $t$ and $t + cT_i$, respectively. If $\tau_i$'s active job $\tau_{i,k}$ at $t$ completes execution in $\mathcal{S}$ at or before $t$, then $\tau_{i,k+c}$ cannot receive more allocation by time $t + cT_i$ than $\tau_{i,k}$ receives by $t$. The following lemma pertains to this scenario.

▶ **Lemma 11.** *For any time $t$ and integer $c$ such that $\min\{t, t + cT_i\} \geq \Phi_i$, if $\tau_i$ has no pending job at $t$ in $\mathcal{S}$, then $\mathsf{lag}_i(t,\mathcal{S}) \leq \mathsf{lag}_i(t + cT_i,\mathcal{S})$.*

**Proof.** Let $\tau_{i,k}$ be the active job of $\tau_i$ at $t$, i.e., $r_{i,k} \leq t < d_{i,k}$. Since $\tau_i$ has no pending job at $t \geq \Phi_i$, by Lemma 7 we have

$$\mathsf{lag}_i(t,\mathcal{S}) = (t - d_{i,k})u_i. \tag{6}$$

Since the jobs of a task are released periodically and $t + cT_i \geq \Phi_i$ holds, $\tau_{i,k+c}$ is the active job of $\tau_i$ at time $t + cT_i$. By Lemmas 7 and 9, we have

$$\begin{aligned}\mathsf{lag}_i(t + cT_i,\mathcal{S}) &\geq (t + cT_i - d_{i,k+c})u_i \\ &= \{\text{Since } \tau_i \text{ releases periodically, } d_{i,k+c} = d_{i,k} + cT_i\} \\ &\quad (t + cT_i - d_{i,k} - cT_i)u_i \\ &= (t - d_{i,k})u_i \\ &= \{\text{By (6)}\} \\ &\quad \mathsf{lag}_i(t,\mathcal{S}).\end{aligned}$$

◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\tau_2$ has no pending job at time 5 but has a pending job at time 8 in $\mathcal{S}$. By Lemma 11, $\mathsf{lag}_2(5, \mathcal{S}) = -2/3 \leq 4/3 = \mathsf{lag}_2(8, \mathcal{S})$.

The following lemma considers the case when $\mathsf{lag}_i(t, \mathcal{S})$ is not larger than $\mathsf{lag}_i(t + cT_i, \mathcal{S})$. Informally, for any non-negative integer $c$, $\tau_i$ receives no more than $cT_i u_i = c \cdot C_i$ units of allocation over the interval $[t, t + cT_i)$. Therefore, if $\tau_{i,k}$ is pending at $t$, then $\tau_{i,k+c}$ must also be pending at $t + cT_i$.

▶ **Lemma 12.** *For any integer $c$ such that $\min\{t, t+cT_i\} \geq \Phi_i$, if $\mathsf{lag}_i(t, \mathcal{S}) \leq \mathsf{lag}_i(t+cT_i, \mathcal{S})$ holds and $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t$ in $\mathcal{S}$, then $\tau_{i,k+c}$ is pending at $t+cT_i$ in $\mathcal{S}$.*

**Proof.** Assume for a contradiction that $\tau_{i,k+c}$ is not pending at time $t + cT_i$. Since $\tau_{i,k}$ is pending at $t \geq \Phi_i$, $r_{i,k} \leq t$ holds, and by Lemma 9 we have

$$\mathsf{lag}_i(t, \mathcal{S}) > (t - d_{i,k})u_i. \tag{7}$$

Since jobs are released periodically and $r_{i,k} \leq t$ holds, we have $r_{i,k+c} \leq t + cT_i$. Thus, $\tau_{i,k+c}$ finishes execution at or before $t + cT_i$ (as it is not pending then). By Lemma 8, we have

$$\begin{aligned}\mathsf{lag}_i(t + cT_i, \mathcal{S}) &\leq (t + cT_i - d_{i,k+c})u_i \\ &= \{\text{Since } \tau_i \text{ releases periodically, } d_{i,k+c} = d_{i,k} + cT_i\} \\ &\quad (t + cT_i - d_{i,k} - cT_i)u_i \\ &= (t - d_{i,k})u_i \\ &< \{\text{By (7)}\} \\ &\quad \mathsf{lag}_i(t, \mathcal{S}),\end{aligned}$$

a contradiction. ◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\mathsf{lag}_2(4, \mathcal{S}) = -1/3 \leq 2/3 = \mathsf{lag}_2(7, \mathcal{S})$. By Lemma 12, since $\tau_{2,2}$ is the earliest pending job of $\tau_2$ at time 4 in $\mathcal{S}$ and time 7 corresponds to $c = 1$, $\tau_{2,3}$ is pending at time 7 in $\mathcal{S}$.

Similarly, we consider the case where $\mathsf{lag}_i(t, \mathcal{S})$ is either not smaller or larger than $\mathsf{lag}_i(t + cT_i, \mathcal{S})$.

▶ **Lemma 13.** *For any integer $c$ such that $\min\{t, t+cT_i\} \geq \Phi_i$, if $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at time $t$ in $\mathcal{S}$, then the following hold.*

**(a)** *If $\mathsf{lag}_i(t, \mathcal{S}) \geq \mathsf{lag}_i(t+cT_i, \mathcal{S})$, then all jobs of $\tau_i$ released before $r_{i,k+c}$ complete execution at or before $t + cT_i$ in $\mathcal{S}$.*

**(b)** *If $\mathsf{lag}_i(t, \mathcal{S}) > \mathsf{lag}_i(t+cT_i, \mathcal{S})$, then all jobs of $\tau_i$ released before $r_{i,k+c}$ complete execution before $t + cT_i$ in $\mathcal{S}$.*

**Proof.** If $k + c = 1$, then $r_{i,k+c} = r_{i,1} = \Phi_i$ and the lemma trivially holds. So assume $k + c > 1$. Since $\tau_{i,k}$ is the earliest pending job at $t \geq \Phi_i$, by Lemma 10,

$$\mathsf{lag}_i(t, \mathcal{S}) \leq (t - r_{i,k})u_i. \tag{8}$$

**(a)** Assume for a contradiction that $\tau_i$ has a job that is released before $r_{i,k+c}$ but does not complete execution at or before $t + cT_i$. Therefore, $\tau_{i,k+c-1}$ does not complete execution at or before $t + cT_i$ as the jobs of each task are sequential. Since $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t$, we have $r_{i,k} \leq t$. Since jobs are released periodically, we have $r_{i,k+c} \leq t + cT_i$, which implies $r_{i,k+c-1} \leq t + cT_i$. Therefore, $\tau_{i,k+c-1}$ is pending at $t + cT_i$. Thus, by Lemma 9,

$$\mathsf{lag}_i(t + cT_i, \mathcal{S}) > (t + cT_i - d_{i,k+c-1})u_i$$

$$\begin{aligned}
&= \{\text{Since } \tau_i \text{ releases periodically, } d_{i,k+c-1} = d_{i,k} + (c-1)T_i\} \\
&\quad (t + cT_i - d_{i,k} - (c-1)T_i)u_i \\
&= (t - d_{i,k} + T_i)u_i \\
&= \{\text{Since } r_{i,k} = d_{i,k} - T_i\} \\
&\quad (t - r_{i,k})u_i \\
&\geq \{\text{By (8)}\} \\
&\quad \mathsf{lag}_i(t, \mathcal{S}),
\end{aligned}$$

a contradiction.

**(b)** Since $\mathsf{lag}_i(t,\mathcal{S}) > \mathsf{lag}_i(t+cT_i,\mathcal{S})$, by (a), all jobs of $\tau_i$ released before $r_{i,k+c}$ finish execution at or before $t+cT_i$. Assume that they do not complete execution before $t+cT_i$. Thus, they complete execution at $t+cT_i$, and no job released at or after $r_{i,k+c}$ executes at or before $t+cT_i$. Thus, $\mathsf{A}_i(0,t+cT_i,\mathcal{S}) = \sum_{j=1}^{k+c-1} C_i = \sum_{j=1}^{k+c-1} T_i u_i = \sum_{j=1}^{k+c-1}(r_{i,j+1} - r_{i,j})u_i = (r_{i,k+c} - \Phi_i)u_i = (r_{i,k} + cT_i - \Phi_i)u_i$. Thus, by the definition of $\mathcal{I}$ and (2), $\mathsf{lag}_i(t+cT_i,\mathcal{S}) = (t+cT_i-\Phi_i)u_i - (r_{i,k}+cT_i-\Phi_i)u_i = (t-r_{i,k})u_i \geq \mathsf{lag}_i(t,\mathcal{S})$, a contradiction. ◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\mathsf{lag}_2(8,\mathcal{S}) = 4/3 > 1/3 = \mathsf{lag}_2(11,\mathcal{S})$. By Lemma 13(b), since $\tau_{2,3}$ is the earliest pending job of $\tau_2$ at time 8 in $\mathcal{S}$ and time 11 corresponds to $c = 1$, all jobs of $\tau_2$ prior to $\tau_{2,4}$ complete execution before time 11 in $\mathcal{S}$.

We now utilize Lemmas 12 and 13(a) to establish a necessary condition for $\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{lag}_i(t+cT_i,\mathcal{S})$ to hold. Intuitively, if $\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{lag}_i(t+cT_i,\mathcal{S})$ holds, then in $\mathcal{S}$ any job $\tau_{i,k}$'s allocation at or before $t$ must equal the allocation of job $\tau_{i,k+c}$ at or before $t+cT_i$.

▶ **Lemma 14.** *For any time $t$ and integer $c$ such that $\min\{t, t+cT_i\} \geq \Phi_i$, if $\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{lag}_i(t+cT_i,\mathcal{S})$, then the following hold.*

**(a)** *If there is no pending job of $\tau_i$ at $t$ in $\mathcal{S}$, then there is no pending job of $\tau_i$ at $t+cT_i$ in $\mathcal{S}$.*

**(b)** *If $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t$ in $\mathcal{S}$, then $\tau_{i,k+c}$ is the earliest pending job of $\tau_i$ at $t+cT_i$ in $\mathcal{S}$.*

**Proof.**

**(a)** Assume that there is a pending job of $\tau_i$ at $t+cT_i$ and let $\tau_{i,k}$ be the earliest pending job of $\tau_i$ at $t+cT_i$. Substituting $t$ and $c$ in Lemma 12 by $t+cT_i$ and $-c$, respectively, job $\tau_{i,k-c}$ is pending at $t$, a contradiction.

**(b)** By Lemma 12, $\tau_{i,k+c}$ is pending at $t+cT_i$. By Lemma 13(a), all jobs of $\tau_i$ released before $r_{i,k+c}$ finish execution at or before $t+cT_i$. Thus, $\tau_{i,k+c}$ is the earliest pending job of $\tau_i$ at $t+cT_i$. ◀

We now give a necessary condition for the $\mathsf{lag}$-monotonicity property to not hold.

▶ **Lemma 15.** *Let $t \geq \Phi_i + T_{max}$ be the first time instant (if one exists) such that $\mathsf{lag}_i(t - T_{max},\mathcal{S}) > \mathsf{lag}_i(t,\mathcal{S})$ holds in $\mathcal{S}$. Then, the following hold.*

**(a)** *$t > \Phi_i + T_{max}$.*

**(b)** *$\tau_i$ executes during $[t-1, t)$, but does not execute during $[t - T_{max} - 1, t - T_{max})$ in $\mathcal{S}$.*

**Proof.**

**(a)** Assume that $t = \Phi_i + T_{max}$. Since $t - T_{max} = \Phi_i$, we have $\mathsf{lag}_i(t - T_{max},\mathcal{S}) = \mathsf{lag}_i(\Phi_i,\mathcal{S}) = 0$. Since $T_i$ divides $T_{max}$, by Lemma 6, we have $\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{lag}_i(\Phi_i + T_{max},\mathcal{S}) \geq 0$. Therefore, $\mathsf{lag}_i(t - T_{max},\mathcal{S}) \leq \mathsf{lag}_i(t,\mathcal{S})$, a contradiction.

**(b)** By (a), $t-1 \geq \Phi_i + T_{max}$ holds. By the definition of $t$, we have

$$\mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) \leq \mathsf{lag}_i(t-1,\mathcal{S}). \tag{9}$$

Assume that $\tau_i$ does not execute during $[t-1,t)$ or does execute during $[t-T_{max}-1, t-T_{max})$. Then, one of the following three cases holds.

**Case 1.** *$\tau_i$ executes during both $[t-T_{max}-1, t-T_{max})$ and $[t-1,t)$.* By Lemma 4(a),

$$\mathsf{lag}_i(t-T_{max},\mathcal{S}) = \mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) + u_i - 1, \tag{10}$$

and

$$\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{lag}_i(t-1,\mathcal{S}) + u_i - 1. \tag{11}$$

Since $\mathsf{lag}_i(t-T_{max},\mathcal{S}) > \mathsf{lag}_i(t,\mathcal{S})$, by (10) and (11), we have $\mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) > \mathsf{lag}_i(t-1,\mathcal{S})$, which contradicts (9).

**Case 2.** *$\tau_i$ does not execute during both $[t-T_{max}-1, t-T_{max})$ and $[t-1,t)$.* By Lemma 4(b),

$$\mathsf{lag}_i(t-T_{max},\mathcal{S}) = \mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) + u_i, \tag{12}$$

and

$$\mathsf{lag}_i(t,\mathcal{S}) = \mathsf{lag}_i(t-1,\mathcal{S}) + u_i. \tag{13}$$

Since $\mathsf{lag}_i(t-T_{max},\mathcal{S}) > \mathsf{lag}_i(t,\mathcal{S})$, by (12) and (13), we have $\mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) > \mathsf{lag}_i(t-1,\mathcal{S})$, which contradicts (9).

**Case 3.** *$\tau_i$ executes during $[t-T_{max}-1, t-T_{max})$ but does not execute during $[t-1,t)$.* Thus, (10) and (13) hold. Therefore, by (10), we have

$$\begin{aligned}
\mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) &= \mathsf{lag}_i(t-T_{max},\mathcal{S}) + 1 - u_i \\
&\geq \{\text{Since } u_i \leq 1\} \\
&\quad \mathsf{lag}_i(t-T_{max},\mathcal{S}) \\
&> \{\text{By the definition of } t\} \\
&\quad \mathsf{lag}_i(t,\mathcal{S}) \\
&\geq \{\text{By (13) and } u_i \geq 0\} \\
&\quad \mathsf{lag}_i(t-1,\mathcal{S}),
\end{aligned}$$

a contradiction to (9). ◀

▶ **Definition 16.** *Let $h_i = T_{max}/T_i$.*

The following lemma gives a $\mathsf{lag}$-monotonicity property for SRT-schedulable systems that is similar to one given previously for HRT-schedulable systems under a job-level fixed-priority scheduler [7]. Informally, we show that, using Lemmas 11–13 and 15, no task can receive more allocation in $\mathcal{S}$ than $\mathcal{I}$ over an interval $[t-T_{max}, t)$ because of the existence of higher-priority jobs of other tasks, i.e., over-allocating a task would require under-allocating another task, violating the priority ordering of the jobs.

▶ **Lemma 17.** *For any task $\tau_i$ and any time $t \geq \Phi_i + T_{max}$, $\mathsf{lag}_i(t-T_{max},\mathcal{S}) \leq \mathsf{lag}_i(t,\mathcal{S})$.*

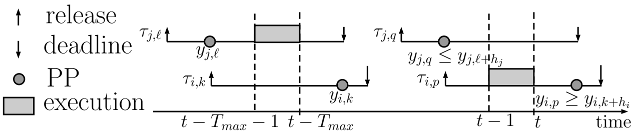


**Figure 2** Illustration of the proof of Lemma 17.

**Proof.** We use Fig. 2 to illustrate the proof. Assume for a contradiction that $t$ is the first time instant such that $t \geq \Phi_i + T_{max}$ and there is a task $\tau_i$ with $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) > \mathsf{lag}_i(t, \mathcal{S})$. By Lemma 15(b), $\tau_i$ executes during $[t-1, t)$. Let $\tau_{i,p}$ be the job of $\tau_i$ that executes during $[t-1, t)$. Since $T_i$ divides $T_{max}$, by the contrapositive of Lemma 11 (with $t$ and $c$ replaced by $t - T_{max}$ and $h_i$, respectively), there is a pending job of $\tau_i$ at $t - T_{max}$. Let $\tau_{i,k}$ be the earliest pending job of $\tau_i$ at $t - T_{max}$.

▷ Claim. $r_{i,k} < t - T_{max}$.

Proof. Assume otherwise. Then, $r_{i,k} = t - T_{max}$ and $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) = 0$ hold. Since jobs are released periodically and $t = r_{i,k} + T_{max}$ holds, there is a non-negative integer $c$ such that $t = \Phi_i + cT_i$, which by Lemma 6 implies $\mathsf{lag}_i(t, \mathcal{S}) = \mathsf{lag}_i(\Phi_i + cT_i, \mathcal{S}) \geq 0$. Therefore, $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) = 0 \leq \mathsf{lag}_i(t, \mathcal{S})$, and $t$ cannot be a time instant with $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) > \mathsf{lag}_i(t, \mathcal{S})$. Therefore, $r_{i,k} < t - T_{max}$ holds. ◁

By the above claim, $\tau_{i,k}$ is pending at $t - T_{max} - 1$. By Lemma 15(b), $\tau_{i,k}$ does not execute during $[t - T_{max} - 1, t - T_{max})$ (see Fig. 2). Since $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) > \mathsf{lag}_i(t, \mathcal{S})$, substituting $t$ and $c$ in Lemma 13(b) by $t - T_{max}$ and $h_i$ (Def. 16), respectively, all jobs of $\tau_i$ released before $r_{i,k+h_i}$ complete execution before $t$ (at or before $t-1$). Thus, $p \geq k + h_i$ and we have

$$\begin{aligned} y_{i,p} &\geq y_{i,k+h_i} \\ &= \{\text{Since } \tau_i \text{ releases periodically, } y_{i,k+h_i} = y_{i,k} + h_i T_i \text{ holds and by Def. 16}\} \\ &\quad y_{i,k} + T_{max}. \end{aligned} \tag{14}$$

Since $\tau_{i,k}$ is pending but does not execute during $[t - T_{max} - 1, t - T_{max})$ and $\tau_{i,p}$ executes during $[t-1, t)$, there must be a task $\tau_j$ that executes during $[t - T_{max} - 1, t - T_{max})$, but does not execute during $[t-1, t)$. Let $\tau_{j,\ell}$ executes during $[t - T_{max} - 1, t - T_{max})$ (see Fig. 2). By Lemma 15(a), $t > \Phi_i + T_{max}$, and hence, $t - 1 \geq \Phi_i + T_{max}$. Thus, by the definition of $t$, $\mathsf{lag}_j(t - T_{max} - 1, \mathcal{S}) \leq \mathsf{lag}_j(t-1, \mathcal{S})$ holds. Since $\tau_{j,\ell}$ executes during $[t - T_{max} - 1, t - T_{max})$, it is the earliest pending job of $\tau_j$ at $t - T_{max} - 1$. Substituting $\tau_{i,k}, t$, and $c$ in Lemma 12 by $\tau_{j,\ell}, t - T_{max} - 1$, and $h_j$, respectively, $\tau_{j,\ell+h_j}$ is pending at $t-1$. Therefore, $\tau_j$ has a pending job at $t-1$; let $\tau_{j,q}$ be the earliest pending job of $\tau_j$ at $t-1$. Thus, we have

$$\begin{aligned} y_{j,q} &\leq y_{j,\ell+h_j} \\ &= \{\text{Since } \tau_j \text{ releases periodically, } y_{j,\ell+h_j} = y_{j,\ell} + h_j T_j \text{ holds and by Def. 16}\} \\ &\quad y_{j,\ell} + T_{max}. \end{aligned} \tag{15}$$

Since $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t - T_{max} - 1$ but does not execute during $[t - T_{max} - 1, t - T_{max})$, and $\tau_{j,\ell}$ executes during $[t - T_{max} - 1, t - T_{max})$, we have two cases.

**Case 1.** $y_{j,\ell} < y_{i,k}$. Substituting $y_{j,\ell}$ by $y_{i,k}$ in (15), we have

$$y_{j,q} < y_{i,k} + T_{max}$$

$$\leq \{\text{By (14)}\}$$
$$y_{i,p}.$$

Therefore, $\tau_{j,q}$ has higher priority than $\tau_{i,p}$. Hence, $\tau_{i,p}$ cannot execute during $[t-1,t)$, while $\tau_{j,\ell}$ is not executing during $[t-1,t)$, a contradiction.

**Case 2.** $y_{j,\ell} = y_{i,k}$ and $j < i$ (as ties are broken by task index). Substituting $y_{j,\ell}$ by $y_{i,k}$ in (15), we have

$$\begin{aligned} y_{j,q} &\leq y_{i,k} + T_{max} \\ &\leq \{\text{By (14)}\} \\ &\quad y_{i,p}. \end{aligned}$$

Therefore, $\tau_{j,q}$ has higher or equal priority than $\tau_{i,p}$. Since $j < i$, $\tau_{i,p}$ cannot execute during $[t-1,t)$, while $\tau_{j,\ell}$ is not executing during $[t-1,t)$, a contradiction. ◀

By (4) and Lemma 17, we have the following LAG-monotonicity property.

▶ **Corollary 18.** *For any time instant $t \geq \Phi_{max} + T_{max}$, $\mathsf{LAG}(t - T_{max}, \mathcal{S}) \leq \mathsf{LAG}(t, \mathcal{S})$.*

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), we have $T_{max} = 6$, $\mathsf{lag}_2(4,\mathcal{S}) = -1/3 \leq 2/3 = \mathsf{lag}_2(10,\mathcal{S})$ and $\mathsf{LAG}(4,\mathcal{S}) = 1 \leq 2 = \mathsf{LAG}(10,\mathcal{S})$.

The following lemma, proved in [28], gives a relationship between lag and the deadline of the earliest pending job of a task. The lemma, originally proved for GEDF, holds for any schedule $\mathcal{S}$ provided that tasks are periodic and (B) holds.

▶ **Lemma 19** ([28]). *If $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at time $t$ in $\mathcal{S}$, then*

$$d_{i,k} \leq t - \frac{\mathsf{lag}_i(t,\mathcal{S})}{u_i} + T_i. \tag{16}$$

▶ **Corollary 20.** *If $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at $t$ in $\mathcal{S}$, then $y_{i,k} \leq t - \frac{\mathsf{lag}_i(t,\mathcal{S})}{u_i} + Y_i$.*

**Proof.** Adding $(Y_i - T_i)$ in both side of (16), we have

$$d_{i,k} + (Y_i - T_i) \leq t - \frac{\mathsf{lag}_i(t,\mathcal{S})}{u_i} + T_i + (Y_i - T_i),$$

which by (1) and the expression $d_{i,k} = r_{i,k} + T_i$ implies

$$y_{i,k} \leq t - \frac{\mathsf{lag}_i(t,\mathcal{S})}{u_i} + Y_i. \quad ◀$$

The following lemma provides a relationship between lag and tardiness. The proof of this lemma only depends on Lemma 19.

▶ **Lemma 21** ([28]). *If $\mathsf{lag}_i(t,\mathcal{S}) \leq L_i$ holds for any $t$, then the tardiness of $\tau_i$ is at most $\frac{L_i}{u_i}$.*

## 3.2 Deriving Tardiness Bounds

We now derive tardiness bounds for pseudo-harmonic periodic tasks using the properties of lag derived in Sec. 3.1. We derive our tardiness bounds by first deriving an upper bound on the lag (Lemma 27) of any task $\tau_i$, and then applying Lemma 21 on the derived upper bound. To derive an upper bound on per-task lag, we first give Lemmas 23–26. Def. 22 is adapted from [1, 9, 20].

▶ **Definition 22.** *A time instant $t$ is called* busy *if at least $\lceil U \rceil$ tasks have pending jobs at $t$, and* non-busy *otherwise. A time interval $[t, t')$ is called* busy *(resp.,* non-busy*) if each instant in the interval is busy (resp., non-busy).*

▶ **Lemma 23.** *If $\tau_i$ continuously executes during $[t, t')$ in $\mathcal{S}$, then $\mathsf{lag}_i(t', \mathcal{S}) \leq \mathsf{lag}_i(t, \mathcal{S})$.*

**Proof.** Follows from Lemma 4(a) and $u_i \leq 1$. ◀

▶ **Lemma 24.** *If $[t, t')$ is a busy interval in $\mathcal{S}$, then $\mathsf{LAG}(t', \mathcal{S}) \leq \mathsf{LAG}(t, \mathcal{S})$.*

**Proof.** By the definition of $\mathcal{I}$, $\mathsf{A}(t, t', \mathcal{I}) \leq U(t' - t)$ holds. By Def. 22, we have $\mathsf{A}(t, t', \mathcal{S}) \geq \lceil U \rceil (t' - t)$. Therefore, by (5) and $U \leq \lceil U \rceil$, $\mathsf{LAG}(t', \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S}) + \mathsf{A}(t, t', \mathcal{I}) - \mathsf{A}(t, t', \mathcal{S}) \leq \mathsf{LAG}(t, \mathcal{S}) + U(t' - t) - \lceil U \rceil (t' - t) \leq \mathsf{LAG}(t, \mathcal{S})$. ◀

▶ **Lemma 25.** *For any $t \geq \Phi_{max} + T_{max}$, if $\mathsf{LAG}(t - T_{max}, \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S})$ holds, then for each $\tau_i$, $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S})$ holds.*

**Proof.** Assume that there is a task $\tau_i$ with $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) \neq \mathsf{lag}_i(t, \mathcal{S})$. Since $t \geq \Phi_{max} + T_{max}$, by Lemma 17, $\mathsf{lag}_j(t - T_{max}, \mathcal{S}) \leq \mathsf{lag}_j(t, \mathcal{S})$ holds for any task $\tau_j$ including $\tau_i$. Therefore, $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) < \mathsf{lag}_i(t, \mathcal{S})$ holds. By (4), we have

$$\begin{aligned}\mathsf{LAG}(t - T_{max}, \mathcal{S}) &= \sum_{\tau_j \in \tau \setminus \{\tau_i\}} \mathsf{lag}_j(t - T_{max}, \mathcal{S}) + \mathsf{lag}_i(t - T_{max}, \mathcal{S}) \\ &< \{\text{Since } \mathsf{lag}_i(t - T_{max}, \mathcal{S}) < \mathsf{lag}_i(t, \mathcal{S}) \text{ and for all } j, \\ &\qquad \mathsf{lag}_j(t - T_{max}, \mathcal{S}) \leq \mathsf{lag}_j(t, \mathcal{S})\} \\ &\quad \sum_{\tau_j \in \tau \setminus \{\tau_i\}} \mathsf{lag}_j(t, \mathcal{S}) + \mathsf{lag}_i(t, \mathcal{S}) \\ &= \mathsf{LAG}(t, \mathcal{S}),\end{aligned}$$

a contradiction. ◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\mathsf{LAG}(7, \mathcal{S}) = 2 = \mathsf{LAG}(13, \mathcal{S})$ holds. By Lemma 25, we have $\mathsf{lag}_1(7, \mathcal{S}) = \mathsf{lag}_1(13, \mathcal{S}) = -1/3$, $\mathsf{lag}_2(7, \mathcal{S}) = \mathsf{lag}_2(13, \mathcal{S}) = 2/3$, and $\mathsf{lag}_3(7, \mathcal{S}) = \mathsf{lag}_3(13, \mathcal{S}) = 5/3$.

▶ **Lemma 26.** *For any $L_i > 0$, if $t$ is the first time instant such that $\mathsf{lag}_i(t, \mathcal{S}) > L_i$, then $\tau_i$ does not execute during $[t - 1, t)$.*

**Proof.** Since $L_i > 0$ and for any $t' \leq \Phi_i$, $\mathsf{lag}_i(t', \mathcal{S}) = 0$ holds, we have $t > \Phi_i$. Therefore, $\mathsf{lag}_i(t - 1, \mathcal{S}) \leq L_i$ holds. Assume that $\tau_i$ executes during $[t - 1, t)$. By Lemma 23, $\mathsf{lag}_i(t, \mathcal{S}) \leq \mathsf{lag}_i(t - 1, \mathcal{S}) \leq L_i$, a contradiction. ◀

We now show that each task $\tau_i$'s $\mathsf{lag}$ cannot exceed $(T_{max} + Y_i - Y_{min})u_i$. Informally, assume that $t$ is the first time instant where a task $\tau_i$'s $\mathsf{lag}$ exceeds $(T_{max} + Y_i - Y_{min})u_i$ in $\mathcal{S}$. If $[t - T_{max}, t)$ is a busy-interval, then by Lemma 24 ($\mathsf{LAG}$ does not increase over a busy interval) and Corollary 18 ($\mathsf{LAG}$-monotonicity), $\mathsf{LAG}$ at $t - T_{max}$ and $t$ must be the same in $\mathcal{S}$, which by Lemma 25 implies $\tau_i$'s $\mathsf{lag}$ at $t - T_{max}$ and $t$ is also same. Otherwise, if there is a non-busy instant $t_b$ in $[t - T_{max}, t)$, then by Corollary 20, $\tau_i$'s earliest pending job's priority must be higher than any job released at or after $t_b$ throughout $[t_b, t)$. Therefore, $\tau_i$ would execute continuously throughout $[t_b, t)$, violating Lemma 26. We now give the formal proof.

▶ **Lemma 27.** *For any task $\tau_i$ and any time instant $t$ in $\mathcal{S}$, $\mathsf{lag}_i(t, \mathcal{S}) \leq (T_{max} + Y_i - Y_{min})u_i$.*

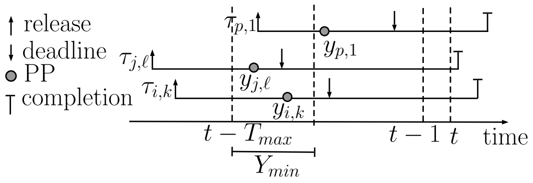


**Figure 3** Illustration of the proof of Lemma 27.

**Proof.** We use Fig. 3 to illustrate the proof. Assume that there is a time instant $t$ such that there is a task $\tau_i$ with $\mathsf{lag}_i(t, \mathcal{S}) > (T_{max} + Y_i - Y_{min})u_i$ and let $t$ be the first such time instant. Since $\mathcal{I}$ executes $\tau_i$ at the rate of $u_i$, $\mathsf{lag}_i(t', \mathcal{S}) \leq T_{max}u_i$ holds for any $t' \leq \Phi_i + T_{max}$. Therefore, $t > \Phi_i + T_{max} \geq T_{max}$ holds.

We first prove that $[t - T_{max}, t)$ is a busy interval. Since $t > T_{max}$, $[t - T_{max}, t)$ is a valid time interval. By Lemma 5, there is a pending job of $\tau_i$ at $t$ because $\mathsf{lag}_i(t, \mathcal{S}) > 0$. Let $\tau_{i,k}$ be the earliest pending job of $\tau_i$ at $t$. By Corollary 20, we have

$$\begin{aligned} y_{i,k} &\leq t - \frac{\mathsf{lag}_i(t, \mathcal{S})}{u_i} + Y_i \\ &< \{\text{Since } \mathsf{lag}_i(t, \mathcal{S}) > (T_{max} + Y_i - Y_{min})u_i\} \\ &\quad t - \frac{(T_{max} + Y_i - Y_{min})u_i}{u_i} + Y_i \\ &= t - T_{max} + Y_{min}. \end{aligned} \tag{17}$$

By (1), we have

$$\begin{aligned} r_{i,k} &= y_{i,k} - Y_i \\ &< \{\text{By (17)}\} \\ &\quad t - T_{max} + Y_{min} - Y_i \\ &\leq \{\text{Since } Y_{min} \leq Y_i\} \\ &\quad t - T_{max}. \end{aligned} \tag{18}$$

Thus, $\tau_{i,k}$ is pending throughout $[t - T_{max}, t)$. Since $t$ is the first time instant with $\mathsf{lag}_i(t, \mathcal{S}) > (T_{max} + Y_i - Y_{min})u_i$, by Lemma 26, $\tau_{i,k}$ does not execute during $[t-1, t)$. Thus, there are at least $m$ tasks with higher priority jobs than $\tau_{i,k}$ at $t-1$. Let $\tau^h$ be the set of tasks having higher priority jobs than $\tau_{i,k}$ at $t-1$. Then, $|\tau^h| \geq m$ holds. By the definition of $\tau^h$, for any task $\tau_j \in \tau^h$, $y_{j,\ell} \leq y_{i,k}$ holds where $\tau_{j,\ell}$ is the earliest pending job of $\tau_j$ at $t-1$ (see Fig. 3). By a calculation similar to that yielding (18), $r_{j,\ell} < t - T_{max}$ holds, which implies $\tau_{j,\ell}$ is pending throughout $[t - T_{max}, t)$. Thus, by (17) we have the following property.

**Property P:** *Each task in $\tau^h \cup \{\tau_i\}$ has pending jobs with PPs less than $T_{max} + Y_i - Y_{min}$ throughout $[t - T_{max}, t)$.*

By Property P, $[t - T_{max}, t)$ is a busy interval. By Lemma 24, we therefore have

$$\mathsf{LAG}(t, \mathcal{S}) \leq \mathsf{LAG}(t - T_{max}, \mathcal{S}). \tag{19}$$

We now consider two cases.

**Case 1.** $t \geq \Phi_{max} + T_{max}$. By Corollary 18, we have

$$\mathsf{LAG}(t, \mathcal{S}) \geq \mathsf{LAG}(t - T_{max}, \mathcal{S}). \tag{20}$$

By (19) and (20), we have

$$\mathsf{LAG}(t, \mathcal{S}) = \mathsf{LAG}(t - T_{max}, \mathcal{S}). \tag{21}$$

Since $t \geq \Phi_{max} + T_{max}$ and (21) holds, by Lemma 25, $\mathsf{lag}_i(t, \mathcal{S}) = \mathsf{lag}_i(t - T_{max}, \mathcal{S})$ holds. Therefore, $t$ cannot be the first time instant with $\mathsf{lag}_i(t, \mathcal{S}) > (T_{max} + Y_i - Y_{min})u_i$.

**Case 2.** $t < \Phi_{max} + T_{max}$. Let $\tau^s$ be the set of tasks such that for each $\tau_p \in \tau^s$, $t - T_{max} < \Phi_p \leq \Phi_{max}$ holds. Since each task $\tau_p \in \tau^s$ releases its first job after $t - T_{max}$, $r_{p,1} > t - T_{max}$ and $\mathsf{lag}_p(t - T_{max}, \mathcal{S}) = 0$ hold (see Fig. 3). Thus, by (1) and $Y_p \geq Y_{min}$, we have

$$(\forall \tau_p \in \tau^s : y_{p,1} > t - T_{max} + Y_{min}). \tag{22}$$

By Property P and (22), no task $\tau_p \in \tau^s$ executes during $[t - T_{max}, t)$. Therefore, we have

$$(\forall \tau_p \in \tau^s : \mathsf{lag}_p(t, \mathcal{S}) \geq 0 = \mathsf{lag}_p(t - T_{max}, \mathcal{S})). \tag{23}$$

By the definition of $\tau^s$, for any task $\tau_q \in \tau \setminus \tau^s$, $t - T_{max} \geq \Phi_q$ holds, which implies $t \geq \Phi_q + T_{max}$. Therefore, by Lemma 17, we have

$$(\forall \tau_q \in \tau \setminus \tau^s : \mathsf{lag}_q(t, \mathcal{S}) \geq \mathsf{lag}_q(t - T_{max}, \mathcal{S})). \tag{24}$$

Since $t$ is the first time instant with $\mathsf{lag}_i(t, \mathcal{S}) > (T_{max} + Y_i - Y_{min})u_i > 0$, $\mathsf{lag}_i(t', \mathcal{S}) \leq (T_{max} + Y_i - Y_{min})u_i$ holds for any $t' < t$. Thus, we have

$$\mathsf{lag}_i(t, \mathcal{S}) > \mathsf{lag}_i(t - T_{max}, \mathcal{S}). \tag{25}$$

By (4), we have

$$\begin{aligned}
\mathsf{LAG}(t, \mathcal{S}) &= \sum_{\tau_j \in \tau} \mathsf{lag}_j(t, \mathcal{S}) \\
&= \sum_{\tau_j \in \tau^s} \mathsf{lag}_j(t, \mathcal{S}) + \sum_{\tau_j \in \tau \setminus (\tau^s \cup \{\tau_i\})} \mathsf{lag}_j(t, \mathcal{S}) + \mathsf{lag}_i(t, \mathcal{S}) \\
&> \{\text{By (23), (24), and (25)}\} \\
&\quad \sum_{\tau_j \in \tau^s} \mathsf{lag}_j(t - T_{max}, \mathcal{S}) + \sum_{\tau_j \in \tau \setminus (\tau^s \cup \{\tau_i\})} \mathsf{lag}_j(t - T_{max}, \mathcal{S}) + \mathsf{lag}_i(t - T_{max}, \mathcal{S}) \\
&= \mathsf{LAG}(t - T_{max}, \mathcal{S}),
\end{aligned}$$

a contradiction to (19). ◀

We now give our tardiness bound in the following Theorem.

▶ **Theorem 28.** *The tardiness of task $\tau_i$ is at most $T_{max} + Y_i - Y_{min}$ in $\mathcal{S}$.*

**Proof.** The theorem follows from Lemmas 21 and 27. ◀

By Theorem 28, we have following tardiness bounds under GEDF and FIFO.

▶ **Theorem 29.** *The tardiness of a task $\tau_i$ in a GEDF and FIFO schedule is at most $T_{max} + T_i - T_{min}$ and $T_{max}$, respectively.*

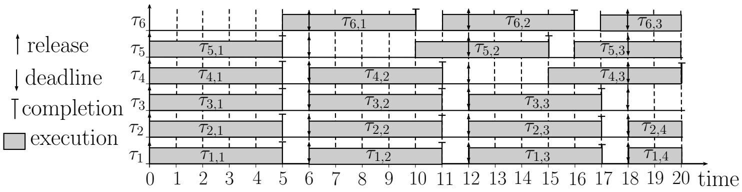


**Figure 4** Schedule corresponding to Ex. 31.

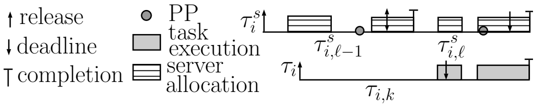


**Figure 5** Scheduling sporadic tasks by GEL-scheduled periodic servers.

**Removing Assumption (B).** Prior work has shown that removing Assumption (B) does not invalidate GEDF tardiness bounds because its removal cannot cause work to shift later [28]. It can be similarly removed for any GEL scheduler.

▶ **Theorem 30.** *Let $\tau$ be a periodic task set, $\mathcal{S}$ be a GEL schedule of $\tau$ satisfying (B), and $\mathcal{S}'$ be a GEL schedule with the same PP for each job of $\tau$ without satisfying (B). Then no job in $\mathcal{S}'$ finishes later than in $\mathcal{S}$.*

**Tightness.** The following example shows the tightness of the tardiness bound in Theorem 28.

▶ **Example 31.** Consider a task system $\tau$ with $m+1$ tasks where $\tau_i = (0, m, m+1)$. For any job-level fixed-priority scheduler, the maximum tardiness among all tasks is $m-1 = T_{max}-2$. For both FIFO and GEDF, the tardiness bound of $\tau$ by Theorem 28 is $T_{max}$. A GEDF/FIFO schedule corresponding to $m = 5$ is shown in Fig. 4. Jobs $\tau_{6,1}, \tau_{5,2}$, and $\tau_{4,3}$ have tardiness of 4.0, 3.0, and 2.0 time units, respectively. Similarly, $\tau_{3,4}$ has tardiness of 1.0 time unit (not shown in Fig. 4). $\tau_1$ and $\tau_2$ have no tardy job. The schedule repeats after 30.0 time units.◀

**Sporadic tasks.** We can enable similar tardiness bounds for sporadic tasks using GEL-scheduled periodic servers. For each task $\tau_i$, we create a server task $\tau_i^s = (0, C_i, T_i)$. We schedule the server tasks by a GEL scheduler where each server job of $\tau_i^s$ receives an allocation of exactly $C_i$ time units. We schedule job $\tau_{i,k}$ on server job $\tau_{i,\ell}^s$ where $d_{i,k} \in (r_{i,\ell}^s, d_{i,\ell}^s]$ (see Fig. 5). Since both $\tau_i$ and $\tau_i^s$ have the same period, no other job of $\tau_i$ is scheduled on $\tau_{i,\ell}^s$. Since $\tau_{i,\ell}^s$ receives allocation of $C_i$ time units, $\tau_{i,k}$ finishes execution at or before $\tau_{i,\ell}^s$ completes. Since $d_{i,\ell}^s - d_{i,k} \leq T_i$, we have the following theorem.

▶ **Theorem 32.** *A pseudo-harmonic sporadic task system $\tau$ can be scheduled using periodic servers scheduled by a GEL scheduler such that each task $\tau_i$'s tardiness is at most $T_{max} + Y_i - Y_{min} + T_i$.*

**Discussion.** While the tardiness bound given in Theorem 28 is tight in general, the tardiness bound is not tight for task systems with a smaller total utilization than $m$. For instance, a HRT-schedulable task system also has the tardiness bound specified in Theorem 28. Although the tardiness bounds in [9, 11, 27] can have smaller bounds when the total utilization is less than $m$ compared to systems with full utilization, they also have similar issues, e.g., positive tardiness bounds for HRT-schedulable task systems.

Although the tardiness bound given in Theorem 28 is $T_{max}$ under FIFO, the tardiness bound under GEDF can be larger than $T_{max}$. The tardiness of a task can actually exceed $T_{max}$ under GEDF as illustrated in the following example.

▶ **Example 33.** Consider a task system with five tasks $\tau_1 = (1, 4, 5)$, $\tau_2 = (3, 3, 4)$, $\tau_3 = (9, 19, 25)$, $\tau_4 = (20, 99, 100)$, $\tau_5 = (75, 70, 100)$ scheduled on four processors by GEDF. It can be shown that the tardiness of the job $\tau_{4,48}$ is 104 time units, which is $T_{max} + 4$.

## 4 Exact Tardiness Bounds

Having derived a tardiness bound for pseudo-harmonic periodic tasks that does not depend on the processor or task count in Sec. 3, we now show how to derive an exact tardiness bound in pseudo-polynomial time. We do so by deriving an upper bound on the length of the prefix of a schedule during which tasks may experience increasing tardiness (afterwards, they do not). We show, in Lemma 39, that if there is a time instant $t \geq \Phi_{max}$ when LAG has the same values at $t$ and $t - T_{max}$, then for any $t' \geq t$, the LAG values at $t'$ and $t' - T_{max}$ are also equal. Intuitively, this implies that the schedule in the interval $[t - T_{max}, t)$ repeats after $t$. Moreover, since the lag of each task is bounded (Lemma 27), we can derive an upper bound on LAG (Lemma 38). Therefore, since LAG does not decrease over any interval of length $T_{max}$ starting after $\Phi_{max}$ (LAG-monotonicity), there must be a finite interval $[\Phi_{max}, t')$ such that LAG strictly increases over any interval of length $T_{max}$ in $[\Phi_{max}, t')$. We derive an upper bound on such an interval in Lemma 41. Intuitively, for each task, a job with the maximum tardiness of the task must complete at or before the schedule starts to cycle. We first consider task systems satisfying (B). We define a *max-tardiness-increasing interval* as follows.

▶ **Definition 34.** *Given a periodic task system $\tau$, a* max-tardiness-increasing interval *in a schedule $\mathcal{S}$ is a finite interval of time $[0, t]$ such that for each task $\tau_i \in \tau$, if the maximum tardiness of $\tau_i$'s jobs that complete execution at or before $t$ is $x_i$ in $\mathcal{S}$, then the tardiness of $\tau_i$ is $x_i$ in $\mathcal{S}$.*

We now derive an upper bound on the max-tardiness-increasing interval of a pseudo-harmonic periodic task system $\tau$ satisfying (B).

▶ **Definition 35.** *Let $F$ be the sum of the $n - 1$ largest values of $C_i(1 - u_i)$; i.e., $F = \sum_{n-1 \; largest} C_i(1 - u_i)$. Let $G$ be the sum of the $\lceil U \rceil - 1$ largest values of $(T_{max} + Y_i - Y_{min})u_i$; i.e., $G = \sum_{\lceil U \rceil - 1 \; largest} (T_{max} + Y_i - Y_{min})u_i$. Let $E = \lceil F + G + 1 \rceil$.*

The following lemma gives a trivial lower bound on the lag of a task at any time $t$ in $\mathcal{S}$. A task's lag is minimum when its active job finishes execution as early as possible in $\mathcal{S}$, i.e., $C_i$ time units after its release.

▶ **Lemma 36.** *For any task $\tau_i$ and time instant $t$, $\mathsf{lag}_i(t, \mathcal{S}) \geq -C_i(1 - u_i)$.*

**Proof.** If $t \leq \Phi_i$, then $\mathsf{lag}_i(t, \mathcal{S}) = 0$, so assume $t > \Phi_i$. Let $\tau_{i,k}$ be the active job of $\tau_i$ at $t$ and $e_i$ be the cost of the completed portion of $\tau_{i,k}$ at or before $t$ in $\mathcal{S}$. Therefore, $\mathsf{A}_i(0, t, \mathcal{S}) = \sum_{j=1}^{k-1} C_i + e_i$. By the definition of $\mathcal{I}$, all jobs of $\tau_i$ prior to $\tau_{i,k}$ complete execution by $t$ in $\mathcal{I}$. Since jobs can only execute after their release, by the time $\tau_{i,k}$ executes for $e_i$ units in $\mathcal{S}$, $\tau_{i,k}$ must execute at least $e_i u_i$ units of $\tau_{i,k}$ in $\mathcal{I}$. Therefore, $\mathsf{A}_i(0, t, \mathcal{I}) \geq \sum_{j=1}^{k-1} C_i + e_i u_i$. Substituting $\mathsf{A}_i(0, t, \mathcal{I})$ and $\mathsf{A}_i(0, t, \mathcal{S})$ in (2), we have $\mathsf{lag}_i(t, \mathcal{S}) \geq \sum_{j=1}^{k-1} C_i + e_i u_i - \sum_{j=1}^{k-1} C_i - e_i = -e_i(1 - u_i)$. Since $e_i \leq C_i$, we have $\mathsf{lag}_i(t, \mathcal{S}) \geq -C_i(1 - u_i)$. ◀

We now give a lower bound on $\mathsf{LAG}$ at $\Phi_{max}$ in $\mathcal{S}$. By the definition of $\Phi_{max}$, there is at least one task with $\mathsf{lag}$ that equals 0 at $\Phi_{max}$.

▶ **Lemma 37.** $\mathsf{LAG}(\Phi_{max}, \mathcal{S}) \geq -F$.

**Proof.** Let $\tau'$ be the set of tasks such that for any $\tau_i \in \tau'$, $\Phi_i = \Phi_{max}$ holds. Therefore, $\mathsf{lag}_i(\Phi_{max}, \mathcal{S}) = 0$ holds for any $\tau_i \in \tau'$. Thus, $\sum_{\tau_i \in \tau'} \mathsf{lag}_i(\Phi_{max}, \mathcal{S}) = 0$. Hence, by (4), we have $\mathsf{LAG}(\Phi_{max}, \mathcal{S}) = \sum_{\tau_i \in \tau} \mathsf{lag}_i(\Phi_{max}, \mathcal{S}) = \sum_{\tau_i \in \tau \setminus \tau'} \mathsf{lag}_i(\Phi_{max}, \mathcal{S})$, which by Lemma 36 implies, $\mathsf{LAG}(\Phi_{max}, \mathcal{S}) \geq \sum_{\tau_i \in \tau \setminus \tau'} -C_i(1 - u_i)$. By the definition of $\Phi_{max}$, $|\tau'| \geq 1$ holds. Therefore, by Def. 35, we have $\mathsf{LAG}(\Phi_{max}, \mathcal{S}) \geq -\sum_{n-1 \text{ largest}} C_i(1 - u_i) = -F$. ◀

We now derive an upper bound on $\mathsf{LAG}$ at any time $t$ in $\mathcal{S}$ by determining an upper bound on $\mathsf{LAG}$ at the latest non-busy time instant at or before $t$.

▶ **Lemma 38.** *For any $t$, $\mathsf{LAG}(t, \mathcal{S}) \leq G$.*

**Proof.** Let $t_b$ be the latest non-busy time instant at or before $t$, otherwise let $t_b = 0$. We first derive an upper bound on $\mathsf{LAG}(t_b, \mathcal{S})$. If $t_b = 0$, then $\mathsf{LAG}(t_b, \mathcal{S}) = 0$. Otherwise, let $\tau' \subseteq \tau$ be the tasks with pending jobs at $t_b$. By (4),

$$\begin{aligned}
\mathsf{LAG}(t_b, \mathcal{S}) = &\sum_{\tau_i \in \tau'} \mathsf{lag}_i(t_b, \mathcal{S}) + \sum_{\tau_i \notin \tau'} \mathsf{lag}_i(t_b, \mathcal{S}) \\
&\leq \{\text{By the contrapositive of Lemma 5, } (\forall \tau_i \notin \tau' : \mathsf{lag}_i(t_b, \mathcal{S}) \leq 0) \text{ holds}\} \\
&\quad \sum_{\tau_i \in \tau'} \mathsf{lag}_i(t_b, \mathcal{S}) \\
&\leq \{\text{By Lemma 27}\} \\
&\quad \sum_{\tau_i \in \tau'} (T_{max} + Y_i - Y_{min}) u_i \\
&\leq \{\text{By Def. 22, } |\tau'| < \lceil U \rceil\} \\
&\quad \sum_{\lceil U \rceil - 1 \text{ largest}} (T_{max} + Y_i - Y_{min}) u_i \\
&= \{\text{By Def. 35}\} \\
&\quad G.
\end{aligned}$$

By Lemma 24, $\mathsf{LAG}(t, \mathcal{S}) \leq \mathsf{LAG}(t_b, \mathcal{S}) \leq G$ holds. ◀

Lemmas 37 and 38 imply that $\mathsf{LAG}$ cannot increase more than $F + G$ units over any interval $[\Phi_{max}, t)$. We use this fact later in Lemma 41. We now show that once $\mathsf{LAG}(t, \mathcal{S}) = \mathsf{LAG}(t - T_{max}, \mathcal{S})$ holds for some $t$, the equality also holds for all time instances after $t$. Informally, by Lemma 25, if $\mathsf{LAG}(t, \mathcal{S}) = \mathsf{LAG}(t - T_{max}, \mathcal{S})$ holds, then for any task $\tau_i$, $\mathsf{lag}_i(t, \mathcal{S}) = \mathsf{lag}_i(t - T_{max}, \mathcal{S})$ also holds. This implies that the scheduling decisions at $t$ are the same as at $t - T_{max}$. Therefore, the schedule in $[t - T_{max}, t)$ repeats in $[t, t + T_{max})$.

▶ **Lemma 39.** *If there is a time instant $t' \geq \Phi_{max} + T_{max}$ such that $\mathsf{LAG}(t' - T_{max}, \mathcal{S}) = \mathsf{LAG}(t', \mathcal{S})$ holds, then for any $t \geq t'$, $\mathsf{LAG}(t - T_{max}, \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S})$ holds.*

**Proof.** Assume for a contradiction that there exists a $t \geq t'$ such that $\mathsf{LAG}(t - T_{max}, \mathcal{S}) \neq \mathsf{LAG}(t, \mathcal{S})$ and let $t$ be the first such time instant. By the definition of $t$ and $t'$, $t > t' \geq \Phi_{max} + T_{max}$ and $t - 1 \geq \Phi_{max} + T_{max}$ hold. Therefore, $\mathsf{LAG}(t - T_{max} - 1, \mathcal{S}) = \mathsf{LAG}(t - 1, \mathcal{S})$ holds. Thus, by Lemma 25, we have

$$(\forall \tau_i : \mathsf{lag}_i(t - T_{max} - 1, \mathcal{S}) = \mathsf{lag}_i(t - 1, \mathcal{S})). \tag{26}$$

Since $T_i$ divides $T_{max}$, by (26) and Lemma 14(a) (with $t$ and $c$ replaced by $t - T_{max} - 1$ and $h_i$, respectively), we have the following property.

**Property Q:** *Any task with no pending job at $t - T_{max} - 1$ has no pending job at $t - 1$.* Let $\tau' \subseteq \tau$ be the set of tasks with pending jobs at $t - T_{max} - 1$. Let $\tau_{i,k}$ be the earliest pending job of $\tau_i \in \tau'$ at $t - T_{max} - 1$. By Def. 16, (26) and Lemma 14(b) (with $t$ and $c$ replaced by $t - T_{max} - 1$ and $h_i$, respectively), $\tau_{i,k+h_i}$ is the earliest pending job of $\tau_i$ at $t - 1$. Since $\tau_i$ releases jobs periodically, we have $y_{i,k+h_i} = y_{i,k} + h_i T_i = y_{i,k} + T_{max}$. Thus, the tasks in $\tau'$ have the same priority ordering at both $t - T_{max} - 1$ and $t - 1$, which along with Property Q implies that the same set of tasks execute during both $[t - T_{max} - 1)$ and $[t - 1, t)$. Hence, $\mathsf{A}(t - T_{max} - 1, t - T_{max}, \mathcal{S}) = \mathsf{A}(t - 1, t, \mathcal{S})$. Since $t - T_{max} - 1 \geq \Phi_{max}$, we have $\mathsf{A}(t - T_{max} - 1, t - T_{max}, \mathcal{I}) = \mathsf{A}(t - 1, t, \mathcal{I})$. Thus, by (5) we have

$$\begin{aligned}
\mathsf{LAG}(t, \mathcal{S}) &= \mathsf{LAG}(t-1, \mathcal{S}) + \mathsf{A}(t-1, t, \mathcal{I}) - \mathsf{A}(t-1, t, \mathcal{S}) \\
&= \{\text{Since } \mathsf{LAG}(t-1, \mathcal{S}) = \mathsf{LAG}(t + T_{max} - 1)\} \\
&\quad \mathsf{LAG}(t - T_{max} - 1, \mathcal{S}) + \mathsf{A}(t-1, t, \mathcal{I}) - \mathsf{A}(t-1, t, \mathcal{S}) \\
&= \{\text{Since } \mathsf{A}(t-1, t, \mathcal{S}) = \mathsf{A}(t - T_{max} - 1, t - T_{max}, \mathcal{S}) \text{ and} \\
&\qquad\quad \mathsf{A}(t-1, t, \mathcal{I}) = \mathsf{A}(t - T_{max} - 1, t - T_{max}, \mathcal{I})\} \\
&= \mathsf{LAG}(t - T_{max} - 1, \mathcal{S}) + \mathsf{A}(t - T_{max} - 1, t - T_{max}, \mathcal{I}) \\
&\qquad - \mathsf{A}(t - T_{max} - 1, t - T_{max}, \mathcal{S}) \\
&= \{\text{By (5)}\} \\
&\quad \mathsf{LAG}(t - T_{max}, \mathcal{S}),
\end{aligned}$$

a contradiction. ◀

▶ **Corollary 40.** *If there is a time instant $t' \geq \Phi_{max} + T_{max}$ such that $\mathsf{LAG}(t' - T_{max}, \mathcal{S}) = \mathsf{LAG}(t', \mathcal{S})$ holds, then $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S})$ holds for any $t \geq t'$ and $\tau_i \in \tau$.*

**Proof.** Follows from Lemmas 39 and 25. ◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\mathsf{LAG}(6, \mathcal{S}) = \mathsf{LAG}(12, \mathcal{S})$ holds. Therefore, for all task $\tau_i$ and $t \geq 12$, $\mathsf{LAG}(t - T_{max}, \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S}) = 2$ and $\mathsf{lag}_i(t - T_{max}, \mathcal{S}) = \mathsf{lag}_i(t, \mathcal{S})$ hold. We now show that there is a time instant $t$ after $\Phi_{max} + T_{max}$ and at or before $\Phi_{max} + ET_{max}$ where LAG has the same value at $t$ and $t - T_{max}$. Therefore, the schedule starts to cycle at or before $\Phi_{max} + ET_{max}$. Intuitively, LAG must increase by at least 1.0 execution unit, if not equal, over each interval $[\Phi_{max} + iT_{max}, \Phi_{max} + (i+1)T_{max})$ where $0 \leq i < E$. Therefore, since $E = \lceil F + G + 1 \rceil$, LAG at $\Phi_{max} + ET_{max}$ must be more than $G$, contradicting Lemma 38.

▶ **Lemma 41.** *There is a time instant $t \in [\Phi_{max} + T_{max}, \Phi_{max} + ET_{max}]$ such that $\mathsf{LAG}(t - T_{max}, \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S})$ holds.*

**Proof.** Assume $\mathsf{LAG}(t - T_{max}, \mathcal{S}) \neq \mathsf{LAG}(t, \mathcal{S})$ holds for all $t \in [\Phi_{max} + T_{max}, \Phi_{max} + ET_{max}]$. Let $t$ be any arbitrary time instant in $[\Phi_{max} + T_{max}, \Phi_{max} + ET_{max}]$. Since $t \geq \Phi_{max} + T_{max}$, by Corollary 18, we have $\mathsf{LAG}(t - T_{max}, \mathcal{S}) \leq \mathsf{LAG}(t, \mathcal{S})$. Thus, $\mathsf{LAG}(t - T_{max}, \mathcal{S}) < \mathsf{LAG}(t, \mathcal{S})$ holds. Since tasks release jobs periodically and $t - T_{max} \geq \Phi_{max}$ holds, we have

$$\mathsf{A}(t - T_{max}, t, \mathcal{I}) = UT_{max}. \tag{27}$$

Since $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ and $h_i = T_{max}/T_i$, we have $U = \frac{\sum_{i=1}^{n} h_i C_i}{T_{max}}$. Therefore, $UT_{max} = \sum_{i=1}^{n} h_i C_i$. Since both $h_i$ and $C_i$ are integers, $UT_{max}$ is also an integer. By (5), we have

$$\mathsf{A}(t - T_{max}, t, \mathcal{S}) = \mathsf{A}(t - T_{max}, t, \mathcal{I}) + \mathsf{LAG}(t - T_{max}, \mathcal{S}) - \mathsf{LAG}(t, \mathcal{S})$$

$$\begin{aligned}
&< \{\text{Since } \mathsf{LAG}(t-T_{max},\mathcal{S}) < \mathsf{LAG}(t,\mathcal{S})\}\\
&\quad \mathsf{A}(t-T_{max},t,\mathcal{I})\\
&= \{\text{Since } \mathsf{A}(t-T_{max},t,\mathcal{I}) = UT_{max}\}\\
&\quad UT_{max} \qquad (28)
\end{aligned}$$

Since $UT_{max}$ and $\mathsf{A}(t-T_{max},t,\mathcal{S})$ are integers, by (28) we have

$$\mathsf{A}(t-T_{max},t,\mathcal{S}) \le UT_{max} - 1. \qquad (29)$$

Now, by (5), we have

$$\begin{aligned}
\mathsf{LAG}(\Phi_{max}+ET_{max},\mathcal{S}) &= \mathsf{LAG}(\Phi_{max},\mathcal{S}) + \mathsf{A}(\Phi_{max},\Phi_{max}+ET_{max},\mathcal{I})\\
&\qquad - \mathsf{A}(\Phi_{max},\Phi_{max}+ET_{max},\mathcal{S})\\
&= \{\text{Since } [\Phi_{max},\Phi_{max}+ET_{max}) =\\
&\qquad \cup_{i=0}^{E-1} [\Phi_{max}+iT_{max},\Phi_{max}+(i+1)T_{max}).\}\\
&\quad \mathsf{LAG}(\Phi_{max},\mathcal{S}) + \sum_{i=0}^{E-1}(\mathsf{A}(\Phi_{max}+iT_{max},\Phi_{max}+(i+1)T_{max},\mathcal{I})\\
&\qquad - \mathsf{A}(\Phi_{max}+iT_{max},\Phi_{max}+(i+1)T_{max},\mathcal{S}))\\
&\ge \{\text{Substituting } t = \Phi_{max}+(i+1)T_{max} \text{ in (27) and (29)}\}\\
&\quad \mathsf{LAG}(\Phi_{max},\mathcal{S}) + \sum_{i=0}^{E-1}(UT_{max} - UT_{max} + 1)\\
&= \mathsf{LAG}(\Phi_{max},\mathcal{S}) + \sum_{i=0}^{E-1} 1\\
&\ge \{\text{By Lemma 37 and Def. 35}\}\\
&\quad -F + F + G + 1\\
&> G,
\end{aligned}$$

a contraction to Lemma 38. ◀

We now show that a job with the maximum tardiness must complete execution at or before $\Phi_{max} + ET_{max}$ by Lemma 42 and Theorem 43.

▶ **Lemma 42.** *If there is a time instant $t' \ge \Phi_{max} + T_{max}$ such that $\mathsf{LAG}(t'-T_{max},\mathcal{S}) = \mathsf{LAG}(t',\mathcal{S})$ holds and $x_i$ is the maximum tardiness of any of task $\tau_i$'s jobs that complete execution at or before $t'$ in $\mathcal{S}$, then the tardiness of $\tau_i$ is $x_i$ in $\mathcal{S}$.*

**Proof.** Assume that the tardiness of $\tau_i$ is more than $x_i$ and let $\tau_{i,k}$ be the first job with tardiness exceeding $x_i$. Let $t$ be the time instant when $\tau_{i,k}$ finishes execution. Then, $t > t'$ holds. Since $\mathsf{LAG}(t'-T_{max},\mathcal{S}) = \mathsf{LAG}(t',\mathcal{S})$ and $t-1 \ge t'$ hold, by Corollary 40, we have $\mathsf{lag}_i(t-T_{max}-1,\mathcal{S}) = \mathsf{lag}_i(t-1,\mathcal{S})$. Since $h_i = T_{max}/T_i$ and $\tau_{i,k}$ is pending at $t-1$, substituting $t$ and $c$ in Lemma 14(b) by $t-1$ and $-h_i$, respectively, $\tau_{i,k-h_i}$ is pending at $t-1-T_{max}$. Therefore, $\tau_{i,k-h_i}$ finishes execution at or after $t-T_{max}$. Hence, we have

$$\begin{aligned}
f_{i,k-h_i} - d_{i,k-h_i} &\ge t - T_{max} - d_{i,k-h_i}\\
&= \{\text{Since } \tau_i \text{ releases periodically, } d_{i,k-h_i} = d_{i,k} - h_iT_i\}\\
&\quad t - T_{max} - d_{i,k} + h_iT_i\\
&= \{\text{By the definition of } t \text{ and Def. 16}\}\\
&\quad f_{i,k} - d_{i,k}.
\end{aligned}$$

Therefore, $\max\{0, f_{i,k-h_i} - d_{i,k-h_i}\} \geq \max\{0, f_{i,k} - d_{i,k}\}$ holds and $\tau_{i,k}$'s tardiness cannot exceed $\tau_{i,k-h_i}$'s tardiness. ◀

For the task system in Ex. 3 and its GEDF schedule in Fig. 1(a), $\mathsf{LAG}(t - T_{max}, \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S})$ holds for the first time at time 12. Job $\tau_{3,1}$ has the maximum tardiness in $\mathcal{S}$.

▶ **Theorem 43.** *If the maximum tardiness of a task $\tau_i$'s jobs that completes at or before $\Phi_{max} + ET_{max}$ is $x_i$ in $\mathcal{S}$, then the tardiness of $\tau_i$ is $x_i$ in $\mathcal{S}$.*

**Proof.** By Lemma 41, there is a time instant $t \in [\Phi_{max} + T_{max}, \Phi_{max} + ET_{max}]$ such that $\mathsf{LAG}(t - T_{max}, \mathcal{S}) = \mathsf{LAG}(t, \mathcal{S})$ holds. Let $z_i$ be the maximum tardiness of $\tau_i$'s jobs that complete execution at or before $t$ in $\mathcal{S}$. By Lemma 42, the tardiness of $\tau_i$ is $z_i$. Since $t \in [\Phi_{max} + T_{max}, \Phi_{max} + ET_{max}]$, by the definition of $x_i$, $z_i \leq x_i$ holds. Since the tardiness of $\tau_i$ in $\mathcal{S}$ is $z_i$, $z_i \geq x_i$ holds. Therefore, $x_i = z_i$. ◀

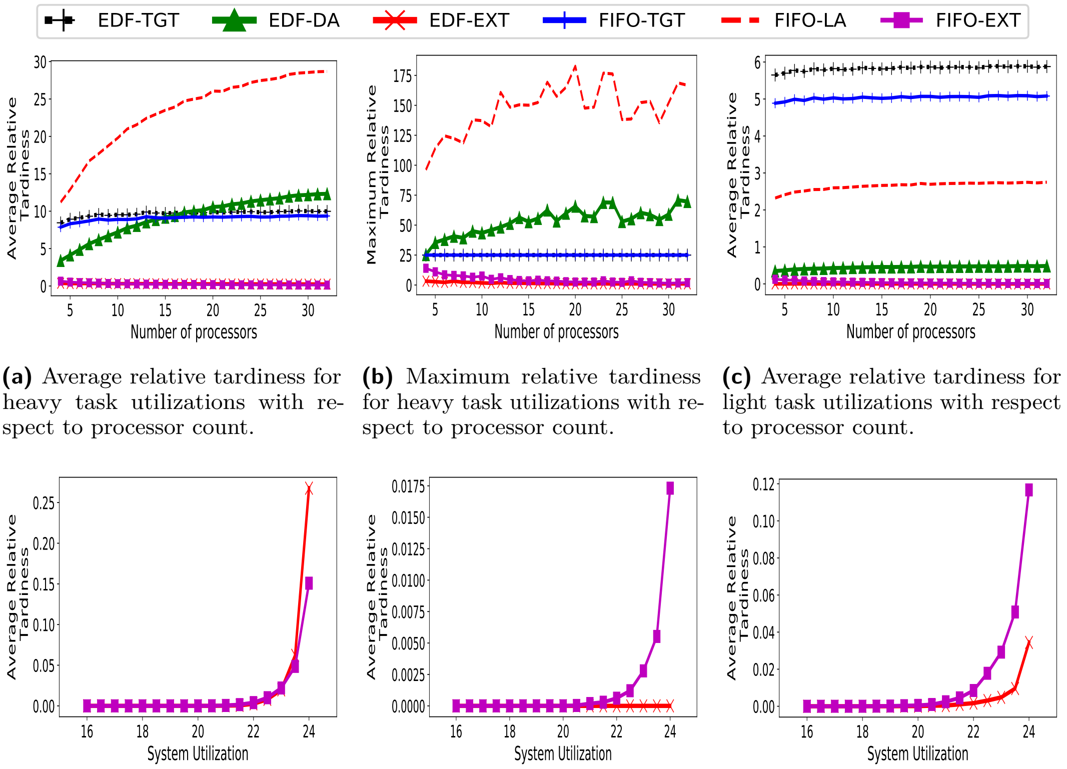
By Theorems 30 and 43, if the maximum tardiness of $\tau_i$'s jobs that complete at or before $\Phi_{max} + ET_{max}$ is $x_i$ in a GEL schedule $\mathcal{S}$ satisfying (B), then $\tau_i$'s tardiness is at most $x_i$ in a GEL schedule $\mathcal{S}'$ not satisfying (B).

**Deriving tardiness.** By Theorem 43, we can determine an exact tardiness bound of each task by simulating a schedule up to time $\Phi_{max} + ET_{max}$. By Def. 35, we have $F = \sum_{n-1 \text{ largest}} C_i(1 - u_i) \leq \sum_{i=1}^{n} C_i = \sum_{i=1}^{n} T_i u_i \leq T_{max} \sum_{i=1}^{n} u_i \leq mT_{max}$. By Def. 35, $G = \sum_{\lceil U \rceil - 1 \text{ largest}} (T_{max} + Y_i - Y_{min}) u_i \leq \sum_{m-1 \text{ largest}} (T_{max} + Y_{max}) = (m-1)(T_{max} + Y_{max})$ holds. Therefore, we have $E = \lceil F + G + 1 \rceil \leq \lceil mT_{max} + (m-1)(T_{max} + Y_{max}) + 1 \rceil$. Since scheduling decisions at each time instant are determined in polynomial time, simulating a schedule up to time $\Phi_{max} + ET_{max}$ takes pseudo-polynomial time. By Lemma 42, we can terminate the simulation early at time $t \geq \Phi_{max} + T_{max}$ by checking whether $\mathsf{LAG}(t, \mathcal{S}) = \mathsf{LAG}(t - T_{max}, \mathcal{S})$ holds. This would require storing the last $T_{max}$ values of LAG at any time. We can also store one value of LAG at any time, e.g., the last time instant that is multiple of $T_{max}$, and check for LAG-equality $T_{max}$ time after the last-stored instance. This would require simulating for at most $T_{max}$ time units more than that required when $T_{max}$ values of LAG are stored. We note that this method can be adapted for non-pseudo-harmonic systems with $T_{max}$ and $G$ replaced with $H$ and a corresponding upper bound on LAG, respectively.

## 5 Experiments

We now present the results of simulation experiments we conducted to evaluate our tardiness bounds and the effectiveness of our approach to derive exact tardiness bounds.

We generated task systems randomly for systems with 4 to 32 processors. We chose *light*, *medium*, *heavy*, or *wide* task utilizations, for which task utilizations were uniformly distributed in $[0.01, 0.3]$, $[0.3, 0.7]$, $[0.7, 1]$, and $[0.01, 1]$, respectively. We chose task periods uniformly from $\{4, 5, 10, 20, 25, 50, 100\}$ms. In case there was no task with a period of 100ms, we randomly chose a task and scaled its parameters to set its period to 100ms. We rounded down each execution cost to its nearest integer and disregarded any task if its execution cost became zero. We chose the offset of each task randomly between 0 and its period. For each utilization cap $m$ and utilization distribution, we generated 1,000 task systems by adding tasks until five attempts to add a next task without exceeding the utilization cap failed. To compare tardiness bounds with respect to system utilization, we considered a 24-processor platform and generated 1,000 task systems for each utilization cap within $[16, 24]$ with a step size of 0.5.



**(a)** Average relative tardiness for heavy task utilizations with respect to processor count.

**(b)** Maximum relative tardiness for heavy task utilizations with respect to processor count.

**(c)** Average relative tardiness for light task utilizations with respect to processor count.

**(d)** Average relative tardiness for heavy task utilizations with respect to system utilization.

**(e)** Average relative tardiness for light task utilizations with respect to system utilization.

**(f)** Average relative tardiness for wide task utilizations with respect to system utilization.

**Figure 6** Average and maximum relative tardiness with respect to the number of processors and system utilizations.

We used *relative tardiness bounds* as our evaluation metric, where a task's relative tardiness is computed by dividing its tardiness by its period. For each task system, we computed exact tardiness bounds using Theorem 43 and tardiness bounds using Theorem 28 under GEDF (EDF-EXT and EDF-TGT, respectively) and FIFO (FIFO-EXT and FIFO-TGT, respectively). We also computed tardiness bounds under GEDF and FIFO using methods by Devi and Anderson [9] (EDF-DA), and Leontyev and Anderson [19] (FIFO-LA), respectively. We did not compare against the tighter bounds under GEDF from [12, 27] as they are computationally expensive to compute and have trends similar to EDF-DA [27] (In our attempt to compute tardiness bounds from [27] using the most efficient implementation from [18], we found that computing tardiness bounds for a task system on 16 or more processors can take several hours to complete). We measured the time taken to compute EDF-EXT and FIFO-EXT for each task system. We present a representative selection of our results in Fig. 6.

▶ **Observation 1.** *For heavy utilizations, the average relative tardiness bound for EDF-TGT was 7.58% smaller than for EDF-DA for large processor counts (at least 12 processors). The maximum relative tardiness bound for EDF-TGT was 56.83% smaller than for EDF-DA. The average and maximum relative tardiness bounds for FIFO-TGT were 58.47% and 83.70% smaller than for FIFO-LA, respectively.*

This can be seen in insets (a) and (b) of Fig. 6. While the mean for EDF-DA can be smaller than that for EDF-TGT for smaller processor counts (Fig. 6(a)), the maximum for EDF-DA is generally larger than for EDF-TGT (Fig. 6(b)). This is because EDF-DA and FIFO-LA tend to be larger when task utilizations are large.

▶ **Observation 2.** *For light utilizations, the average and maximum relative tardiness bounds for EDF-TGT were 1199% and 447% larger than for EDF-DA, respectively. The average and maximum relative tardiness bounds for FIFO-TGT were 90.47% and 13.65% larger than for FIFO-LA, respectively.*

This can be seen in Fig. 6(c). EDF-DA (resp., FIFO-LA) is tighter than EDF-TGT (resp., FIFO-TGT) for light per-task utilizations. This is because EDF-DA and FIFO-LA are functions of the sum of largest $m-1$ task utilizations, while EDF-TGT and FIFO-TGT do not depend on task utilizations. Note that it is possible to derive a tardiness bound that is a function of task utilizations by a method similar to [9,19] using the upper bound on LAG from Lemma 38.

▶ **Observation 3.** *Across all task systems, the average relative tardiness for EDF-EXT and FIFO-EXT was 0.09 and 0.17, respectively. The maximum relative tardiness for EDF-EXT and FIFO-EXT was 4.75 and 14.0, respectively. For heavy and light utilizations, the average relative tardiness for EDF-EXT was 1.11% larger and 99.9% smaller than FIFO-EXT, respectively.*

This can be seen in insets (a), (b), and (c) of Fig. 6. Average and maximum relative tardiness are usually smaller under GEDF than FIFO. However, average and maximum tardiness can be larger under GEDF than FIFO (see Ex. 33).

▶ **Observation 4.** *For heavy utilizations and high system utilization, the average relative tardiness for EDF-EXT was larger than FIFO-EXT. For the remaining utilization distributions, the average relative tardiness for EDF-EXT were smaller than FIFO-EXT.*

This can be seen in insets (d), (e), and (f) of Fig. 6. GEDF has smaller average relative tardiness than FIFO on average.

▶ **Observation 5.** *Across all task systems, the average time to compute EDF-EXT and FIFO-EXT was 386 and 64.5 ms, respectively. The maximum time to compute EDF-EXT and FIFO-EXT was 6.95 and 0.63 sec, respectively. (These computations were done on 2.50 GHz Intel processors with 30M cache and 256GB RAM.)*

This implies that exact tardiness bounds can often be efficiently computed. The running time increases when the number of processors is large. Note that the running time may increase significantly when $T_{max}$ is large.

## 6 Conclusion

In this paper, we have presented a tardiness bound for pseudo-harmonic periodic tasks under GEL schedulers. This is the first tardiness bound under any practical global scheduler that does not increase with respect to the number of tasks or processors. We have shown the tightness of our bound and provided a method to determine similar tardiness bounds for pseudo-harmonic sporadic tasks. We have also provided a method to compute exact tardiness bounds for pseudo-harmonic periodic tasks under GEL schedulers.

Several other issues regarding tardiness under global schedulers remain unresolved. For example, we plan to investigate whether non-preemptive GEL schedulers provide tardiness bounds that do not depend on the processor or task count for pseudo-harmonic task systems.

We also want to investigate whether a similar tardiness bound exists for non-pseudo-harmonic task systems under any GEL scheduler. Finally, we want to devise faster methods to compute exact tardiness bounds for both pseudo-harmonic and non-pseudo-harmonic task systems.

## References

1 S. Ahmed and J. Anderson. A soft-real-time-optimal semi-clustered scheduler with a constant tardiness bound. In *RTCSA'20*, pages 1–10. IEEE, 2020.

2 J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

3 S. Anssi, S. Kuntz, S. Gérard, and F. Terrier. On the gap between schedulability tests and an automotive task model. *Journal of Systems Architecture*, 59(6):341–350, 2013.

4 S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

5 S. Baruah, J. Gehrke, and C. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *IPSS'95*, pages 280–288. IEEE, 1995.

6 J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Using harmonic task-sets to increase the schedulable utilization of cache-based preemptive real-time systems. In *RTCSA '96*, pages 195–202. IEEE, 1996.

7 L. Cucu-Grosjean and J. Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of Systems Architecture*, 57(5):561–569, 2011.

8 U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *RTSS '05*, pages 330–341. IEEE, 2005.

9 U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

10 S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

11 J. Erickson, J. Anderson, and B. Ward. Fair lateness scheduling: reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems*, 50(1):5–47, 2014.

12 J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global EDF. In *ECRTS'10*, pages 14–23. IEEE, 2010.

13 L. Abeni et al. Deadline task scheduling. Linux kernel documentation. `https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-deadline.rst`. [Online; accessed 06-May-2021].

14 Y. Fu, N. Kottenstette, Y. Chen, C. Lu, X. Koutsoukos, and H. Wang. Feedback thermal control for real-time systems. In *RTAS'10*, pages 111–120. IEEE, 2010.

15 J. Goossens, E. Grolleau, and L. Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Systems*, 52(6):808–832, 2016.

16 C. Kenna, J. Herman, B. Brandenburg, A. Mills, and J. Anderson. Soft real-time on multiprocessors: Are analysis-based schedulers really worth it? In *RTSS'11*, pages 93–103. IEEE, 2011.

17 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *WATERS'15*, 2015.

18 M. Leoncini, M. Montangero, and P. Valente. A parallel branch-and-bound algorithm to compute a tighter tardiness bound for preemptive global EDF. *Real-Time Systems*, 55(2):349–386, 2019.

19 H. Leontyev and J. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *ECRTS'07*, page 71. IEEE, 2007.

20 H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71, 2010.

21 H. Li, J. Sweeney, K. Ramamritham, R. Grupen, and P. Shenoy. Real-time support for mobile robotics. In *RTAS'03*, pages 10–18. IEEE, 2003.

22 V. Nélis, P. Yomsi, and J. Goossens. Feasibility intervals for homogeneous multicores, asynchronous periodic tasks, and FJP schedulers. In *RTNS'13*, pages 277–286. ACM, 2013.

23 P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *RTSS'11*, pages 104–115. IEEE, 2011.

24 C. Shih, S. Gopalakrishnan, P. Ganti, M. Caccamo, and L. Sha. Scheduling real-time dwells using tasks with synthetic periods. In *RTSS'03*, pages 210–219. IEEE, 2003.

25 S. Tang and J. Anderson. Towards practical multiprocessor EDF with affinities. In *RTSS'20*, IEEE, pages 89–101, 2020.

26 S. Tang, S. Voronov, and J. Anderson. GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In *ECRTS'19*, pages 13:1–13:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

27 P. Valente. Using a lag-balance property to tighten tardiness bounds for global EDF. *Real-Time Systems*, 52(4):486–561, 2016.

28 K. Yang and J. Anderson. On the soft real-time optimality of global EDF on uniform multiprocessors. In *RTSS'17*, pages 319–330. IEEE, 2017.

# Feasibility Analysis of Conditional DAG Tasks

Sanjoy Baruah
Washington University in Saint Louis, MO, USA

Alberto Marchetti-Spaccamela
La Sapienza University, Rome, Italy

## Abstract

Feasibility analysis for Conditional DAG tasks (C-DAGs) upon multiprocessor platforms is shown to be complete for the complexity class PSPACE. It is shown that as a consequence integer linear programming solvers (ILP solvers) are likely to prove inadequate for such analysis. A demarcation is identified between the feasibility-analysis problems on C-DAGs that are efficiently solvable using ILP solvers and those that are not, by characterizing a restricted class of C-DAGs for which feasibility analysis is shown to be efficiently solvable using ILP solvers.

**2012 ACM Subject Classification** Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Real-time schedulability

**Keywords and phrases** Multiprocessor feasibility analysis, Conditional Directed Acyclic Graphs, PSPACE-complete

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.12

**Funding** *Sanjoy Baruah*: National Science Foundation Grants CNS-1814739 and CPS-1932530.

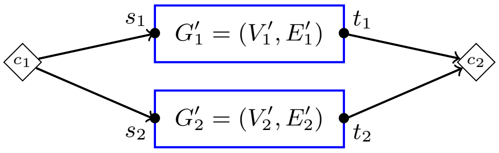
## 1 Introduction

This paper investigates the *feasibility analysis problem for C-DAG tasks*: the problem of determining whether a given real-time workload which is specified in the Conditional Directed Acyclic Graph task (C-DAG) model [6, 17] and is to be implemented upon a particular multiprocessor platform, can be scheduled to always complete by a specified deadline. Since it follows from earlier results [19] that a simpler version of this problem, in which the workload is specified as a DAG (i.e., without any conditional nodes) is already NP-hard in the strong sense, we should not expect to obtain algorithms with polynomial or pseudo-polynomial running times that solve our problem exactly. Two approaches to such feasibility analysis problems (i.e., those that are provably NP-hard in the strong sense) have previously been investigated in the real-time literature: (i) design approximation algorithms that run in polynomial or pseudo-polynomial time; or (ii) derive exact algorithms that (necessarily, assuming $P \neq NP$) run in exponential time. The latter approach is often based upon transforming the feasibility analysis problem into an integer linear program (ILP), and leveraging the tremendous recent improvements that have been obtained in the performance of ILP solvers to achieve running times that are acceptable in practice for reasonably large problem instances. In this paper we prove that an approach based on transformation to ILPs is unlikely to be applicable to the general C-DAG feasibility-analysis problem – to our knowledge, this is amongst the first feasibility-analysis problems for which such a negative result regarding the use of ILPs has been obtained in the real-time literature. We also identify an important restricted case for which ILP-solvers can in fact prove helpful: this special case essentially limits the number of conditional constructs that may be present.

**Our Contributions.** Two major technical results are proved in this paper:

1. the C-DAG feasibility analysis problem is PSPACE complete; and
2. it is in NP if the number of conditional constructs is *a priori* bounded by a constant.

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 12; pp. 12:1–12:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Vertices $s_1$ and $t_1$ (vertices $s_2$ and $t_2$, resp.) are the sole source vertex and sink vertex of $G'_1$ (of $G'_2$, resp.).

**Figure 1** A canonical conditional construct.

While at first glance these may appear to be highly theoretical results that are a poor fit for ECRTS, we will establish that they do in fact have major implications to real-time systems design and implementation. We will show that it follows from our first result that it is highly unlikely we will be able to solve general C-DAG feasibility analysis problems in polynomial time even when calls to an ILP solver are "for free" (and hence, regardless of how good your ILP solver may be). The second result clearly shows that the root cause of this is the presence of the conditional constructs, and thereby demarcates the boundary between feasibility-analysis problems that are efficiently transformable to ILPs and those that are not. We also offer evidence that the size of the ILP for solving an instance of this restricted case grows exponentially with the number of conditional constructs that are present. This in turn suggests a design guideline: conditional constructs be considered as a scarce "resource" to be used only when their increased expressiveness is essential, since their presence can slow down feasibility analysis exponentially.

**Organization.** The remainder of this manuscript is organized as follows. We describe the Conditional DAG model in Section 2, and briefly review some needed results from complexity theory in Section 3. Our main technical results are in Section 4 (the PSPACE completeness proof) and Section 5 (the more tractable special case). We conclude in Section 6 by listing some additional implications of our findings and placing these within the context of related research, and briefly list some interesting directions for future research.

## 2 The Conditional DAG (C-DAG) Model

Task models based upon Directed Acyclic Graphs (DAGs) seek to expose parallelism in real-time workloads: the *sporadic DAG model* [7] (see [4, Chapter 21] for a text-book description) is an early example. A task in this model is specified as a 3-tuple $(G, D, T)$, where $G$ is a directed acyclic graph (DAG), and $D$ and $T$ are positive integers representing the relative deadline and period parameters of the task respectively. The task repeatedly releases *dag-jobs*, each of which is a collection of sequential jobs. Successive dag-jobs are released a duration of at least $T$ time units apart. The DAG $G$ is specified as $G = (V, E)$, where $V$ is a set of vertices and $E$ a set of directed edges between these vertices. Each $v \in V$ represents a job, which corresponds to the execution of a sequential piece of code and is characterized by a worst-case execution time (WCET). The edges represent dependencies between the jobs: if $(v_1, v_2) \in E$ then job $v_1$ must complete execution before job $v_2$ can begin execution. A release of a dag-job of the task at time-instant $t$ means that all $|V|$ jobs $v \in V$ are released at $t$. If a dag-job is released at time $t$ then all $|V|$ jobs that were released at $t$ must complete execution by time $t + D$.

**Conditional DAG tasks.** The Conditional DAG (C-DAG) task model was introduced [6, 17] to model the execution of conditional (e.g., **if-then-else**) constructs in parallel real-time code. A C-DAG task, too, is specified as a 3-tuple $(G, D, T)$, where $G = (V, E)$ is a DAG, and $D$ and $T$ are positive integers denoting the relative deadline and period parameters of the task. They differ from regular sporadic DAGs in that certain vertices $\in V$ are designated as *conditional vertices* that are defined in matched pairs, each such pair defining a *conditional construct*. Let $(c_1, c_2)$ be such a pair in the DAG $G = (V, E)$ – see Figure 1. Informally speaking, vertex $c_1$ represents a point in the code where a conditional expression is evaluated and, depending upon the outcome of this evaluation, control will subsequently flow along one of two different possible branches. It is required that these two different branches meet again at a common point in the code, represented by the vertex $c_2$. More formally,

1. There are two outgoing edges from $c_1$ in $E$ (say, to the vertices $s_1$ and $s_2$), and two incoming edges to $c_2$ (say, from the vertices $t_1$ and $t_2$), in $E$ – see Figure 1.
2. For each $\ell \in \{1, 2\}$, let $V'_\ell \subseteq V$ and $E'_\ell \subseteq E$ denote all the vertices and edges on paths reachable from $s_\ell$ that do not include vertex $c_2$. By definition, $s_\ell$ is the sole source vertex of the DAG $G'_\ell \stackrel{\text{def}}{=} (V'_\ell, E'_\ell)$. Vertex $t_\ell$ must be the sole sink vertex of $G'_\ell$.
3. It must hold that $V'_1 \bigcap V'_2 = \emptyset$. Additionally for each $\ell \in \{1, 2\}$, with the exception of $(c_1, s_\ell)$ there should be no edges in $E$ into vertices in $V'_\ell$ from vertices that are not in $V'_\ell$.

Edges $(v_1, v_2)$ between pairs of vertices neither of which are conditional nodes represent precedence constraints exactly as in traditional sporadic DAGs, while edges involving conditional nodes represent conditional execution of code. More specifically, let $(c_1, c_2)$ denote a defined pair of conditional vertices that together define a conditional construct. The semantics of conditional DAG execution mandate that

- After the job $c_1$ completes execution, exactly one of its two successor jobs becomes eligible to execute; it is not known beforehand which successor job this may be.
- Job $c_2$ begins to execute upon the completion of exactly one of its two predecessor jobs.

In the remainder of this paper we make the *simplifying assumption that each of the conditional vertices $c_1$ and $c_2$ demarcating a conditional construct has zero execution time.*

**The C-DAG feasibility analysis problem.** We are interested, from a real-time systems perspective, in understanding how to implement specified collections of C-DAG tasks upon a shared multiprocessor platform in a correct and resource-efficient manner. The *federated scheduling* paradigm [15], in which each task is restricted to execute upon a specified subset of the processors (and each processor is assigned to no more than one task), is a widely-studied approach for implementing collections of tasks represented using DAG-based models upon multiprocessor platforms. It is readily seen that federated scheduling of constrained-deadline tasks – tasks $(G, D, T)$ for which the deadline parameter $D$ is no larger than the period $T$ – reduces to the problem of scheduling a single C-DAG upon a dedicated set of processors within a duration not exceeding the relative deadline parameter. Hence the problem considered in this paper is this:

▶ **Definition 1** (The C-DAG feasibility analysis problem). GIVEN *a C-DAG $G$, a number $m \in \mathbb{N}$ of processors upon which $G$ is to execute, and a relative deadline parameter $D$,* DETERMINE *whether it is* feasible *to schedule $G$ on the $m$ processors such that it always completes execution within an interval of duration $D$, regardless of which conditional constructs in $G$ evaluate to true and which evaluate to false?* ⌟

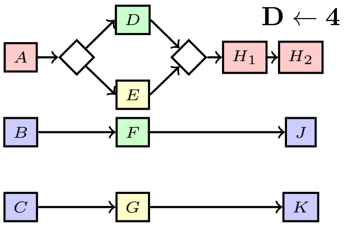
The problem definition above is incomplete: several variants can be defined based upon restrictions that are placed on how jobs may execute. For instance permitting or prohibiting preemption results in different variants. Variants may be also be defined based upon which processors each job is allowed to execute on:

**global:** any job may execute upon any processor, and the decision as to which processor a job executes upon may be made at run-time. When preemption is permitted, a preempted job may resume execution upon a different processor.

**partitioned:** each job may execute upon only one processor, and the determination as to which processor a job executes upon is made prior to run-time.

**restricted** (or typed [13]): each job is pre-assigned to a particular processor. I.e., a mapping from vertices to processors is provided as part of the problem specifications.

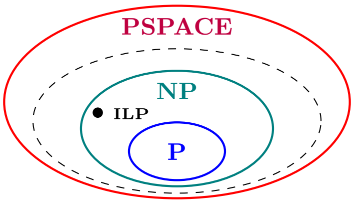
**Why this is a difficult problem.** It has been widely recognized [11, 6, 17, 22] that *combinatorial explosion* is a major reason why C-DAG feasibility analysis is such a difficult problem: exponentially many different combinations of outcomes are possible of the evaluation of the conditional constructs in a single task, each of which may require a very different collection of jobs to be scheduled for execution. There is, however, an additional aspect to the difficulty of this problem that has received somewhat less attention: its inherently *on line* nature. Consider the following simple illustrative example for a typed C-DAG (i.e., where vertices are pre-assigned to individual processors):



**Example:** *Each vertex has WCET equal to one (except the conditional vertices – recall they have WCET zero). Processor assignments are color-coded: $A$, $H_1$, & $H_2$ share a processor, as do $B, C, J$, & $K$; $D$ & $F$; and $E$ & $G$.*
*If the conditional construct executes $D$, then $C$ should execute during $[0,1]$ – otherwise the "blue" processor will idle over $[2,3]$. Else (i.e., the conditional construct executes $E$), $B$ should execute during $[0,1]$.*

There are two possible outcomes of the sole conditional construct, and it may be verified that upon either outcome the set of vertices that must be executed is individually schedulable. However, which of vertices $B$ or $C$, both assigned to the same processor, should execute over time-interval $[0,1]$ necessarily differs in these two schedules and hence depends upon the outcome of the conditional construct's evaluation. But the conditional construct is only executed *after* time-instant 1, and hence this information is revealed too late. Thus this C-DAG is *in*feasible despite the sets of vertices needing to be executed upon either outcome being feasible.

**Summarizing Prior Complexity Results.** Ullman showed [19] that it is NP-complete in the strong sense to determine whether a given DAG can be scheduled to meet a specified deadline under global or partitioned scheduling upon an identical multiprocessor platform, regardless of whether preemption is permitted or forbidden. Jansen subsequently showed [13] that feasibility analysis of DAGs is NP-hard in the strong sense for restricted/ typed C-DAGs (where each job is pre-assigned to a particular processor), again under both preemptive and non-preemptive scheduling. Since these basic problems are already NP-hard in the strong sense, so are the corresponding problems for the more general C-DAG model. It is easily seen that all these problems are also in NP for (regular) DAGs.



The innermost (blue) solid line represents the problems in P, the intermediate (teal) one includes problems that are in NP, and the outermost (red) one further includes problems that are in PSPACE. The dotted black line depicts the class $P^{NP}$.
As shown in the Venn diagram, the problem of solving ILPs is in NP but not in P (assuming P≠NP).

**Figure 2** Venn diagram depicting the relationship between some complexity classes.

## 3 Computational Complexity: Some Background

We now provide a brief introduction to concepts of computational complexity theory that are used in this manuscript.[1] We will make reference to the following four complexity classes:

1. P is the set of problems that can be *solved* by algorithms with running time polynomial in the size of their inputs.
2. NP is the set of problems that can be *verified* by algorithms with running time polynomial in the size of their inputs.
3. $P^{NP}$ is the set of problems that can be solved in polynomial time by an algorithm that has access to an *oracle* for some NP-complete problem, where an oracle can be thought of as a "black box" that is able to solve a specific decision problem in a single step.
4. PSPACE is the set of problems that can be solved by algorithms using an amount of *space* (memory) that is polynomial in the size of their inputs. Since this complexity class has not previously been widely used in real-time scheduling theory, we discus it a bit more below, and provide some intuition of its relationship to C-DAG feasibility analysis.

It is widely believed, although not proved, that $\big(\mathrm{P} \subsetneq \mathrm{NP} \subsetneq \mathrm{P}^{\mathrm{NP}} \subsetneq \textsc{pspace}\big)$ – see Figure 2.

**PSPACE.** The class PSPACE can be thought of as representing the existence of a winning strategy for a particular player in bounded-length perfect-information games that can be played in polynomial time. I.e., consider a two-player game where players alternate making moves for a total of $n$ moves. Given moves $m_1, \ldots, m_n$ by the players, let $M(m_1, \ldots, m_n) = 1$ if and only if player 1 has won the game. Then player 1 has a winning strategy in the game if and only if there exists a move $m_1$ that player 1 can make such that for every possible response $m_2$ of player 2 there is a move $m_3$ for player 1, ... and so on. Formalizations of many popular two-player games, including checkers, generalized geography, and Sokoban, have been proven to be PSPACE-complete [12].

We can cast C-DAG feasibility in this two-player game framework. Given a C-DAG and a deadline $D$, then the first move of player 1 (the scheduler) is to decide the set of jobs to be scheduled until the first branch is executed; then player 2 (the environment) decides the outcome of the branch. The game continues until the scheduling is completed and the first player wins the game if and only if its strategy is able to complete the schedule in $D$ time units for all outcomes of branches (i.e. all decisions of the second player).

[1] In order to keep things simple the presentation in this section is intentionally informal and not always precise: for instance, while most of the concepts discussed below differ in their applicability to *decision problems* – those for which there is a "YES/ NO" answer – and *optimization problems*, we do not make this distinction here but treat both decision and optimization problems in similar fashion.

**ILP solvers.** Determining whether an integer linear program (ILP) has a solution or not is known to be NP-complete in the strong sense [14]. Assuming P $\neq$ NP, this implies that ILP solvers with polynomial or pseudo-polynomial running time cannot be developed. Despite this inherent intractability, however, the optimization community has devoted immense effort to devise very efficient implementations of ILP solvers, and highly optimized libraries with such efficient implementations are widely available today in both open-source and commercial offerings. Modern ILP solvers, executing upon powerful computing clusters, are commonly capable of solving ILPs with tens of thousands of variables and constraints.

## 4 C-DAG feasibility analysis is PSPACE-complete

One of our main results is a negative one: that the C-DAG feasibility analysis problem (Definition 1) is PSPACE-hard for all the variants – preemptive and non-preemptive; global and partitioned and restricted (or typed) – described in Section 2. As stated in Section 3 above, a PSPACE complete problem is highly unlikely to be in NP or $\text{P}^{\text{NP}}$; hence we cannot solve it in polynomial time by making additional calls to an ILP-solver, even if each such call took $\Theta(1)$ (i.e., constant) time. In the remainder of this section we will prove this intractability result for the variant[2] of the C-DAG feasibility analysis problem where preemption is permitted and migration is restricted (i.e., each job is pre-assigned to a particular processor):

▶ **Theorem 1.** *The C-DAG feasibility problem when each job is pre-assigned to a particular processor is* PSPACE *complete.*

It is trivial to show that this problem is in PSPACE – an algorithm that repeatedly simulates the scheduling of the C-DAG under all possible combinations of outcomes of the conditional constructs would require polynomial space. The rest of this section is devoted to proving that this problem is also PSPACE-hard. (Note that this hardness result strengthens a recent result [2] showing this problem to be hard for the complexity class co-$\text{NP}^{\text{NP}}$, since the near-consensus view in computational complexity theory is that the class co-$\text{NP}^{\text{NP}}$ is strictly contained in the class PSPACE.) PSPACE-hardness for our C-DAG feasibility analysis problem is proved by deriving a polynomial-time reduction to the C-DAG feasibility analysis problem from the following problem, which has previously [18, 21] been shown to be PSPACE complete:

▶ **Definition 2** (The Quantified Boolean Formula Problem (QBF)).
Instance. A boolean formula in the following form:

$$\exists x_1 \; \forall y_1 \; \exists x_2 \; \forall y_2 \; \ldots \exists x_n \; \forall y_n \bigwedge_{j=1}^{m} (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}) \tag{1}$$

where each $x_i$ and each $y_i$ is a boolean variable, and each $\ell_{j,k}$ is one of the $x_i$ or $y_i$ Boolean variables or its negation.
Question. Does this formula evaluate to **true**? ⌟

We will describe a polynomial-time algorithm that accepts as input a Boolean formula of the form given in Expression 1 above, and outputs a C-DAG, an assignment of jobs of the C-DAG to processors, and a deadline $D \overset{\text{def}}{=} 2n + 3$, such that the C-DAG can complete

[2] We have also proved this result for the variant that allows for global preemptive scheduling. We are choosing to present the variant with pre-assigned processors for pedagogical reasons: the main ideas in the proof of the hardness of the global preemptive case are also revealed in this proof while a lot of grungy details that are not particularly novel but must be addressed for the global preemptive version are not needed here.

execution by the deadline if and only if Expression 1 is **true**. Since QBF is known to be PSPACE-complete, this polynomial-time reduction from QBF to C-DAG feasibility analysis suffices to show that C-DAG feasibility analysis is PSPACE hard. We start with a high-level overview of our polynomial-time reduction.

- We will define three kinds of "gadgets" – subgraphs that have each been designed to achieve some particular purpose – in Sections 4.1, 4.2, and 4.3. The first kind is used to represent the clauses in Expression 1; the second, the existentially quantified (i.e., $x_i$) variables and the third, the universally quantified (i.e., $y_i$) variables. Our C-DAG will be the union of $m$ gadgets of the first kind, $n$ gadgets of the second kind, and $n$ gadgets of the third kind.
- For each boolean variable $x_i$ ($y_i$, respectively), our C-DAG will have two jobs labeled $X_i$ and $\neg X_i$ ($Y_i$ and $\neg Y_i$, respectively). We will state that job $X_i$ "***corresponds to***" literal $x_i$ and job $\neg X_i$ corresponds to the literal $\neg x_i$ (analogously, that $Y_i$ corresponds to $y_i$ and $\neg Y_i$ corresponds to $\neg y_i$).

  We will see, in Sections 4.2 and 4.3, that we construct the gadgets for the $x_i$'s and the $y_i$'s in a manner that enforces the constraint that at most one of each pair of jobs $X_i$ and $\neg X_i$ ($Y_i$ and $\neg Y_i$, respectively) can execute to completion by time-instant $2n$ in any schedule. We can think of all these jobs that complete execution by time-instant $2n$ as defining a truth assignment to the $2n$ variables $\{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$: boolean variable $x_i$ is assigned **true** if job $X_i$ is executed and **false** if $\neg X_i$ is executed, and analogously for the $y_i$ variables.[3] Furthermore, we will see in Sections 4.2 and 4.3 that such a truth assignment happens in a manner that is consistent with the order and interpretation of the quantifiers upon the boolean variables.
- We will show, in Section 4.1 below, that the gadget representing each clause will complete by the deadline if and only if at least one of the literals in the clause evaluates to **true** in the truth assignment defined as above. Therefore, the gadgets representing all the clauses can complete by the deadline if and only if the truth assignment defined above is a satisfying one for all the clauses.

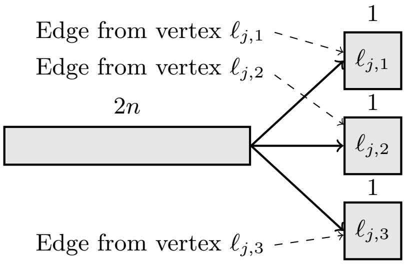
We detail the construction of the three kinds of gadgets in Sections 4.1–4.3; in Section 4.4 we show that the C-DAG thus obtained is feasible if and only if Expression 1 is true, and hence this is indeed a polynomial-time reduction from QBF to C-DAG feasibility analysis.

### 4.1 Gadget for representing the clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$

For the $j$'th clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$, we have four jobs with precedence constraints as depicted in Figure 3, all of which are assigned to a single dedicated processor. The WCET of each job is written above the job in Figure 3. We will say that each of the three unit-sized jobs "***represents***" one of the three literals in the clause. Observe that the sum of the WCETs of the four jobs is $2n + 1 + 1 + 1 = \mathbf{2n + 3}$, which equals the deadline $D$; since all these jobs are assigned to the same processor the processor must therefore never idle over $[0, D]$ in schedules that meet the deadline. This enforces the following schedule for these jobs:

1. the job with WCET $2n$ must execute over the interval $[0, 2n]$, and
2. at least one of the three unit-sized jobs, each of which has one additional input edge from the job corresponding to the literal that it represents, must become eligible to execute at time-instant $2n$.

[3] If neither $X_i$ nor $\neg X_i$ (neither $Y_i$ nor $\neg Y_i$, respectively) are executed for any $i$, the truth assignment will be a partial one.


Edge from vertex $\ell_{j,1}$
Edge from vertex $\ell_{j,2}$
Edge from vertex $\ell_{j,3}$
$2n$
1
1
1
$\ell_{j,1}$
$\ell_{j,2}$
$\ell_{j,3}$

These four jobs are all assigned to the same processor; no other jobs are assigned to this processor. The WCET of each job is written above the job (i.e., the job with no predecessors has WCET $= 2n$ and the other three jobs each have WCET $= 1$). Each of the unit-sized jobs represents a literal of the clause; the dotted lines represent edges from the jobs that correspond to the literals (the notions of *representation* and *correspondence* are both explained in Section 4).

**Figure 3** "Gadget" representing the $j$'th clause.

Equivalently, in order for the part of the C-DAG we are constructing that is represented by this gadget to complete by the deadline, it is necessary that the truth assignment defined by the $X_i$, the $\neg X_i$, the $Y_i$ and the $\neg Y_i$ jobs that completed execution by time-instant $2n$ have at least one of the literals $\ell_{j,1}$, $\ell_{j,2}$, and $\ell_{j,3}$ assigned the value true. I.e., this truth assignment must be a satisfying one for the clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$.

Hence all $m$ gadgets of the form depicted in Figure 3, constructed for all $m$ clauses in Expression 1, can complete by the deadline if and only if the truth assignment defined by the $X_i$, the $\neg X_i$, the $Y_i$ and the $\neg Y_i$ jobs that completed execution by time-instant $2n$ is a satisfying one for each of the clauses in the QBF given in Expression 1. This is formally stated in Fact 1:

▶ **Fact 1.** A schedule can complete the jobs representing (as depicted in Figure 3) all $m$ clauses by the deadline $D = 2n + 3$ if and only if the truth assignment, defined by the jobs in $\bigcup_{1 \le i \le n} \{X_i, \neg X_i, Y_i, \neg Y_i\}$ that have executed to completion by time-instant $2n$ in the schedule, is a satisfying assignment for all the clauses. ⌟

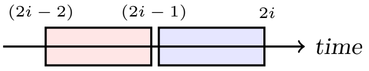
**Requirements of the remaining gadgets.** The remainder of the C-DAG – i.e., the gadgets for the $x_i$ and the $y_i$ boolean variables – must ensure that this truth assignment that is defined by the $X_i$, the $\neg X_i$, the $Y_i$ and the $\neg Y_i$ jobs that completed execution by time-instant $2n$ is an accurate reflection of the alternating quantifiers in Expression 1:

$$\exists x_1 \; \forall y_1 \; \exists x_2 \; \forall y_2 \; \ldots \exists x_n \; \forall y_n$$

This desired alternation of quantifiers is achieved by ensuring the C-DAG is constructed to enforce the requirement that for each $i$, $1 \le i \le n$,

1. Prior to time-instant $2n$, in any correct schedule the scheduler can execute the pair of jobs $X_i$ and $\neg X_i$, both of which are assigned to the same processor, only over the interval $[2i - 2, 2i - 1]$ – see Figure 4. Therefore, it can choose to execute only one of this pair of jobs to completion prior to time-instant $2n$. (We will also see that it can execute the other job in the pair over $[2n, 2n + 1]$; hence both complete by time-instant $2n + 1$.)
2. Prior to time-instant $2n$, in any correct schedule the scheduler can execute only $\underline{\text{one}}$ of the pair of jobs $Y_i$ and $\neg Y_i$, over the interval $[2i - 1, 2i]$ – see Figure 4. *The decision as to which job in the pair is able to execute over* $[2i - 1, 2i]$ *is not made by the scheduler, but is determined during run-time based on whether certain conditional constructs evaluate to true or false.* (We will also see that the scheduler can execute the other job in the pair over the time-interval $[2n, 2n + 1]$; hence both the jobs complete by time-instant $2n + 1$.)

The existential quantification ($\exists$) of the $x_i$ variables is reflected by the fact that the scheduler gets to decide whether to execute $X_i$ or $\neg X_i$ over the interval $[2i - 2, 2i - 1]$, while the



The scheduler may choose to execute one of $\{X_i, \neg X_i\}$ over the interval $[2i-2, 2i-1]$. Run-time evaluation of conditional constructs enables only one of $\{Y_i, \neg Y_i\}$ to execute over interval $[2i-1, 2i]$.

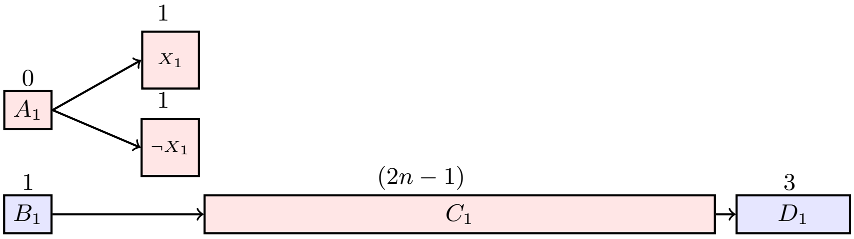
**Figure 4** Illustrating the schedule over $[2i-2, 2i]$ for each $i$.

universal quantification ($\forall$) of the $y_i$ variables is reflected by the fact that the environment (i.e., run-time conditions) determines which of $Y_i$ or $\neg Y_i$ to execute, and the scheduler must make subsequent scheduling decisions for both outcomes (i.e., regardless of whether $Y_i$ or $\neg Y_i$ is the job that was selected) by the environment. Notice that the order of the quantifiers is also maintained: the scheduler must decide to execute one of $\{X_i, \neg X_i\}$ *before* one of $\{Y_i, \neg Y_i\}$ is scheduled. And *after* one of $\{Y_i, \neg Y_i\}$ is chosen for execution by the environment, the scheduler must decide to schedule one of $\{X_{i+1}, \neg X_{i+1}\}$, and so on. In this manner the truth assignment to the variables $\{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$ that is defined by the schedule based on the jobs that complete execution by time-instant $2n$ reflects the order and alternation of the quantifiers in Expression 1.

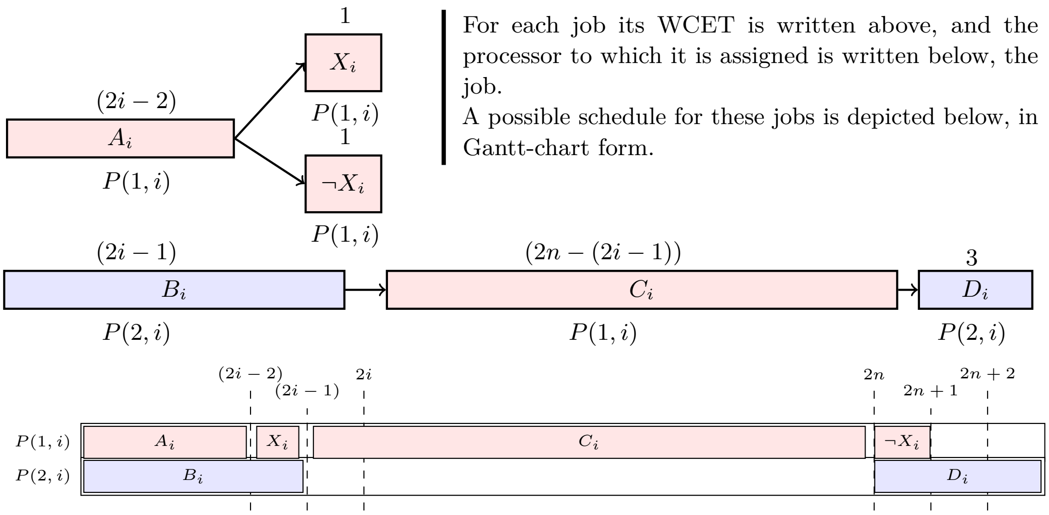
It remains to describe how these restrictions on the execution of the $X_i, \neg X_i, Y_i$, and $\neg Y_i$ jobs in a manner that reflects the order and nature of the quantifiers is enforced – this we do in describing our other two kinds of gadgets. As stated previously, we will have one gadget for each $x_i$ variable and one for each $y_i$ variable; each gadget is defined on a unique set of jobs that are assigned to a unique set of processors. Our C-DAG is the union of all $2n$ of these gadgets and the $m$ subgraphs of the form of Figure 3 (one per clause).

## 4.2 Gadget for enforcing the desired execution of $X_i$ and $\neg X_i$

We first discuss the instantiation of this gadget for $(i = 1)$, before subsequently describing the general case. The four jobs labeled $A_1$, $B_1$, $C_1$ and $D_1$ depicted below serve to ensure that prior to time-instant $2n$ the scheduler can execute the two jobs labeled $X_1$, $\neg X_1$ only over the time-interval $[0, 1]$ in any correct schedule.



The jobs $A_1, X_1, \neg X_1$, and $C_1$ are all assigned to one processor, while $B_1$ and $D_1$ are both assigned to another processor; furthermore, these two processors are used for no other purpose. (In these diagrams vertex colors encode their processor assignments.) Since the chain of jobs $B_1 \to C_1 \to D_1$ has cumulative WCET $1 + (2n-1) + 3 = \mathbf{2n+3}$ which is equal to the deadline $D$, these three jobs must execute without interruption. Hence in any correct schedule the processor shared by jobs $A_1, X_1, \neg X_1$, and $C_1$ is only available to jobs $X_1$ and $\neg X_1$ during the interval $[0, 1]$, and after time $2n$. Thus at most one of these jobs may complete execution prior to time-instant $2n$, and this job must do so by executing over the interval $[0, 1]$. (We point out that the other one may execute over the time-interval $[2n, 2n+1]$ and thereby complete by time-instant $2n + 1$.)



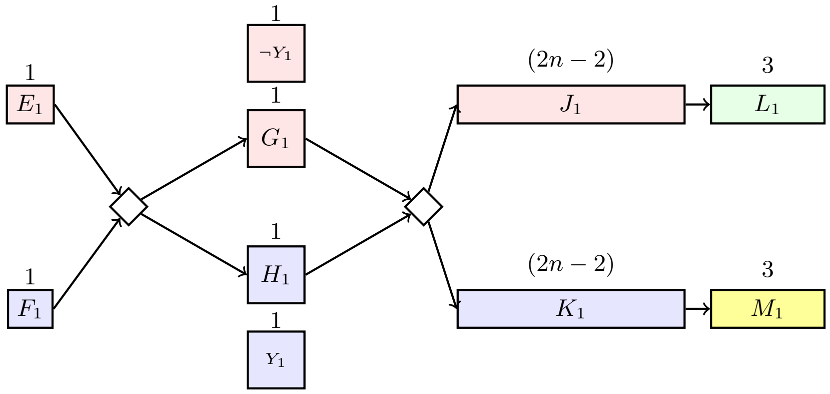
For each job its WCET is written above, and the processor to which it is assigned is written below, the job.
A possible schedule for these jobs is depicted below, in Gantt-chart form.

**Figure 5** Gadget for $X_i$, comprising the four jobs $A_i$–$D_i$ plus the two jobs $X_i$ and $\neg X_i$, and the four edges shown, assigned to the two processors $P(1,i)$ and $P(2,i)$.

The gadget depicted in Figure 5 generalizes the one described above for all $i$, $1 \leq i \leq n$. In this figure the two processors upon which the jobs are to execute are named as $P(1,i)$ and $P(2,i)$ – the processor to which each job is assigned is written below the job. A correct schedule for the jobs upon these two processors is depicted as a Gantt chart below the gadget.

## 4.3 Gadget for enforcing the desired execution of $Y_i$ and $\neg Y_i$

As with the $x_i$'s above, we first discuss the instantiation of this gadget for $(i = 1)$; we will subsequently generalize to arbitrary $i$. The eight jobs labeled $E_1$, $F_1$, $G_1$,$H_1$, $J_1$, $K_1$, $L_1$, and $M_1$ along with one conditional construct,[4] and ten edges as depicted below, together ensure that at most one of the two jobs labeled $Y_1$, $\neg Y_1$, execute over the time-interval $[1,2]$ in any correct schedule while the other must execute after time-instant $2n$; furthermore, which of $Y_1$, $\neg Y_1$ executes over $[1,2]$ is determined not by the scheduler but by which branch of the conditional construct ends up being executed during run-time.



[4] Recall that in this paper we are assuming that the two nodes demarcating the start and the end of a conditional construct each have WCET zero.

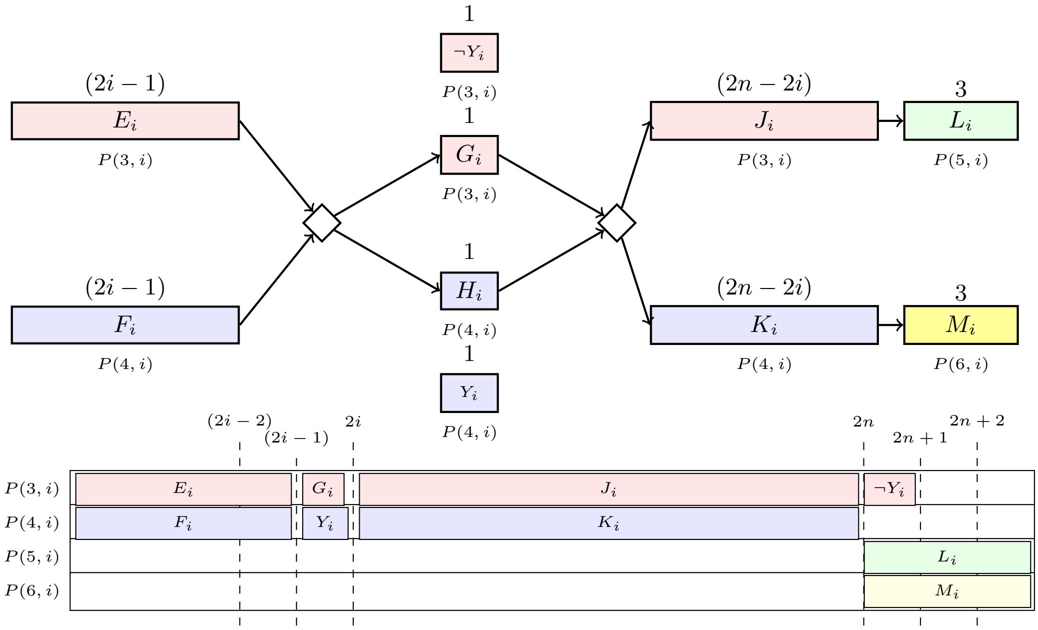


**Figure 6** Gadget for $Y_i$ (discussed in Section 4.3).

These ten jobs ($E_1$–$M_1$, plus the jobs $Y_1$ and $\neg Y_1$) are assigned to four processors in the following manner; no other jobs are assigned to any of these four processors[5];

- Jobs $E_1, G_1, \neg Y_1$ and $J_1$ are assigned to one processor.
- Jobs $F_1, H_1, Y_1$ and $K_1$ are assigned to a second processor;
- Job $L_1$ is assigned to a third processor; and job $M_1$ to a fourth processor.

Let us first suppose that during some execution of this C-DAG the conditional construct takes the upper branch (i.e., causes job $G_1$ to execute).

- Since the WCETs of the chain of jobs $E_1 \to G_1 \to J_1 \to L_1$ sum to the deadline $3n + 3$, this chain of jobs must execute without interruption in any correct schedule. This in turn implies that job $\neg Y_1$, which is assigned to the same processor as jobs $E_1, G_1$, and $J_1$, cannot execute prior to time-instant $2n$. (It may execute over the interval $[2n, 2n + 1]$ since there are no other jobs assigned to its processor.)
- In order for the chain $E_1 \to G_1 \to J_1 \to L_1$ to be able to execute without interruption, job $F_1$ must execute over the time-interval $[0, 1]$. Furthermore, the chain of jobs $K_1 \to M_1$ is only eligible to execute after the conditional construct completes: this happens when job $G_1$ completes (at time-instant 2). Note that jobs $J_1$ and $L_1$ must now execute without interruption over the interval $[2, 2n + 3]$ in order to meet the deadline $D = 2n + 3$. Therefore, the processor shared by jobs $F_1, H_1$ (which does not need to execute when the conditional construct takes the upper branch), $Y_1$, and $K_1$, is only free over the interval $[1, 2]$ prior to time-instant $2n$; this implies that the job $Y_1$ must execute over the interval $[1, 2]$ if it is to complete prior to time-instant $2n$.

[5] As in Figure 5, processor assignments are color-coded in this diagram. (Note that a fresh set of processors is used for each gadget and hence these colors do not “carry over” from Figure 5.)

When the conditional construct takes the lower branch and causes $H_1$ to execute, the situation mirrors the one above: job $\neg Y_1$ may execute over the interval $[1, 2]$ but job $Y_1$ may only execute after time-instant $2n$. Summarizing, we conclude that *in a feasible schedule that completes by the deadline $D = 2n + 3$, one of the two jobs $Y_1, \neg Y_1$ may execute over the interval $[1, 2]$ and the other may execute over $[2n, 2n + 1]$; the determination as to which does which is made during run-time based on whether the conditional construct evaluates to true or false.*

The gadget depicted in Figure 6 generalizes the one described above for all $i$, $1 \leq i \leq n$. In this figure the four processors upon which the jobs are to execute are named as $P(3, i), P(4, i), P(5, i)$ and $P(6, i)$; as in Figure 5, the processor to which each job is assigned is again written below the job. A correct schedule for the jobs upon these four processors is depicted as a Gantt chart below the gadget.

The restrictions upon the execution of the jobs $X_i, \neg X_i, Y_i$, and $\neg Y_i$ that are enforced by the gadgets of Figure 5 and Figure 6 are stated in Facts 2 and 3 below (also see Figure 4):

► **Fact 2.** For each $i$, $1 \leq i \leq n$, a scheduler may complete at most one of the two jobs $\{X_i, \neg X_i\}$ by time-instant $2n$ in any correct schedule. The choice as to which of these two jobs (if any) to complete by time-instant $2n$ must be made by the scheduler *after* it has already been decided which of the jobs $\left(\bigcup_{1 \leq j < i} \{X_j, \neg X_j, Y_j, \neg Y_j\}\right)$ will complete by time instant $2n$.

► **Fact 3.** For each $i$, $1 \leq i \leq n$, a scheduler may complete at most one of the two jobs $\{Y_i, \neg Y_i\}$ by time-instant $2n$. The determination as to which of these two jobs (if either) to complete by time-instant $2n$ is made based on the outcome of the execution of a conditional construct during run-time, *after* it has already been decided which of $\left(\bigcup_{1 \leq j < i} \{X_j, \neg X_j, Y_j, \neg Y_j\} \bigcup \{X_i, \neg X_i\}\right)$ will complete by time instant $2n$. ⌟

## 4.4 Putting the pieces together

Consider now the truth assignment to the $2n$ variables $\{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$ that is defined by the schedule over $[0, 2n]$ in the following manner: for each $i$, $1 \leq i \leq n$, boolean variable $x_i$ is assigned true if job $X_i$ is executed and false if $\neg X_i$ is executed, and boolean variable $y_i$ is assigned true if job $Y_i$ is executed and false if $\neg Y_i$ is executed. By Fact 2, a value is assigned by the scheduler to $x_i$ in this assignment *after* values have been determined for $x_j$ and $y_j$ variables for all $j < i$, while by Fact 3 the value of $y_i$ that is determined by the execution of conditional constructs at run-time happens *after* values have been determined for $x_j$ and $y_j$ variables for all $j < i$, as well as after the value of $x_i$ has been assigned by the scheduler. Fact 4 follows.

► **Fact 4.** The truth assignment to the $x_i$ and $y_i$ variables defined by the execution of $X_i, \neg X_i, Y_i$, and $\neg Y_i$ jobs over $[0, 2n]$ is done in a manner that is compliant with the order of alternation of quantifiers in Expression 1. ⌟

**Summarizing the reduction.** We have seen that the DAG we construct for a given quantified boolean formula

$$\exists x_1 \, \forall y_1 \, \exists x_2 \, \forall y_2 \, \ldots \exists x_n \, \forall y_n \bigwedge_{j=1}^{m} (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$$

comprises

1. For each of the $m$ clauses, a sub-graph with four vertices and six edges as depicted in Figure 3 that is to execute upon a single processor;
2. For each of the $n$ $x_i$ variables, a sub-graph with the six vertices and four edges as depicted in Figure 5 that is to execute upon two processors; and
3. For each of the $n$ $y_i$ variables, a sub-graph with the ten vertices, one conditional construct, and ten edges as depicted in Figure 6 that is to execute upon four processors.

It is easily seen that the reduction from quantified boolean formula to DAG is a polynomial-time one: the resulting DAG has $(4m+16n)$ vertices, $n$ conditional constructs, and $(6m+14n)$ edges, and is to be scheduled upon $(m + 6n)$ processors, and that it can be obtained in polynomial time from the quantified boolean formula.

▶ **Lemma 1.** *If Expression 1 is true, then the C-DAG constructed above can be scheduled to always complete by its deadline.*

**Proof.** Suppose that Expression 1 is true. This implies that variable $x_1$ can be assigned a value such that for every assignment of value to $y_1$ the formula

$$\exists x_2 \forall y_2 \exists x_3 \ldots \bigwedge_{j=2}^{m} (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$$

is true. If the assigned value to $x_1$ is true (false) then the scheduler completes job $X_1$ $(\neg X_1)$ by time $2n$; then, when the outcome of the first conditional construct is known, the job from amongst $\{Y_1, \neg Y_1\}$ that can be completed by time $2n$ is scheduled. By Fact 3 this decision is made before the scheduler gets to decide which job of the jobs amongst $\{X_2, \neg X_2\}$ will complete by time $2n$.

By repeated applications of Facts 2 and 3, we can ensure that the jobs amongst the $X_i$, $\neg X_i$, $Y_i$, and $\neg Y_i$ jobs that execute over the interval $[0, 2n]$ mimic each truth assignment to the boolean variables $\{x_1, x_2, ..., x_n\} \cup \{y_1, y_2, ..., y_n\}$ that are made in a manner consistent with the alternation of quantifiers in Expression 1. It follows from Fact 1 that the gadget representing each clause (these are the gadgets depicted in Figure 3) will complete by the deadline for each such truth assignment. ◀

▶ **Lemma 2.** *If the C-DAG constructed above can be scheduled to always complete by its deadline, then Expression 1 is true.*

**Proof.** Suppose that the C-DAG that we have constructed can be scheduled to always complete by its deadline, for all possible evaluations of the $n$ conditional constructs in it. (Recall that one conditional constructs is present in each of the gadgets described in Section 4.3, and these are the only conditional constructs in the C-DAG t.)

Consider the schedule for any one of the $2^n$ different possible combinations of outcomes for the execution of these $n$ conditional constructs. Fact 1 ensures that the truth assignment defined by the jobs in $\bigcup_{1 \le i \le n} \{X_i, \neg X_i, Y_i, \neg Y_i\}$ that have executed to completion by time-instant $2n$ in this schedule is a satisfying assignment for all the clauses in Expression 1; by Fact 4, this truth assignment is compliant with the order of alternation of quantifiers in Expression 1.

Our premise is that the C-DAG completes by its deadline for each of the $2^n$ different possible combinations of outcomes for the execution of the conditional constructs. It follows that each clause in Expression 1 evaluates to **true** in the corresponding truth assignments defined by the jobs in $\bigcup_{1 \le i \le n} \{X_i, \neg X_i, Y_i, \neg Y_i\}$ that have executed to completion by time-instant $2n$. Finally, it follows from Fact 3 that these $2^n$ different possible combinations of outcomes of the execution of the conditional constructs represent all possible interpretations of the universal quantifications of the $y_i$ variables. The lemma follows. ◀

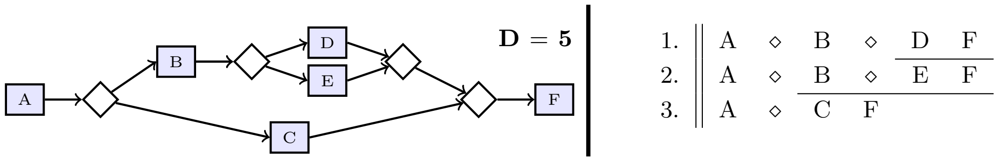


**Figure 7** A C-DAG instance with two conditional constructs. Each vertex has WCET=1, and all are assigned to the same processor. Its "certificate" of feasibility is shown on the right: it comprises three schedules, all of which are identical until the first conditional construct is executed (depicted as a ◇). The top two schedules, which correspond to the upper branch being taken, are further identical until the second conditional construct is executed.

Lemmas 1 and 2 together establish that the C-DAG feasibility problem is PSPACE-hard when each job is pre-assigned to a particular processor. We have already seen that this problem is in PSPACE; this therefore completes the proof of Theorem 1.

## 5 A More Tractable Special Case

Theorem 1 above tells us that we are unlikely to be able to efficiently (i.e., in polynomial time) reduce the problem of determining whether a C-DAG is feasible to the problem of solving one, or even polynomially many, ILPs. In this section we will show that for C-DAGs satisfying the additional restriction that *the number of conditional constructs is bounded by a constant*, the feasibility-analysis problem can indeed be polynomial-time reduced to a single ILP. Our method of showing this is indirect, and based upon the following reasoning.

- As mentioned in Section 3, it is NP-complete to determine whether an ILP has a solution [19]. It follows from definition that a consequence of a problem being NP-complete is that all other problems in NP can be reduced to it in polynomial time.
- Hence in order to show that feasibility analysis for C-DAGs in which the number of conditional constructs is bounded by some constant can be reduced to an ILP in polynomial time, it suffices to show that this feasibility analysis problem is in NP.

Below we will show that this problem is indeed in NP. We do so by appealing to the definition of the complexity class NP: as stated in Section 3, a problem is defined to be in NP if a claimed solution to any problem instance can be *verified* by an algorithm with running time polynomial in the size of the instance. Hence we will describe a *verification algorithm* [10, page 1063] that accepts as input a C-DAG and a "certificate" claiming to show that the C-DAG is feasible, and verifies, in time polynomial in the representation of the C-DAG, whether the certificate does indeed show feasibility.[6]

The certificate for a C-DAG instance with $k$ conditional constructs will be an explicit enumeration of the at most $2^k$ individual schedules, one each for the vertices that must be executed upon each possible combination of outcomes of the execution of the $k$ conditional expressions. The number of schedules in the certificate may be fewer than $2^k$ since not all outcomes may be possible – e.g., the C-DAG depicted in Figure 7 has two conditional constructs but only 3 possible outcomes. A certificate with the three schedules is provided in Figure 7 for when this C-DAG is to be implemented on a single processor.

[6] We acknowledge that the following description of this verification algorithm is at a high level and somewhat "hand-wavy"; however we believe it is adequate for conveying the main ideas as to what information is contained in the certificate, and how the verifier checks this information.

Given such a certificate, the *verification algorithm* verifies that

1. Each schedule in the certificate is indeed a feasible schedule for the vertices that must be executed upon some possible outcome of the execution of the conditional constructs.
2. The sets of vertices that must be executed upon all possible outcomes have schedules in the certificate.
3. The schedules in the certificate are *consistent* in the following sense:
   - They are all identical (i.e., schedule the same jobs at the same instants) until the end of the first execution of a conditional expression (the diamond-shaped node marking the beginning of a conditional construct).
   - After that the set of schedules is partitioned into two subsets, one representing each of the two possible outcomes of the execution of that conditional expression.
   - Each of these two subsets must satisfy the two properties above: all schedules in the subset are identical up to the next execution of a conditional expression, and split into two sets representing the schedules for the two different outcomes thereafter.
   - This repeats until each set contains a single schedule.

This establishes that C-DAG feasibility analysis is in NP, and can therefore be reduced in polynomial time to the NP-complete problem ILP. We are currently working on developing such a polynomial-time algorithm: although the main ideas are fairly straightforward – in essence, use integer decision variables to specify the different schedules in the certificate and write constraints to enforce the requirements listed above as being checked by the verification algorithm, there are a lot of rather tedious details that must be enumerated.

The number of variables and the number of constraints in the ILP depend upon the number of schedules in the certificate. Notice the relationship between the number of conditional constructs $k$ and the number of schedules in the certificate (at most $2^k$) – this suggests that ILPs with fewer conditional constructs are likely to be representable using smaller ILPs.

## 6 Context and Conclusions

Real-time scheduling theory has begun considering the use of ILP solvers to obtain efficient algorithms for solving feasibility analysis problems. Several schedulability analysis problems have recently been solved by representing them as ILPs (e.g.,[8, 3]); here we have shown that an important problem *cannot* be solved efficiently in this manner (under the widely-held assumption that NP $\subsetneq$ PSPACE). We note some additional implications of our main technical results.

1. Observe that the workload model for heterogeneous multiprocessor platforms is unchanged from the one for identical multiprocessors for typed systems (those in which all vertices are pre-assigned to individual processors). Therefore *our results for typed systems also hold for heterogeneous multiprocessors*. Many are also applicable to the recently proposed more general Heterogeneous Parallel Conditional (HPC) DAG model [22].
2. Most solvers that are used in system design (including SAT solvers, many SMT [1] solvers, etc.) actually solve problems that are in NP.[7] Hence our main negative conclusion holds for all these solvers as well: they are unlikely to be helpful for C-DAG feasibility analysis.

[7] One important reason for this is that the results returned by such solvers can be *verified* efficiently, in polynomial time. Solutions obtained by using solvers that solve problems not in NP must either be accepted “on faith”, or inordinate amounts of time are required to validate their correctness.

3. In this work we have required that problems be reducible to ILPs in polynomial time in order to be considered tractable. As an alternative, we could have instead required that there be a polynomial-*sized* ILP representation. However, this alternative definition is unsatisfactory: one could conceivably determine feasibility for any instance of a problem via exhaustive enumeration by taking inordinate amounts of time, and then represent its feasibility as a simple ILP of just one or two variables and constraints which has a solution if and only if the instance is feasible. Hence, one could argue that just about any feasibility-analysis problem can be represented by a small ILP: the true measure of tractability is how rapidly such an ILP can be obtained.

**Some Related Work.** ILP solvers have previously been used in real-time system design and analysis – see, e.g., [16, 20]. But in the real-time scheduling theory community, where the focus has primarily been on obtaining efficient algorithms with polynomial or pseudo-polynomial running times, ILP-based techniques have traditionally not found much favor for obvious reasons. The recent dramatic improvements in performance of modern solvers mentioned in Section 3 is starting to change this, and the real-time scheduling theory community has begun to investigate the use of ILP-based methods [5, 8, 3, 9].

**Future work.** We have established a conceptual and technical framework for both showing problems to not be efficiently solvable using ILP solvers, and for identifying restricted versions that are so solvable. We plan to apply our framework to better demarcate the boundary between what if efficiently solvable and what is not with ILP solvers, as well as extend the framework to answer additional questions of interest. For a start, we plan to investigate notions of *approximability* – we could, e.g., seek sufficient ILP-based feasibility-analysis algorithms of the following kind: *given an instance generate, in polynomial time, an ILP such that (i) if it is feasible, then the instance is feasible upon unit-speed processors; and (ii) if it is infeasible, then the instance is not feasible on speed-s processors (for some $s \leq 1$).*

With regards to C-DAG feasibility, we have identified one specific structural property – restrict the number of conditional constructs – that enables efficient solution via ILP's. The reason such instances are efficiently solved is that certificates attesting to their feasibility contain relatively few schedules. We are currently identifying other such structural properties of C-DAGs that also possess this property (of having "small" certificates of feasibility).

## References

1 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825.

2 Sanjoy Baruah. Feasibility Analysis of Conditional DAG Tasks is co-NP$^{\text{NP}}$-Hard (Why This Matters). In *Proceedings of the Twenty-Ninth International Conference on Real-Time and Network Systems*, RTNS '21, New York, NY, USA, 2020. ACM.

3 Sanjoy Baruah. Scheduling DAGs when processor assignments are specified. In *Proceedings of the Twenty-Fifth International Conference on Real-Time and Network Systems*, RTNS '20, New York, NY, USA, 2020. ACM.

4 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems.* Springer Publishing Company, Incorporated, 2015.

5 Sanjoy Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. Ilp-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors.

In *Proceedings of the 2016 28th EuroMicro Conference on Real-Time Systems*, ECRTS '16, Toulouse (France), 2016. IEEE Computer Society Press.

6 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.

7 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.

8 Sanjoy K. Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling*, December 2018. doi:10.1007/s10951-018-0593-x.

9 Slim Ben-Amor. *Multicore Scheduling of Dependent Tasks with Probabilistic Execution Times*. PhD thesis, Sorbonne Université, 2021.

10 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

11 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.

12 Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation*. A K Peters, 2009.

13 Klaus Jansen. Analysis of scheduling problems with typed task systems. *Discrete Applied Mathematics*, 52(3):223–232, 1994.

14 R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

15 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.

16 Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, 1997. doi:10.1109/43.664229.

17 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.

18 L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

19 J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

20 Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In *Verification, Model Checking, and Abstract Interpretation*, pages 309–322, 2004. doi:10.1007/978-3-540-24622-0_25.

21 C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1976.

22 Houssam-Eddine Zahaf, Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Giuseppe Lipari. The HPC-DAG task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, pages 1–1, 2020.

# Scheduling Replica Voting in Fixed-Priority Real-Time Systems

**Pietro Fara** ✉ 🏠
Scuola Superiore Sant'Anna, Pisa, Italy

**Gabriele Serra** ✉ 🏠
Scuola Superiore Sant'Anna, Pisa, Italy

**Alessandro Biondi** ✉ 🏠
Scuola Superiore Sant'Anna, Pisa,Italy

**Ciro Donnarumma** ✉
Rete Ferroviaria Italiana S.P.A., Rome, Italy
Scuola Superiore Sant'Anna, Pisa, Italy

## Abstract

Reliability and safety are mandatory requirements for safety-critical embedded systems. The design of a fault-tolerant system is required in many fields (e.g., railway, automotive, avionics) and redundancy helps in achieving this goal. Redundant systems typically leverage voting techniques applied to the outputs produced by tasks to detect and even tolerate failures.

This paper studies the integration of distributed voting protocols in fixed-priority real-time systems from a scheduling perspective. It analyzes two scheduling strategies for implementing voting. One is attractive and friendly for software developers and based on suspending the task execution until the replica provides the data to be voted. The other one is inspired by the Logical Execution Time (LET) paradigm and requires introducing additional tasks in the system to accomplish voting-related activities. Queuing and delays introduced by inter-replica communication interfaces are also analyzed.

Experimental results are finally presented to compare the two strategies, showing that LET-inspired voting is much more predictable and hence more suitable than the other strategy for fixed-priority real-time systems.

**2012 ACM Subject Classification** Computer systems organization → Dependable and fault-tolerant systems and networks; Computer systems organization → Real-time systems

**Keywords and phrases** Real-time systems, safety-critical systems, voting, redundancy, fault-tolerance, logical execution time

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.13

## 1 Introduction

Embedded computing systems have become more and more pervasive in our lives: they are used to fulfill evermore functions, a lot of which are related to the safety of people and the surrounding environment. Indeed, embedded systems are nowadays widely present in avionic, railway, automotive, and military applications in a way that their failures could lead to catastrophic consequences. As these systems are related to our safety, they are commonly called safety-critical embedded systems.

In most application domains there exist a lot of regulations to which a safety-critical system must comply [11, 12]. Such regulations mandate the use of certain techniques to improve the *reliability* and the *safety* of a system. These techniques can be mainly classified into two categories: *fault avoidance* (also known as fault intolerance) and *fault tolerance*. Fault avoidance techniques aim at drastically reducing by design the probability of failure. This approach is generally not viable for complex systems because, even by performing a

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 13; pp. 13:1–13:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

meticulous design, it could be impossible to eliminate all the internal sources of faults, so that the system will eventually experience a failure. On the other hand, *fault tolerance* techniques aim at making a system capable of properly react to faults, avoiding that they lead to a failure of the functionality offered by the system [31]. Redundancy is a widespread approach to build fault-tolerant systems. Redundant systems are built by several subsystems, called *replicas*, that perform the same computations over time. The replication allows the detection and/or the masking of a fault through the *voting* (i.e., comparison) of the results computed by all replicas. A redundant architecture is said to be *r-out-of-n* (with $r \leq n$) if it is built by $n$ replicas, $r$ of which have to properly work to make the whole system failure-free [30].

The *2-out-of-2* architecture is the most used redundant architecture in the railway domain. It is an architecture that enables the detection of faults: if the results provided by the two replicas are different, a fault is detected and the system goes into a fail-safe state (e.g., shutdown). As described by Shenghua and Li [29], in the railway domain, the *2-out-of-2* architecture is employed in a hierarchical framework where the entire system with two replicas is further replicated to build a *1-out-of-2* system of systems. In normal conditions, only the primary system provides the output to the external environment but, as soon as the voting in the primary system detects a fault, the primary is shut down and the secondary system (in *hot standby*) takes over. As a result, this architecture is capable of masking faults increasing system availability.

Another architecture used in many domains, such as avionics, is the *2-out-of-3* (also known as Triple Modular Redundancy) [17, 32]. It implements *majority-voting* and is known to be capable of providing both fault-detection and fault-masking [31]: if one replica is faulty (i.e., its output is different by the other two), a fault will be detected and the system output will continue to rely on the outputs of the other two replicas.

Even though several other redundancy schemes have been proposed, most of them are based on the same idea: replicated systems perform the same computations on the same inputs, sending through a communication network their results to be voted. Voting can be either centralized or distributed. In the former case, voting is implemented on a centralized node that collects and votes all the results provided by the replicated subsystems. In this case, the voter itself is clearly a single-point-of-failure. In the latter case, each replica has its voter, either implemented with a hardware component or with a software algorithm, and votes its data against the one produced by the other replicas.

In this work, we focus on distributed voting implemented with software techniques, which is a more and more widespread approach (e.g., in the railway domain) to achieve flexibility and contain cost in realizing fault-tolerant systems.

The implementation of distributed voting requires dealing with the transmission of data among replicas, the waiting and synchronization among replicas, and the execution of the voting protocol itself. These aspects clearly impact on the timing properties of real-time tasks and call for the investigation of different strategies to suitably schedule all voting-related activities.

## 1.1 This work

Informed by experience in safety-critical software for the railway industry, in this work we analyze and compare two different strategies for scheduling voting-related activities under 2-out-of-2 redundancy. The first one corresponds to a case that is particularly attractive and friendly for software developers: data is transmitted among replicas whenever they are produced by the tasks and each task waits for the reception of the data sent by the other replica by suspending its execution (e.g., by using a classical condition variable). When tasks are resumed, the data to be voted is available and they can proceed with the execution of the voting protocol and then complete it. The second one is a new approach proposed in this work inspired by the Logical Execution Time (LET) [19, 28] paradigm where voting is delayed at the end of the tasks' periods and delegated to dedicated tasks. Although the first approach may be preferable by software developers, this work shows that it introduces several sources of unpredictability that make it particularly challenging to be analyzed from a worst-case perspective.

In summary, this work makes the following contributions:

- It provides a response-time analysis for real-time tasks under two strategies for scheduling voting-related activities, one of the two being novel and proposed in this work.
- It provides an analysis of queuing effects and worst-case transmission delays introduced during inter-replica communications.
- It compares the two strategies by means of an experimental evaluation.

To the best of our records, this is the first work that analyzes in detail the timing properties of distributed voting protocols implemented upon a fixed-priority real-time system with periodic multitasking. Software engineers from the railway industry collaborated in this work. Since this work only addresses how voting operations are scheduled, other aspects such as fault detection and recovery strategies are not discussed as they depend on the target application and the adopted voting protocol.

**Paper structure.** The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the system model by considering both tasks and inter-replica communication. Section 4 formalizes the behavior of the two voting strategies. Section 5 analyzes queuing effects and delays in inter-replica communications. Section 6 provides response-time analysis under the two voting strategies. Section 7 presents the experimental results and Section 8 concludes the paper.

## 2 Related Work

Several works in the literature studied fault-tolerant systems from both a hardware and software perspective.

Davies et al. [14] proposed a hardware-level solution, called *Synchronization Voting*, for achieving inter-replica synchronization in a redundant system, overcoming the need for a common external clock, which is a source of common-mode failures. Their approach consists of using a set of synchronizer modules (one for each replica) that, by exchanging mutual feedback, allow replicas to correct for their inevitable drift. McConnel et al. [27] continued this work by presenting voter designs for different signaling conventions (transition, level, and pulse). These papers present elegant solutions to implement inter-replica synchronization and voting at the hardware-level, but they do not consider the effects of multitasking on the replicated systems, where the voting has to be implemented on the outputs produced by tasks. Eris et al. [16] focused on railway systems (with 2-out-of-2 redundancy) with diverse programming. Their approach allows the voter to move the system from a safe state toward a less safe state only when all replicas agree. They also analyzed the effects of the synchronization issues (i.e., race conditions) on the railway signaling protocols by proposing a solution based on a centralized voter acting as a replica coordinator. Again, multitasking has not been considered.

Some real-time scheduling strategies, aimed at improving the system resilience against transient faults, have been proposed by Kim and Shin [21], and Kwak and Kim [24]. They are based on executing different copies of the same task at different times so that the probability

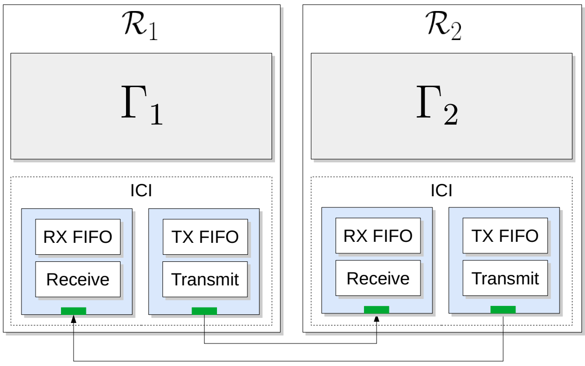


**Figure 1** An overview of the system architecture.

that a common transient fault affects all of them is reduced. Back et al. [1] proposed TL-NMR, a task-level *N Modular Redundancy* schema, which allows the execution of several copies of tasks in parallel upon multiprocessor platforms scheduled by Global Fixed-Priority. The authors provided an algorithm that allows selecting the number of copies for each task along with a schedulability test based on the response-time analysis. However, these papers, focused only on the schedulability of the tasks' copies, without considering issues related to inter-replica synchronization and the impact on the scheduling of voting protocols. Another work that improves the fault-tolerance in the presence of environmentally-induced faults is due to Gujarati et al. [18]. The authors proposed an algorithm, along with a suspension-free model of its real-time implementation (based on the Liu and Layland task model), that allows a distributed real-time system to solve the *Interactive Consistency* problem in the presence of Byzantine faults. The authors also provided a detailed real-time-aware reliability analysis of the proposed solution.

Bernat et al. [4, 5] presented a real-time fault-tolerant architecture capable of handling transient overload conditions through the firm real-time task model. The proposed architecture comprises multiple replicated subsystems, each executing a copy of the same task set, and a dedicated processor for the voting called *Redundancy eXecutive* (RX). Whenever a task finishes its execution, it sends the computed results to the RX and suspends its execution. As soon as the RX has collected enough replicas' results (some replicas could be failed), executes the voting protocol and sends the voted output back to the tasks' copies, allowing them to resume their computation. Similar to our work, the authors also provided a detailed schedulability analysis based on the response-time analysis. They consider every contribution to the tasks' execution time, such as the communication time spent into the results exchanging and executing the voting algorithm on the RX subsystem. These works have several limitations. First, they consider one scheduling scheme for voting only. Second, they rely on a dedicated subsystem to execute the voting algorithm, introducing a higher system cost and requiring to deal with the RX subsystem's potential faults. Third, they do not provide any experimental evaluation.

From the perspective of voting protocols, researchers consolidated several algorithms such as the ones presented in [2, 3, 7, 8, 14, 25, 33].

## 3 System model

This work considers a 2-out-of-2 redundant system with two replicas $\mathcal{R}_1$ and $\mathcal{R}_2$. Each replica $\mathcal{R}_k$ consists in a uni-processor platform that executes a set $\Gamma_k = \{\tau_1^k, \dots, \tau_n^k\}$ of $n$ periodic tasks. Each periodic task $\tau_i^k$ is characterized by a worst-case execution time (WCET) $C_i^k$, a release period $T_i^k$, and a relative deadline $D_i^k \leq T_i^k$. Tasks are scheduled according to fixed-priority preemptive scheduling. The set of higher-priority tasks with respect to $\tau_i^k$ that execute on the same replica $\mathcal{R}_k$ is denoted by $hp(i,k)$.

Each periodic task $\tau_i^1$ running in the primary is associated with a corresponding periodic task $\tau_i^2$ running in the secondary and the two tasks form a *replica pair* $r_i = \{\tau_i^1, \tau_i^2\}$. The tasks in a replica pair share the same period and deadline, i.e., $T_i^1 = T_i^2$ and $D_i^1 = D_i^2, \forall i = 1, \dots, n$. Given a replica $\mathcal{R}_k$, the other replica is referred to as $\mathcal{R}_{or(k)}$, where $or(k) = (k+1) \bmod 2$. The clocks of the two replicas are *synchronized* so that the release of the periodic tasks in each replica pair is synchronized. The WCET of the tasks can be different from replica to replica, i.e., $C_i^1$ can be larger or shorter than $C_i^2$ for some pairs of tasks $\tau_i^1$ and $\tau_i^2$.

**Inter-replica communication and voting.** The two replicas are connected via two wired *inter-replica communication interfaces* (ICI): one for sending the data from $\mathcal{R}_1$ to $\mathcal{R}_2$, and one for sending data from $\mathcal{R}_2$ to $\mathcal{R}_1$. An overview of the system architecture is shown in Figure 1. For instance, the ICI can be realized with serial peripheral interfaces (SPI) for transmitting data and digital lines connected to general-purpose input/output (GPIO) for the synchronization signals.

Data transmission via the ICI occurs by acting on memory-mapped device registers. The ICI provides an output (resp., input) buffer organized as a first-in-first-out (FIFO) queue of $Q$ elements, each of size $b$ bytes. The ICI also provides synchronization signals to notify events among replicas (e.g., the completion of a computation). The minimum read/write rate in accessing such registers is denoted by $\beta$ (in bytes per time unit), while the maximum one is denoted by $\overline{\beta}$. The minimum transmission rate guaranteed by the ICI is denoted by $\alpha$ (in bytes per time unit). The minimum read/write rate to access memory is $\gamma$[1]. For instance, this means that a task that intends to send $x$ bytes via one of the ICI spends **(i)** at most $x/\gamma$ time units to read the data to be sent from memory, **(ii)** at least $x/\overline{\beta}$ time units and at most $x/\beta$ time units of its computation time to fill the ICI queue with data, and **(iii)** that such data will be transmitted to the other replica in at most $x/\alpha$ time units.

Periodic tasks may produce *vital outputs*, i.e., data that are critical for the system. Both the tasks $\tau_i^1$ and $\tau_i^2$ of each replica pair produce the same set of vital outputs. Before the completion of each job, each periodic task has to *vote* its vital outputs (if any) with the corresponding task of its replica pair. Voting is implemented via a distributed voting protocol [26] that exchanges data via the ICI.

Both the tasks in a replica pair $r_i$ exchange $M_i$ data packets with a fixed size of $b$ bytes that contain the data to be voted.

Tasks that intend to send data via an ICI that has its queue full, busy-wait until at least one slot in the queue becomes empty. In reception, the ICI can either operate in *polling* mode or in *interrupt* mode. In the former case, tasks receive packets by actively sampling the ICI queue, possibly wasting processor cycles if the queue is empty. In the latter case,

[1] The authors acknowledge that memory write times are generally shorter than read times. A common rate $\gamma$ has been considered just for the sake of simplicity as it does not particularly affect the results of this paper.

the ICI notify receipt of packets through interrupts. The ICI are programmed to raise one interrupt every time a packet is received. The corresponding *interrupt service routine* (ISR) is in charge of reading the packets in the queue, by acting on the ICI device registers, and copying them into a memory buffer shared with the task interested by the packet (interrupts that are raised while the ISR is pending are ignored and the corresponding packets are processed by the same). Each ISR introduces an overhead of at most $\sigma^{\mathrm{ISR}}$ time units due to the management of the ISR activation and completion (i.e., this overhead does not include the time required to process packets).

The time that $\tau_i^k$ spends to perform computations lasts at most $E_i^k$ time units. This parameter does not account for packet transmissions and receptions and the execution of the voting protocol. The transmissions performed by each job of the tasks consist of copying the vital outputs from memory into the transmission registers of the ICI. For tasks of the replica pair $r_i$ such transmissions take at most $VT_i = M_i \cdot PT$ time units, where $PT = \left(\frac{b}{\gamma} + \frac{b}{\beta}\right)$. The maximum time needed to receive the packets of the tasks of $r_i$ and store them in a shared-memory buffer, to be later consumed by the voting protocol, is denoted by $VR_i = M_i \cdot PT$.

The utilization of the ICI, intended as the amount of bytes transmitted per time unit in the long run, is defined as $U^{\mathrm{ICI}} = \sum_{i=1}^{n}(M_i b)/T_i$. To avoid dealing with cases in which the ICI is overutilized, which clearly makes the system not feasible, we require $\alpha > U^{\mathrm{ICI}}$ and $\beta > U^{\mathrm{ICI}}$.

After the two tasks in a replica pair $r_i$ exchanged the data to be voted, a voting protocol can be executed, which takes at most $VP_i$ time units.

The data transmission is performed by using one of the ICI in a mutually-exclusive manner. To this end, each task may have to acquire and release a lock before and after transmitting each packet, respectively. The immediate priority ceiling (IPC) locking protocol is adopted. The case in which all tasks have to vote data is equivalent to a resource shared by all tasks: hence, under the IPC protocol, the critical sections to access the communication interface are equivalent to non-preemptive sections.

## 4 Voting implementations

This work is focused on analyzing and comparing two schemes to schedule the execution of the voting protocol and the related data transmissions.

The first one, named *passive waiting*, is an approach that can be implemented with a minimal impact on general-purpose programming paradigms, as it corresponds to the case in which a task sequentially performs the following three operations: (i) compute, (ii) wait for the other replica to complete by self-suspending its execution, and (iii) execute the voting protocol. Note that passive waiting can be implemented with classical semaphores and condition variables. The second one is inspired by the Logical Execution Time (LET) paradigm and requires introducing additional tasks in the system.

The following rules characterize the behavior of each of the considered scheduling schemes:

- *Transmission Rule*: it defines how data transmission is performed among replicas.
- *Reception Rule*: it defines the behavior of the replica that receives the data.
- *Waiting Rule*: it defines how a task $\tau_i^k$ has to wait for the corresponding task $\tau_i^{or(k)}$ in the other replica.
- *Voting Rule*: it defines how the voting protocol is executed.

## 4.1 Passive waiting

Under passive waiting tasks are composed of three serialized phases: **(i)** an execution phase (E), in which the task computes the data to be voted; **(ii)** a transmission phase (VT) where vital outputs are transmitted to the other replica; and **(iii)** a final phase where the voting protocol (VP) is executed. The reception of packets is handled by ISRs (the ICI are used in interrupt mode).

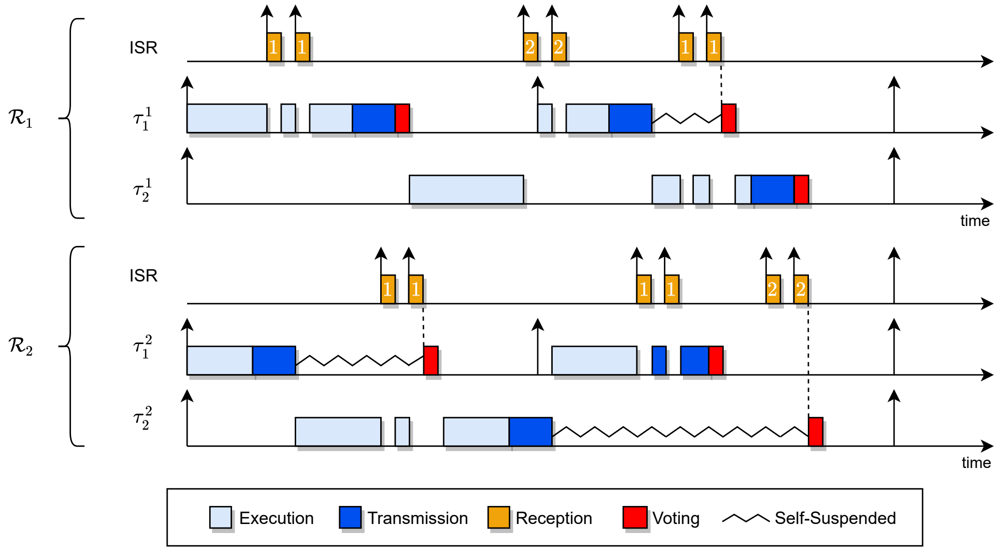


**Figure 2** Example schedule of two replica-pairs under passive waiting.

**Transmission rule.** When completing its computations, each task $\tau_i^k$ transmits $M_i$ packets of data to be voted on by the other replica. For each packet to transmit, the task first acquires the lock on the communication interface, then transmits the packet, and finally releases the lock.

**Reception rule.** Whenever a replica $\mathcal{R}_k$ receives a data packet, an ISR is executed by preempting any task in execution in $\mathcal{R}_k$, i.e., the ISRs run at the highest priority level and are not affected by the locking of the communication interface as two independent ICI are used for transmission and reception. The ISRs perform the operations specified in Section 3. For each task $\tau_i^k$, when the last of the $M_i$ packets sent by $\tau_i^{or(k)}$ (i.e., from the other replica $\mathcal{R}_{or(k)}$) is received, the ISR that handles the packet notifies $\tau_i^k$ that all its data is ready in a shared-memory buffer to be voted on.

**Waiting rule.** When a task $\tau_i^k$ completes its transmission phase it *self-suspends* its execution until all the $M_i$ packets sent by the other replica task $\tau_i^{or(k)}$ are received and processed by the corresponding ISRs. The self-suspension is skipped if all $M_i$ packets have already been received and processed by ISRs.

**Voting rule.** The voting protocol is executed after all the $M_i$ packets have been received and processed by ISRs, i.e., after the eventual self-suspension enforced by the voting rule. The task terminates after the execution of the voting protocol.

An example schedule under passive waiting is illustrated in Figure 2. In this example, two replica-pairs are needing to vote two packets each. The first job of $\tau_1^1$, according to the waiting rule, does not experience any suspension as it already received all the packets from the other replica when it becomes ready to vote. On the other hand, the first job of $\tau_1^2$ completes its execution and transmission phases before $\tau_1^1$, so it self-suspends until the delivery of the second packet. As soon as the ISR of replica $R_2$ handles the last packet, $\tau_1^2$ is awakened to execute the voting protocol. The behaviors of the second jobs of the previous tasks are dual: task $\tau_1^2$ completes without any suspension, instead, $\tau_1^1$ self-suspends to wait for the other replica. Note that, at its release time, the second job of $\tau_1^2$ is blocked by $\tau_2^2$ because the latter acquires the lock on the ICI to transmit a packet.

Note that with this approach the ICI queues may contain packets of different tasks at the same time. Indeed, some task $\tau_i^k$ can start sending packets and then be preempted by another task $\tau_j^k$ that sends its packets, and so on. As such, packets must contain the identifier of the sender task to be correctly dispatched by ISRs in the other replica.

## 4.2 LET-inspired voting

The underlying idea of this scheduling scheme is to get rid of both the waiting times and the any-time data transmission of the preceding scheme by confining all voting-related activities in predefined time intervals.

Together with the task set $\Gamma_k$, each replica $\mathcal{R}_k$ serves the execution of a set $\Upsilon_k = \{\upsilon_1^k, \dots, \upsilon_n^k\}$ of *voting tasks*, one for each task $\tau_i^k$, each of them executing at the same priority equal to a value higher than the priority of any task in $\Gamma_k$. Voting tasks are executed with the same period of the corresponding (regular) task, i.e., $T_i^{k,V} = T_i^k, \forall i, \forall k$. Tasks communicate with their corresponding voting tasks via shared-memory buffers. A task completes as soon as it finishes its computations, leaving the data to be voted in a shared-memory buffer. Then, the voting-related activities are delegated to the corresponding voting task $\upsilon_i^k$, which is synchronously activated with $\tau_i^k$. Note that, being $\upsilon_i^k$ executed at a higher priority than $\tau_i^k$, it always executes before $\tau_i^k$. As such, each $j$-th job of the voting task $\upsilon_i^k$ accomplishes the voting-related activities for the *preceding job*, i.e., the $(j-1)$-th one, of $\tau_i^k$. Voting tasks are synchronously-released among replicas and executed in the same order on both replicas (voting tasks are selected according to their identifier whenever they are simultaneously pending). The execution of the voting tasks is also synchronized among replicas, meaning that a rendez-vous point is provided at their completion so that each voting task $\upsilon_i^k$ finishes together to $\upsilon_i^{or(k)}$. The latter synchronization is implemented by means of the synchronization signals offered by the ICI.

Voting tasks access the ICI in polling mode (no ICI-related ISRs are present under this voting scheme). The voting tasks perform the transmission and reception of packets in inverse order on the two replicas, as stated by the following rules.

**Transmission rule.** After completing their computations, the tasks terminate their execution by leaving the packets to be transmitted in memory buffers shared with their corresponding voting tasks. The transmission is then delegated to the voting tasks. On replica $\mathcal{R}_1$, the voting task $\upsilon_i^1$ of $\tau_i^1$ transmits $M_i$ packets to $\mathcal{R}_2$ as soon as it is activated. On replica $\mathcal{R}_2$, the voting task $\upsilon_i^2$ of $\tau_i^2$ transmits $M_i$ packets to $\mathcal{R}_1$ after it received the packets sent by $\upsilon_i^1$.

**Receiving rule.** On replica $\mathcal{R}_1$, the voting task $\upsilon_i^1$ of $\tau_i^1$ receives (in polling mode) $M_i$ packets sent from $\mathcal{R}_2$ after it transmitted its packets. On replica $\mathcal{R}_2$, the voting task $\upsilon_i^2$ of $\tau_i^2$ receives (in polling mode) $M_i$ packets from $\mathcal{R}_1$ as soon as it is activated.

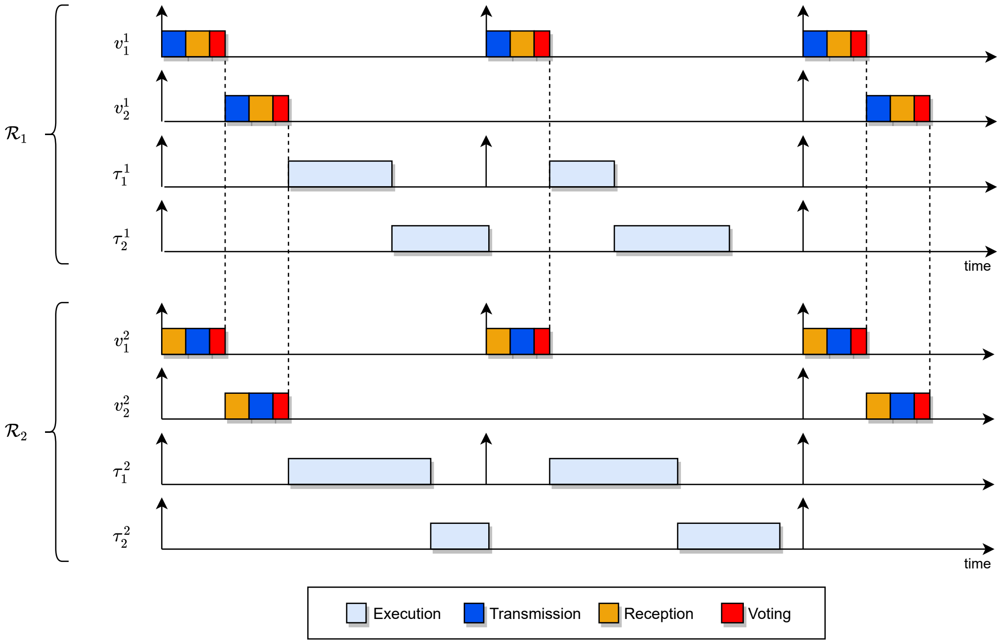


**Figure 3** Example schedule under LET-inspired voting for a system with two replica pairs.

**Waiting rule.** None: when a task has finished its execution phase it terminates.

**Voting rule.** The voting protocol is executed by voting tasks after they completed both the packet transmission and reception. When the voting protocol terminates, the voting tasks busy waits until the corresponding voting task on the other replica sends a signal through the ICI synchronization line to notify the completion of the voting protocol.

The behavior of this LET-inspired scheduling scheme for voting is illustrated in Figure 3. Note that, since voting tasks are synchronously released together with their corresponding regular tasks and have a higher priority, voting is guaranteed to occur before starting executing the next job of regular tasks. In this way, voting is still *logically* occurring in the temporal context given by the period of the tasks that generate the data to be voted.

## 5 Inter-replica communication

This section deals with the analysis of inter-replica communications employing the ICI. Two problems are addressed. First, since under passive waiting packets can be sent at any time and that the communication is asynchronous (the ICI works in interrupt mode), packets of different tasks can be enqueued together in the ICI queues. This makes the worst-case transmission delay experienced by the packets of a certain task particularly challenging to be bounded, especially if considering the additional delays introduced by the waiting for the emptying of the queue. For this reason, we derive an analysis to ensure that the ICI queues are never full, hence getting rid of these additional delays by construction. Subsequently, we also provide a bound on the maximum delay introduced by the ICI.

### 5.1 Queuing analysis

We begin by bounding the amount of data sent within arbitrary time windows.

▶ **Lemma 1.** *In any time window of length $t$, the tasks can provide in the ICI queue at most $g(t)$ bytes of data, where*

$$g(t) = \min \left\{ \sum_{i=1}^{n} \left\lceil \frac{t+T_i}{T_i} \right\rceil M_i b,\ \overline{\beta} t \right\}. \tag{1}$$

**Proof.** In any time window of length $t$ a periodic task in $r_i$ can release at most $\lceil (t+T_i)/T_i \rceil$ jobs (e.g., see [9], Ch. 5). Each job of the tasks in $r_i$ sends at most $M_i$ packets, each of size $b$ bytes. Hence the first term in the minimum of Eq. (1). Note that the amount of data the tasks can send within a time window is also limited by the maximum rate with which the ICI queue can be filled, which is given by $\overline{\beta}$. Hence the lemma follows. ◀

The above lemma can then be used to derive a safe condition under which the ICI queues are never full.

▶ **Lemma 2.** *No task can find the ICI queues full if*

$$\forall t > 0, \quad g(t) - \alpha t \leq Qb. \tag{2}$$

**Proof.** Assume by contradiction that at a certain time instant $t_1$ a task finds an ICI queue full. Let $t_0 < t_1$ be the latest time at which the ICI queue has been empty and let $t = t_1 - t_0$. It holds that $(t_0, t_1]$ is an interval of length $t$ in which the ICI has always been busy with packets to transmit to the other replica. Let $x(t)$ be the amount of bytes issued by the tasks to be provided in the ICI queue in $(t_0, t_1]$. Note that during this interval the ICI must have sent at least $\alpha t$ bytes: hence, if the queue is full at time $t_1$ it holds that $x(t) - \alpha t > Qb$.

By Lemma 1, in any time window of length $t$ the cumulative amount of bytes provided in the ICI queue is bounded by $g(t)$. Hence, $g(t) \geq x(t)$, which implies $g(t) - \alpha t > Qb$. This contradicts Eq. (2). Hence the lemma follows. ◀

Note that Lemma 2 does not consist in a practical test as any possible value of $t$ shall be checked. This issue is solved below by limiting the test to a finite number of check-points.

▶ **Lemma 3.** *Lemma 2 holds also if* $\forall t \in \Phi, \quad g(t) - \alpha t \leq Qb$, *where*

$$\Phi = \bigcup_{i=1}^{n} \{kT_i + \epsilon \leq t^*, k = 0, 1, 2, \ldots\} \cup \{\psi\} \tag{3}$$

*with*

$$t^* = \frac{2\sum_{i=1}^{n} M_i b}{\alpha - \sum_{i=1}^{n} \frac{M_i b}{T_i}}, \quad \psi = \left\{ t \leq t^* \mid \sum_{i=1}^{n} \left\lceil \frac{t+T_i}{T_i} \right\rceil M_i b = \overline{\beta} t \right\}, \tag{4}$$

*and $\epsilon > 0$ arbitrarily small.*

**Proof.** We prove the lemma by showing that function $g(t) - \alpha t$ can be maximal only for values $t \in \Phi$. First note that the minimum of two functions is upper bounded by the upper bound of one of the two functions. Hence $g(t) \leq G(t) = \sum_{i=1}^{n} \left( \frac{t+T_i}{T_i} + 1 \right) M_i b$.

Note that both $G(t)$ and $\alpha t$ are two lines with slope $U^{\text{ICI}} = \sum_{i=1}^{n} (M_i b)/T_i$ and $\alpha$, respectively. Recall that $\alpha > U^{\text{ICI}}$ (see Section 3). Therefore $G(t)$ and $\alpha t$ intersect and, from their intersection on, we have $g(t) \leq G(t) \leq \alpha t$ and hence also $g(t) - \alpha t \leq 0$.

The intersection occurs for the value $t^*$ such that $G(t^*) = \alpha t^*$ and can be computed by solving the latter equality with respect to $t^*$, hence getting the expression at the left of Eq. (4). Therefore, for values of $t > t^*$ function $g(t) - \alpha t$ cannot be maximal.

If $g(t) = \sum_{i=1}^{n} \left\lceil \frac{t+T_i}{T_i} \right\rceil M_i b$ note that function $g(t) - \alpha t$ can be maximal only for those values of $t$ that correspond to a step of the ceiling term of $g(t)$. The values are of the form $t = kT_i + \epsilon$ with $k$ being a non-negative integer and $\epsilon > 0$ arbitrarily small. Conversely, if $g(t) = \overline{\beta} t$, being both the latter function and $\alpha t$ monotonic increasing, function $g(t) - \alpha t$ can be maximal only for those values of $t$ for which at $t' = t + \epsilon$ (when $\alpha \leq \overline{\beta}$) or $t' = t - \epsilon$ (when $\alpha > \overline{\beta}$), with $\epsilon > 0$ arbitrarily small, it holds $g(t') \neq \overline{\beta} t$. These values of $t$ must be an intersection between the two components that define $g(t)$, which are those of the set $\psi$ at the right of Eq. (4). Hence the lemma follows. ◀

## 5.2 Delay analysis

▶ **Definition 4.** *The ICI-related delay $\Delta^{ICI}$ is an upper bound on the maximum time that can elapse from the time a packet is stored in the ICI queue by the sender task to the time the packet is available to be read from the ICI queue at the receiver.*

In the following the ICI-related delay is studied with queuing theory for networks [6] [23]. Under this approach, the ICI-related delay is decomposed as

$$\Delta^{\text{ICI}} = d_{\text{prop}} + d_{\text{trans}} + d_{\text{proc}} + d_{\text{queue}},$$

where $d_{\text{prop}}$ is the propagation delay, $d_{\text{trans}}$ is the transmission delay, $d_{\text{proc}}$ is the processing delay at the receiver, and $d_{\text{queue}}$ is the queuing delay. We proceed by individually bounding the above delay components.

**Propagation delay.** This delay corresponds to the physical propagation of the data along the wires that connect the two replicas. Clearly, it depends on both the technology used to realize the ICI and the wire length, as well as other physical properties such as the wire material. For instance, a typical SPI has a propagation delay of 5 $ns/m$ [22], which is hence mostly negligible in an integrated system with short wiring. Hence $d_{\text{prop}} \approx 0$.

**Transmission delay.** This delay is simply bounded by the minimum guaranteed transmission rate $\alpha$ of the ICI as $d_{\text{trans}} \leq b/\alpha$.

**Processing delay.** This delay corresponds to the time taken by the ICI peripheral to make a packet available to be read from the ICI queue after it has been received. For instance, for SPI it is typically in the order of a very few microseconds (e.g., see [34]) and is hence mostly negligible. Thus $d_{\text{proc}} \approx 0$.

**Queuing delay.** This delay corresponds to the maximum time some data can remain in the ICI queues before being actually transmitted. In order to bound this delay component, the maximum number of packets that can be enqueued in the ICI queues at any time must be bounded first.

▶ **Lemma 5.** *The ICI queues never contain more than $Q^{MAX}$ packets, where*

$$Q^{MAX} = \max_{t \in \Phi} \left\{ \left\lceil \frac{g(t) - \alpha t}{b} \right\rceil \right\} \tag{5}$$

*and $\Phi$ is defined as in Lemma 3.*

**Proof.** Assume by contradiction that at a certain time instant $t_1$ there are more than $Q^{\mathrm{MAX}}$ packets in an ICI queue. Let $t_0 < t_1$ be the latest time at which the ICI queue has been empty and let $t = t_1 - t_0$. Similarly as argued in the proof of Lemma 2 this implies $g(t) - \alpha t > Q^{\mathrm{MAX}} b$, which in turn also implies $\left\lceil \frac{g(t)-\alpha t}{b} \right\rceil > Q^{\mathrm{MAX}}$. By Lemma 3, function $g(t) - \alpha t$ can be maximal only for values $t \in \Phi$, hence Eq. (5) also gives the maximal value of $\left\lceil \frac{g(t)-\alpha t}{b} \right\rceil$ that must be both equal and larger to $Q^{\mathrm{MAX}}$. This is a contradiction. The lemma follows. ◀

The maximum time a packet can be delayed while being in the queue is guaranteed not to be larger than the cumulative transmission time of all the preceding packets in the queue, which can be at most $Q^{\mathrm{MAX}} - 1$. Hence

$$d_{\mathrm{queue}} \leq (Q^{\mathrm{MAX}} - 1) \cdot b/\alpha. \tag{6}$$

## 6 Response-time analysis

This section focuses on bounding the worst-case response time of tasks under both passive waiting and LET-inspired voting.

### 6.1 Passive waiting

Following Section 4, besides its regular execution, which lasts at most $E_i^k$ time units, each task also executes the transfer of the data to be voted into the ICI registers and the voting protocol, which last at most $VT_i$ and $VP_i$ time units, respectively, on both replicas. Hence, the cumulative WCET of task $\tau_i^k$ is given by

$$C_i^k = E_i^k + VT_i + VP_i. \tag{7}$$

The analysis of tasks under passive waiting is split into two parts. First we bound the *partial response time* of a task, which is defined as the response time up to the copy into the ICI registers of the data to be voted, i.e., just before the start for the waiting of the other replica. Subsequently, the response time of the whole task is bounded as a function of the partial response time.

Up to the partial response time, task $\tau_i^k$ can be delayed by (i) its own execution and the transfer of the data to be voted into the ICI registers, which can last at most $E_i^k + VT_i$ time units, (ii) the interference generated by high-priority tasks, (iii) the blocking time generated by low-priority tasks, and (iv) the interference generated by the ISRs (which run at higher priorities). We proceed by bounding these components individually.

Note that, under passive waiting, tasks behave as self-suspending tasks [13]. As such, high-priority interference can be bounded utilizing a state-of-the-art result provided that the WCET bound of Equation (7) is used.

▶ **Lemma 6.** *Under passive waiting, the high-priority interference generated to a job of task $\tau_i$ by high-priority tasks in any interval of length $t$ is bounded by*

$$I_i^{k,hp}(t) = \sum_{\tau_j^k \in hp(i,k)} \left\lceil \frac{t + \overline{R_j^k} - C_j^k}{T_j} \right\rceil \cdot C_j^k,$$

*where $\overline{R_j^k}$ is an upper bound on the response time of $\tau_j^k$.*

**Proof.** Follows by [13] (Theorem 1). ◀

Now, we proceed by bounding the non-preemptive blocking generated by low-priority tasks because of the locking of the ICI.

▶ **Lemma 7.** *Under passive waiting, a job of task $\tau_i$ can be blocked at most twice, one before its partial response time and one after, and each time by at most $PT$ time units.*

**Proof.** Due to the transmission rule under passive waiting (Sec. 4.1), a task can lock one of the ICI to transmit a packet, hence entering a non-preemptive section that can delay a higher-priority task. As the lock is released after the packet is stored in the ICI registers, the non-preemptive section can last at most $PT$ time units. Tasks can be prevented from execution due to non-preemptive blocking (i) at their release, and (ii) when resuming their execution after self-suspensions, which occurs after their partial response time. Case (ii) can happen only once as tasks suspend once to wait for the completion of the replica task. Hence the lemma follows. ◀

▶ **Lemma 8.** *Let $\overline{PR}_j^k$ be an upper bound on the partial response time of task $\tau_j^k$. Under passive waiting, the interference generated to a job of task $\tau_i^k$, in any interval of length $t$, by ISRs that handle packets for $\tau_j^k$ is bounded by*

$$I_{i,j}^{k,ISR}(t) = \left\lceil \frac{t + \overline{PR}_j^{or(k)} + \Delta^{ICI}}{T_j} \right\rceil \cdot M_j \cdot (\sigma^{ISR} + PT).$$

**Proof.** Consider an arbitrary time interval $[0, t]$ and a replica $\mathcal{R}_k$. ISRs are activated by packets sent by jobs of tasks running in the other replica $\mathcal{R}_{or(k)}$. Each job of task $\tau_j^{or(k)}$ in $\mathcal{R}_{or(k)}$ can send at most $M_j$ packets, each requiring $PT$ time units to be read by ISRs in $\mathcal{R}_k$. Each job of task $\tau_j^{or(k)}$ can also activate at most $M_j$ ISRs in $\mathcal{R}_k$, one per packet sent, each introducing an overhead of at most $\sigma^{\text{ISR}}$ time units. Overall, the total ISR-related workload generated by a job $\tau_j^{or(k)}$ is bounded by $M_j \cdot (\sigma^{\text{ISR}} + PT)$.

Now, note that tasks can send packets only before the occurrence of their partial response time. Hence, a job of task $\tau_j^{or(k)}$ released before time $-(\overline{PR}_j^k + \Delta^{\text{ICI}})$ cannot activate an ISR in $\mathcal{R}_k$ during $[0, t]$ as its packets would have already been sent and transmitted before the beginning of the interval. Hence, only jobs of $\tau_j^{or(k)}$ released in interval $[-(\overline{PR}_j^k + \Delta^{\text{ICI}}), t]$ may activate ISRs in $[0, t]$. This means that there are most $\left\lceil \frac{t+\overline{PR}_j^{or(k)}+\Delta^{\text{ICI}}}{T_j} \right\rceil$ jobs of $\tau_j^{or(k)}$ that can activate ISRs in $\mathcal{R}_k$ during $[0, t]$. Hence the lemma follows. ◀

Bounds on contributions (i)-(iv) mentioned above are hence now available. Following classical response-time analysis, a bound on the worst-case partial response time $PR_i^k$ of each task $\tau_i^k$ can then be computed as the least positive fixed point of the recurrence:

$$PR_i^k = E_i^k + VT_i + I_i^{k,\text{hp}}(PR_i^k) + PT + \sum_{j=1}^{n} I_{i,j}^{k,\text{ISR}}(PR_i^k). \tag{8}$$

Note that Equation (8) uses the interference bound of Lemma 8, which in turn requires the knowledge of an upper bound on the partial response time $\overline{PR}_j^k$ that is to be computed by Equation (8), hence introducing a circular dependency. This issue can be solved with a typical refinement algorithm for response-time bounds starting from a safe value (e.g., see [10]), such as the task deadline.

It is now possible to bound the total response time of the tasks by bounding the worst-case response time of the execution of the voting protocol.

▶ **Lemma 9.** *After at most*

$$J_i^k = \max\{PR_i^k, PR_i^{or(k)}\} + \Delta^{ICI} + Q^{MAX} \cdot (\sigma^{ISR} + PT) \tag{9}$$

*time units from the task release, the voting protocol of task $\tau_i^k$ is ready to start executing.*

**Proof.** Given the task behavior under passive waiting specified in Section 4.1, the voting protocol of task $\tau_i^k$ can start executing only after that (i) all its $M_i$ packets have been sent to the other replica, and (ii) all packets sent by the replica task $\tau_i^{or(k)}$ have been received and handled by ISRs.

Let us consider times related to the release of $\tau_i^k$. At time $\max\{PR_i^k, PR_i^{or(k)}\}$ both $\tau_i^k$ and $\tau_i^{or(k)}$ have sent their packets by definition of partial response time. The last packet sent by $\tau_i^{or(k)}$ will take at most $\Delta^{\mathrm{ICI}}$ to be transmitted to $\mathcal{R}_k$. Hence, at time $\max\{PR_i^k, PR_i^{or(k)}\} + \Delta^{\mathrm{ICI}}$ all packets sent by $\tau_i^{or(k)}$ must already have been received by $\mathcal{R}_k$. When the last of such packets is received it may still be the case that there are some other packets ahead in the ICI queue to be processed: by Lemma 5, they can be at most $Q^{\mathrm{MAX}} - 1$ and each of them can take at most $(\sigma^{\mathrm{ISR}} + PT)$ time units to be processed as discussed in the proof of Lemma 8. At most other $(\sigma^{\mathrm{ISR}} + PT)$ time units are needed to process the last packet sent by $\tau_i^{or(k)}$. Hence the lemma follows. ◀

The above lemma allows studying the execution of the voting protocol of each task $\tau_i^k$ as a sub-task with jitter $J_i^k$ whose completion corresponds to the completion of $\tau_i^k$.

▶ **Theorem 10.** *The response time of task $\tau_i^k$ is bounded by $J_i^k + R_i^k$, where $R_i^k$ is the least positive fixed point of the following recurrence:*

$$R_i^k = VP_i + PT + I_i^{k,hp}(R_i^k) + \sum_{\substack{j=1\\ j\neq i}}^{n} I_{i,j}^{k,ISR}(R_i^k). \tag{10}$$

**Proof.** Task $\tau_i^k$ completes when the execution of the voting protocol completes. The latter lasts at most $VP_i$ time units and can be delayed by (i) non-preemptive blocking, (ii) the execution of high-priority tasks, and (iii) the execution of ISRs. By Lemma 7, non-preemptive blocking is no larger than $PT$ time units. By Lemma 6, high-priority task interference is bounded by $I_i^{k,hp}(t)$. Note that only ISRs that handle packets of other tasks $\tau_j^k \neq \tau_i^k$ can interfere with the execution of the voting protocol as the latter becomes eligible for execution only when all packets of $\tau_i^k$ have been received. Hence, by Lemma 8, the last term in Eq. (10) bounds the ISR interference.

Due to the fact that all the phenomena that can delay the execution of the voting protocol are safely bounded, by standard response-time analysis the least positive fixed point of Eq. (10) bounds the largest amount of time the execution of the voting protocol can take to complete from the time it becomes ready to execute. Therefore, after recalling Lemma 9, $J_i^k + R_i^k$ is a safe response time and the theorem follows. ◀

## 6.2 LET-inspired voting

Under the LET-inspired scheduling scheme for voting, the tasks compute their results and terminate without undertaking any voting-related activity. Therefore, the WCET of each task $\tau_i^k$ can be computed as just $C_i^k = E_i^k$.

Conversely, the voting tasks (i) receive the data produced by the other replica, (ii) transmit the data to the other replica, (iii) execute the voting protocol, and (iv) finally wait for the completion of the corresponding voting task on the other replica. As specified in

Section 4.2, voting tasks are synchronized among replicas: being synchronously released and synchronously terminated, the execution of the voting tasks running on the two replicas perfectly overlaps in time. This allows bounding the WCET of the voting tasks as follows.

▶ **Theorem 11.** *The WCET of voting task $\upsilon_i^k$ is bounded by*

$$C_i^{k,V} = 2\left(VT_i + \frac{M_i b}{\alpha} + VR_i\right) + VP_i. \tag{11}$$

**Proof.** Following the behavior specified in Section 4.2, voting tasks execute the transmission and reception of packets in different orders. We then separately study the voting tasks on the two replicas. On $\mathcal{R}_1$, $\upsilon_i^1$ first executes the transmission and then the reception. The time taken to perform these operations is due to (i) the actual copies to and from the ICI device registers and (ii) the eventual busy waiting either because the ICI queue is full during transmission or because the ICI queue is empty during the reception. Contribution (i) can be at most $VT_i + VR_i$ time units.

Since voting tasks are synchronously executed on the two replicas and the transmission and reception phases are performed in inverse orders on the two replicas, when $\upsilon_i^1$ is transmitting packets $\upsilon_i^2$ can continuously make progress in receiving them, and vice versa. Furthermore, since the completion of voting tasks is synchronized among replicas, the ICI queues are guaranteed to be empty whenever the voting tasks are activated. Hence, during the execution of $\upsilon_i^1$ and $\upsilon_i^2$ only packets related to replica pair $r_i$ can be present in the ICI queues. This means that $\upsilon_i^2$ can take at most $\frac{M_i b}{\alpha} + VR_i$ time units to receive the packets sent by $\upsilon_i^1$ and $\upsilon_i^2$ can take at most $\frac{M_i b}{\alpha} + VT_i$ time units to transmit its packets to $\upsilon_i^1$. These terms bound the corresponding waiting times experienced because the ICI queues are either full or empty. Hence, contribution (ii) is bounded by $2\frac{M_i b}{\alpha} + VR_i + VT_i$.

Finally, since the reception is performed in polling mode, when $\upsilon_i^1$ starts executing the voting protocol $\upsilon_i^2$ must already have transmitted its packets, otherwise the reception phase of $\upsilon_i^1$ would not be completed. Hence, $\upsilon_i^1$ can either execute the voting protocol for at most $VP_i$ time units or wait for the completion of just the execution of the voting protocol in $\mathcal{R}_2$, which lasts anyway at most $VP_i$ time units. Overall, $\upsilon_i^1$ can execute for at most $(VT_i + VR_i) + (2\frac{M_i b}{\alpha} + VR_i + VT_i) + VP_i$ time units, hence matching Eq. (11).

Now, let us consider $\upsilon_i^2$. For the same reasons discussed above, this task can wait at most $\frac{M_i b}{\alpha} + VT_i$ time units during the reception of packets performed at the beginning of the task. At the time $t^*$ at which $\upsilon_i^2$ received all packets it is guaranteed that $\upsilon_i^1$ has completed its transmission phase. Hence, $\upsilon_i^2$ cannot wait for more than the time $\upsilon_i^1$ can take to complete its reception phase and the execution of the voting protocol, which is bounded by $\frac{M_i b}{\alpha} + VR_i + VT_i + VP_i$ as discussed above. From $t^*$ on, $\upsilon_i^2$ can also executes its transmission phase and voting protocol for no more than $VT_i + VP_i$ time units. Hence, the total time $\upsilon_i^2$ can take from $t^*$ to its completion, either busy waiting or executing, is bounded by $\max\{\frac{M_i b}{\alpha} + VR_i + VT_i + VP_i, VT_i + VP_i\} = \frac{M_i b}{\alpha} + VR_i + VT_i + VP_i$. Hence, the total execution time of $\upsilon_i^2$ is bounded again by Eq. (11). The theorem follows. ◀

With the above lemma in place, it is now possible to bound the worst-case response time of the tasks as follows. The worst-case response time of task $\tau_i^k$ is bounded by the least positive solution of the following recurrence:

$$R_i^k = C_i^k + I_i^{k,\text{hp}}(R_i^k) + I_i^{k,\text{V}}(R_i^k),$$

where $I_i^{k,\text{hp}}(R_i^k)$ is a bound on the interference generated by high-priority tasks and $I_i^{k,\text{V}}(R_i^k)$ is a bound on the interference generated by voting tasks.

▶ **Lemma 12.** *It holds* $I_i^{k,hp}(R_i^k) = \sum_{\tau_j^k \in hp(i,k)} \left\lceil \frac{R_i^k}{T_j^k} \right\rceil \cdot C_j^k$.

**Proof.** Under LET-inspired voting, tasks $\tau_i^k$ behave as regular periodic tasks (note that no suspensions are involved). Thus, the lemma follows from standard response-time analysis for periodic tasks under preemptive fixed-priority scheduling [20]. ◀

▶ **Lemma 13.** *It holds* $I_i^{k,V}(R_i^k) = \sum_{\upsilon_j^k \in \Upsilon_k} \left\lceil \frac{R_i^k}{T_j^{k,V}} \right\rceil \cdot C_j^{k,V}$.

**Proof.** Voting tasks have a higher priority than any task in $\Gamma_k$, hence they all generate high-priority interference to $\tau_i^k$. They are also periodically activated and execute as standard periodic tasks. Hence the lemma follows as for Lemma 12 provided that the WCET bound of Theorem 11 is used. ◀

## 6.3 Discussion

As it can be noted from the above sections, the analysis of voting with passive waiting is much more challenging than the one under LET-inspired voting due to the various sources of unpredictability introduced by that scheme. In addition, passive waiting requires the analysis of packet queuing presented in Section 5 to deal with any-time packet transmissions.

On the other hand, passive waiting is relatively simple to implement from the perspective of the programmer and does not require introducing additional tasks in the system. Furthermore, it introduces limited priority inversion related to voting: indeed, a high-priority task can be delayed by voting-related activities of low-priority tasks only by the transmission of one packet and the reception of packets by means of ISRs.

Conversely, LET-inspired voting does not require the packet queuing analysis since the voting data is communicated in precise time intervals during which the interested voting tasks are synchronously executed on both replicas. Nevertheless, this approach tends to introduce larger priority inversion because all voting-related activities are executed by LET tasks at the highest priority. Hence, the whole transmission and reception of packets as well as the voting protocol of a low-priority task can interfere with the execution of a high-priority task.

# 7 Experimental results

This section reports the results of an experimental evaluation that was conducted to compare those two voting scheduling strategies studied in this paper.

**Workload generation.** Given a target task set utilization $U$ and a number of tasks $n$, $N$ task sets have been generated with the Emberson et al.'s generator [15], which was configured to randomly select the task periods in the range $[T^{\min}, T^{\max}]$ with log-uniform distribution. The task sets $\Gamma_1$ and $\Gamma_2$ of the two replicas were then generated as follows. For each replica pair $r_i$, one replica was randomly selected to be the slower in executing it, say $\mathcal{R}_k$, then $E_i^{or(k)}$ was set to the WCET value obtained by the task generator and $E_i^k = E_i^{or(k)} \cdot \xi$, where $\xi$ was randomly selected in $[\xi^{min}, 1]$ with uniform distribution. Note that, since the WCET provided by the task generator is used to control the worst-case duration of the execution phase of tasks (parameter $E_i^k$), the utilization $U$ used to control the generation refers to the maximum per-replica utilization *without voting-related activities.* A random number $\lfloor p^{\text{vital}} \cdot n \rfloor$ of tasks, with $p^{\text{vital}}$ randomly chosen in $[0.6, 0.8]$ with uniform distribution, were selected to be vital in each replica pair, and hence to require voting. For each vital replica pair, the number of packets $M_i$ was randomly generated in $[0, M^{\max}]$ with uniform distribution.

**Table 1** Nominal setting of the parameters that control the workload generation.

| Parameter | Value | Description |
|---|---|---|
| $N$ | 500 | Number of task sets |
| $n$ | 10 | Number of tasks per replica |
| $p^{\text{vital}}$ | [0.6, 0.8] | Vital task ratio for each replica pair |
| $T^{\min}$ | 5000 | Task minimum period ($\mu s$) |
| $T^{\max}$ | 500000 | Task maximum period ($\mu s$) |
| $\xi^{\min}$ | 0.85 | Minimum faster-replica speed coefficient |
| $M^{\max}$ | 5 | Maximum number of packets sent by a task |
| $\alpha$ | 15 | ICI bandwidth ($MB/s$) |
| $b$ | 16 | Number of bytes per packet |
| $Q$ | 8 | ICI queue size (packets) |
| $\gamma$ | 13.24 | Minimum read/write rate to access memory ($MB/s$) |
| $\underline{\beta}$ | 13.24 | Minimum read/write rate to access device registers ($MB/s$) |
| $\overline{\beta}$ | 56.47 | Maximum read/write rate to access device registers ($MB/s$) |
| $\sigma^{\text{ISR}}$ | 2 | ISR overhead ($\mu s$) |
| $\lambda^{\text{VP}}$ | 50 | Time required to vote a packet of data ($\mu s$) |

Parameters $VR_i$ and $VT_i$ were computed accordingly as a function of $M_i$. The WCET of the voting protocol was also generated as $VP_i = \lambda^{\text{VP}} \cdot M_i$ where $\lambda^{\text{VP}} \geq 0$ is another parameter that control the generation. For non-vital replica pairs we set $VP_i = VT_i = VR_i = 0$. All tasks were assigned implicit deadlines (i.e., $D_i = T_i$).

To configure the device register and memory access rates $\underline{\beta}$, $\overline{\beta}$, and $\gamma$ we took the Xilinx Ultrascale+ SoC (considering the Cortex-A cores running at 1.2 GHz) as a reference platform, from which we obtain respectively, 725, 170, and 170 clock cycles by profiling. The configuration of other parameters that are not mentioned above is varied in the experiments presented next and, whenever mentioned, is kept fixed to the nominal setting reported in Table 1.

**Experiments.** A first experiment was conducted by varying the utilization without voting $U$ and testing $N = 500$ task sets per utilization value. The results under four representative configurations are reported in Figure 4. The plots report the schedulability performance of the proposed analysis techniques for voting with passive waiting (Section 4.1) and LET-inspired voting (Section 4.2), as well as for the system without voting activities (used as a reference upper bound of the schedulability performance). These results were obtained under the setting reported above each plot, where the parameters that are not mentioned were set to the nominal configuration of Table 1.

As it can be noted from the plots, LET-inspired voting always outperforms passive waiting. Passive waiting is strongly penalized in the presence of short ICI queues (see Fig. 4(a) vs. Fig. 4(b)) due to the queuing analysis, while LET-inspired voting is almost insensitive to the ICI queue size as expected. The performance of both approaches degrades as the number of packets sent by tasks increases (see Fig. 4(c) vs. Fig. 4(d)), but LET-inspired voting is capable of guaranteeing much better schedulability performance than passive waiting as $M^{\max}$ increases.

Another experiment was conducted to study the dependency of the schedulability performance of the two approaches as a function of other parameters different than $U$. The results are reported in Figure 5, where 500 task sets have been tested for each value of the varied parameters. Figure 5(a) illustrates the dependency of the schedulability performance on the minimum task period $T^{\min}$ in a condition of high system load (U= 0.9). This figure clearly

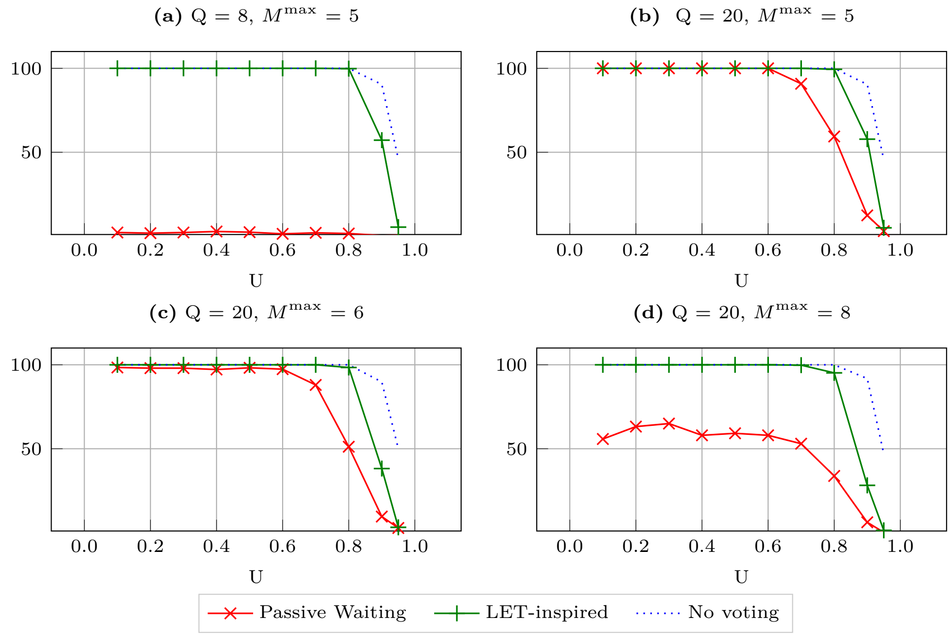


**Figure 4** Schedulability ratio (y-axis of the plots) as a function of the voting-unrelated utilization $U$ used to control the task set generation under four representative configurations.

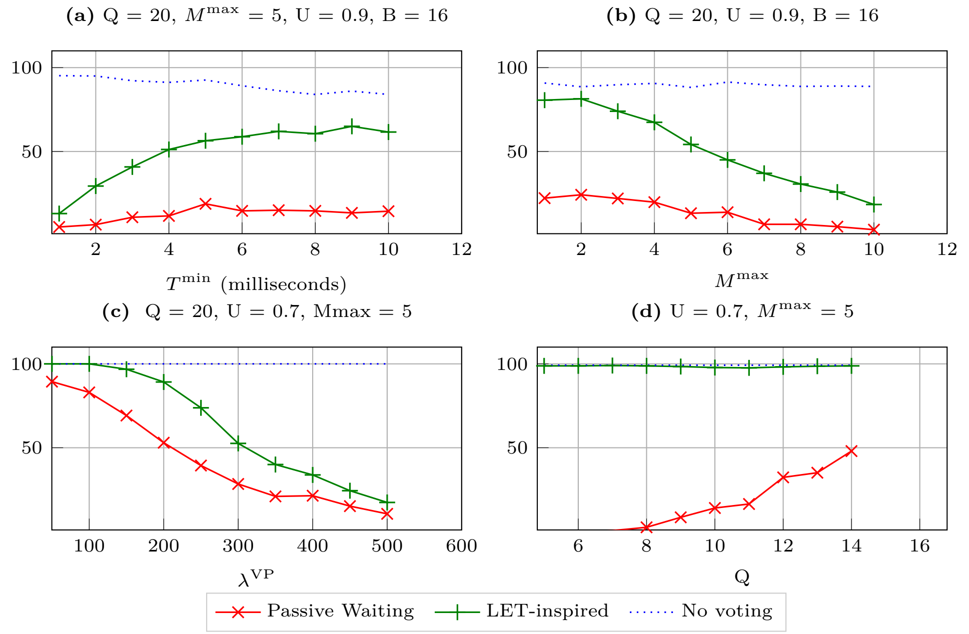


**Figure 5** Schedulability ratio (y-axis of the plots) as a function of $T^{\min}$, $M^{\max}$, $\lambda^{\mathrm{VP}}$, and $Q$ under four representative configurations.

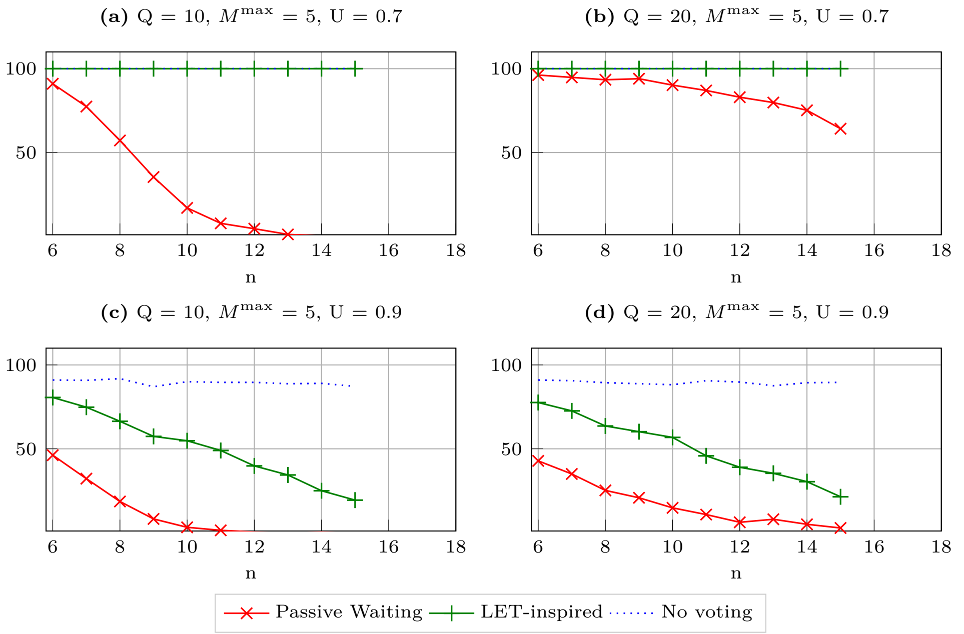


**Figure 6** Schedulability ratio (y-axis of the plots) as a function of $n$ under four representative configurations.

shows that LET-inspired voting is penalized in the presence of very short task periods due to the priority-inversion generated by voting tasks discussed in Section 6.3. Figures 5(b) and 5(c) show how the performance of both approaches degrades as either $M^{\max}$ or $\lambda^{\mathrm{VP}}$ increases, and that the gap between the two reduces for large values of these parameters. Finally, Figure 5(d) confirms that passive waiting exhibits very poor performance in the presence of short ICI queues and that LET-inspired voting is insensitive to this parameter. The last experiment was conducted to assess how the schedulability ratio of both approaches varies as a function of the number of tasks in the tested task sets. Figure 6(a) shows that the performance of passive waiting quickly degrades by increasing the number of tasks while LET-inspired voting is not affected by the size of the task set. Figure 6(b) reports the results under the same configuration of Figure 6(a) but considering larger ICI queues: in this case, the performance of passive waiting definitively improves but is still lower than the one of LET-inspired voting. Furthermore, Figure 6(c) and Figure 6(d) show that the performance of both approaches decreases as the number of tasks increases at high utilization ($U = 0.9$). Nevertheless, LET-inspired voting always outperforms passive waiting in all the tested cases.

## 8 Conclusion and future work

This paper studied two scheduling strategies for distributed voting protocols in 2-out-of-2 redundant real-time systems, namely passive waiting (based on task self-suspensions to wait for the other replica) and LET-inspired voting. Both queuing and delays related to inter-replica communication interfaces have been studied. Response-time analysis for real-time tasks under the two strategies has been presented. The pros and cons of the two scheduling strategies have also been discussed. The two strategies have been experimentally compared in terms of schedulability performance. The experimental results revealed that LET-inspired voting is always preferable to passive waiting, exhibiting even a 100% performance gap in the presence of short packet queues of inter-replica communication interfaces. In other configurations with longer queues, LET-inspired voting is also capable of scheduling up to five more times task sets than passive waiting.

Future work should investigate the possibility of improving the analysis of passive waiting, both in terms of packet queuing and response times, and on the design of improved scheduling strategies that can better control the priority inversion introduced by LET-inspired voting.

## References

1 Jaemin Baek, Jeonghyun Baek, Jeeheon Yoo, and Hyeongboo Baek. An n-modular redundancy framework incorporating response-time analysis on multiprocessor platforms. *Symmetry*, 11(8):960, 2019.

2 Julian M Bass. *Voting in real-time distributed computer control systems*. PhD thesis, University of Sheffield, 1995.

3 H Benítez-Pérez, G Latif-Shabgahi, HA Thompson, S Bennett, PJ Fleming, and JM Bass. Integration and comparison of fdi and fault masking features in embedded systems. *IFAC Proceedings Volumes*, 32(2):7712–7717, 1999.

4 Guillem Bernat, Jose Miro-Julia, and Julian Proenza. A technique to analyze the tolerance to transient overloads of a fault-tolerant real-time system. In *Proceedings 1997 High-Assurance Engineering Workshop*, pages 221–226. IEEE, 1997.

5 Guillem Bernat, Jose Miro-Julia, Julian Proenza, et al. Fixed priority schedulability analysis of a distributed real-time fault tolerant architecture. In *PDPTA*, pages 479–487, 1997.

6 Dimitri Bertsekas and Robert Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992.

7 DM Blough and GF Sullivan. Voting using predispositions. *IEEE Transactions on reliability*, 43(4):604–616, 1994.

8 Douglas M Blough and Gregory F Sullivan. A comparison of voting strategies for fault-tolerant distributed systems. In *Proceedings Ninth Symposium on Reliable Distributed Systems*, pages 136–145. IEEE, 1990.

9 B. Brandenburg. Scheduling and locking in multiprocessor real-time operating systems. In *Ph.D. dissertation, The University of North Carolina at Chapel Hill*, 2011.

10 D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 421–433, 2018. `doi:10.1109/RTSS.2018.00056`.

11 EN CEI. Cei en 50126-1. *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Part 1: Generic RAMS Process*, 2019.

12 EN CEI. Cei en 60730-1. *Automatic electrical controls - Part1: General requirements*, 2019.

13 J. Chen, G. Nelissen, and W. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 61–71, 2016. `doi:10.1109/ECRTS.2016.31`.

14 Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Computer Architecture Letters*, 27(06):531–539, 1978.

15 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010.

16 Oytun Eriş, Uğur Yıldırım, Mustafa S Durmuş, Mehmet T Söylemez, and Salman Kurtulan. N-version programming for railway interlocking systems: Synchronization and voting strategy. *IFAC Proceedings Volumes*, 45(24):177–180, 2012.

17 Saurabh Gohil, Aravind Basavalingarajaiah, and Varadharajan Ramachandran. Redundancy management and synchronization in avionics communication products. In *2011 Integrated*

*Communications, Navigation, and Surveillance Conference Proceedings*, pages C3–1. IEEE, 2011.

18 Arpan Gujarati, Sergey Bozhko, and Björn B Brandenburg. Real-time replica consistency over ethernet with reliability bounds. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 376–389. IEEE, 2020.

19 T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003. doi:10.1109/JPROC.2002.805825.

20 M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, January 1986. doi:10.1093/comjnl/29.5.390.

21 Hagbae Kim and Kang G Shin. Sequencing tasks to minimize the effects of near-coincident faults in tmr controller computers. *IEEE transactions on computers*, 45(11):1331–1337, 1996.

22 Thomas Kugelstadt. Extending the spi bus for long-distance communication. *Analog Applications Journal*, 2011. URL: https://www.ti.com/lit/an/slyt441/slyt441.pdf.

23 J.F. Kurose and K.W. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, Limited, 2010. URL: https://books.google.it/books?id=2hv3PgAACAAJ.

24 Seong Woo Kwak and Byung Kook Kim. Task-scheduling strategies for reliable tmr controllers using task grouping and assignment. *IEEE Transactions on Reliability*, 49(4):355–362, 2000.

25 G Latif-Shabgahi, JM Bass, and S Bennett. Complete disagreement in redundant real-time control applications. *IFAC Proceedings Volumes*, 31(4):223–228, 1998.

26 G Latif-Shabgahi, Julian M Bass, and Stuart Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328, 2004.

27 Stephen R McConnel and Daniel P Siewiorek. Synchronization and voting. *IEEE Transactions on Computers*, 100(2):161–164, 1981.

28 P. Pazzaglia, D. Casini, A. Biondi, and M. Di Natale. Optimal memory allocation and scheduling for dma data transfers under the let paradigm. In *58th Design Automation Conference (DAC)*, 2021.

29 Dai Shenghua and Li Yishi. Research on 2-out-of-2 multiplying 2 redundancy system used in high-speed train. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, volume 2, pages 483–486. IEEE, 2011.

30 Martin L Shooman. *Reliability of computer systems and networks*. Wiley Online Library, 2002.

31 Daniel Siewiorek and Robert Swarz. *Reliable computer systems: design and evaluatuion*. Digital Press, 2017.

32 Daniel P Siewiorek and Priya Narasimhan. Fault-tolerant architectures for space and avionics applications. *NASA Ames Research http://ic. arc. nasa. gov/projects/ishem/Papers/Siewi*, 2005.

33 Zhijun Tong and Richard Y Kain. Vote assignments in weighted voting mechanisms. *IEEE Transactions on Computers*, 40(5):664–667, 1991.

34 Xilinx. Zynq-7000 soc: Dc and ac switching characteristics - ds191, 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds191-XC7Z030-XC7Z045-data-sheet.pdf#G1940899.

# A Residual Service Curve of Rate-Latency Server Used by Sporadic Flows Computable in Quadratic Time for Network Calculus

**Marc Boyer**

ONERA / DTIS – Université de Toulouse, F-31055 Toulouse, France

**Pierre Roux**

ONERA / DTIS – Université de Toulouse, F-31055 Toulouse, France

**Hugo Daigmorte**

RealTime-at-Work, F-54600 Villers-lès-Nancy, France

**David Puechmaille**

RealTime-at-Work, F-54600 Villers-lès-Nancy, France

## Abstract

Computing response times for resources shared by periodic workloads (tasks or data flows) can be very time consuming as it depends on the least common multiple of the periods. In a previous study, a quadratic algorithm was provided to upper bound the response time of a set of periodic tasks with a fixed-priority scheduling. This paper generalises this result by considering a rate-latency server and sporadic workloads and gives a response time and residual curve that can be used in other contexts. It also provides a formal proof in the Coq language.

**2012 ACM Subject Classification** Networks → Formal specifications; Networks → Network performance evaluation; Networks → Network reliability; Software and its engineering → Formal methods; General and reference → Verification

**Keywords and phrases** Network Calculus, response time, residual curve, rate-latency server, sporadic workload, formal proof, Coq

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.14

**Supplementary Material** The code of the Coq proof is provided.

*Software*: `http://doi.org/10.5281/zenodo.4518843`

*Software (ECRTS 2021 Artifact Evaluation approved artifact)*:

`https://doi.org/10.4230/DARTS.7.1.2`

## 1 Introduction

Network calculus is a theory designed to compute upper bounds on delays and memory usage in distributed real-time systems. Given such a system, network calculus offers different ways to model it and different algorithms, producing different bounds at different computation costs.

Even if network calculus is able to analyse realistic industrial configurations in a few seconds [10], some operations have an exponential worst case complexity, related to the least common multiple (lcm) of the periods of the involved flows.

Currently, when modelling periodic or sporadic flows, one often use either an affine (*i.e.* fluid) model, with linear complexity, or a staircase model, with exponential complexity. This paper presents a quadratic solution for a very common operation, involved in the computation of a residual service for common scheduling policies.

This paper is inspired by [2], that gave a quadratic algorithm for the response time of a set of periodic real-time tasks on a CPU with fixed-priority scheduling. Since network calculus also offers methods to compute upper bounds on the response time of such systems,

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 14; pp. 14:1–14:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

we had a look on the proof itself, and we found that it relies on the computation of the CPU capacity that is left to some task by the higher priority flows. This notion also exists in network calculus, where it is called “residual service” or “left-over capacity”. This paper adapts the result in [2] to the network calculus framework and generalises it.

Since the proof is quite long, a formal proof, checked by the Coq proof assistant [13], is also provided.

After a presentation of a relevant subset of network calculus in Section 2, and an overview of related work in Section 3, the result itself is presented in Section 4, and evaluated on benchmarks in Section 5.

## 2 Network calculus

This section provides a recall of network calculus formalism in Section 2.1, with a focus on sporadic workload and rate-latency servers in Section 2.2.

Let $\mathbb{R}$ denote the set of real numbers, $\mathbb{R}^+$ the subset of non-negative real numbers, $\mathbb{Z}$ the set of integers, for any $i, j \in \mathbb{Z}$, $[\![i, j]\!] = \{i, i + 1, \ldots, j\}$, $\lceil \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ the ceiling function ($\lceil 1.2 \rceil = 2$, $\lceil 4 \rceil = 4$, $\lceil -1.2 \rceil = -1$). For any set $X$, $|X|$ denotes its cardinal. For any number $x \in \mathbb{R}$, $[x]^+ = \max(x, 0)$. For any function $f : \mathbb{R}^+ \to \mathbb{R}$, its non-decreasing non-negative closure (illustrated in Figure 1) is defined by $[f]^+_\uparrow (t) = \max_{0 \le s \le t} [f(s)]^+$.

### 2.1 Generic results

Network calculus is a theory for deriving deterministic upper bounds in networks. Network calculus mainly manipulates non decreasing functions to model flows, workload and server capacity. This section provides a short introduction. A more thorough treatment can be found in [12, 20, 5].

In network calculus, input and output flows of data are modelled by cumulative functions which represent the amount of data observed at some point the flow up to time $t$. Servers are just relations between input and output flows: a server $S$ receives an arrival/input flow, $A(t)$, and delivers the data after some delay, as a departure/output flow, $D(t)$. We always have the relation $D \le A$, meaning that data can only go out after its arrival.

If the order of data within the flow is preserved, the *delay* at time $t$ is defined as $hDev(A, D, t)$, and the worst delay is $hDev(A, D)$, with

$$hDev(A, D, t) \overset{def}{=} \inf \left\{ d \in \mathbb{R}^+ \mid A(t) \le D(t + d) \right\}, \qquad hDev(A, D) \overset{def}{=} \sup_{t \in \mathbb{R}^+} hDev(A, D, t)$$

(see Figure 3 for an illustration).

However, the exact input/output data flows are in general unknown at design time, or too complex, and the calculus of these cumulative functions cannot be obtained. Nevertheless, the evolution of input/output data flows can be bounded considering contracts on the traffic and the services in the network. For this purpose, network calculus provides the concepts of arrival curve (illustrated in Figure 2) and service curve.

▶ **Definition 1** (Arrival curve). *Let $A$ be a flow, and $\alpha$ a function. Then, $\alpha$ is said to be an arrival curve for flow $A$, iff $\forall (t, d) \in \mathbb{R}^+ \times \mathbb{R}^+,\ A(t + d) - A(t) \le \alpha(d)$.*

The expression $\alpha(d)$ is an upper bound on the amount of data that can be generated on any interval of duration $d$. For a given flow $A$, one may consider several arrival curves.

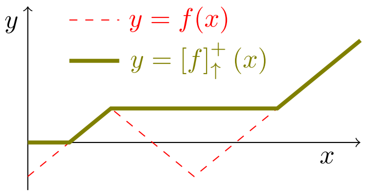


**Figure 1** Non-negative non-decreasing closure.

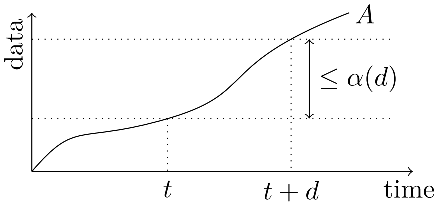


**Figure 2** Arrival curve.

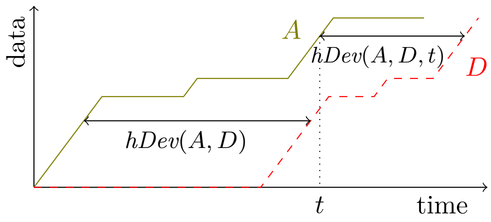


**Figure 3** Delay of the flow A.

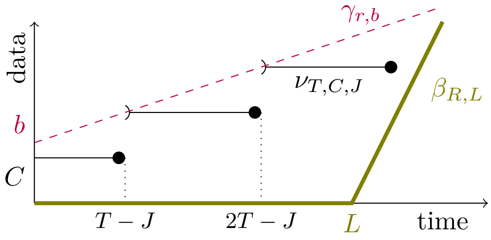


**Figure 4** Common curves.

▶ **Definition 2** (Minimal service). *A server $S$ offers a strict minimal service curve $\beta$ iff for all input/output $A$,$D$ and for any backlogged period $(s,t]$ (*i.e.* such that $\forall x \in (s,t] : A(x) > D(x)$)*

$$D(t) - D(s) \geq \beta(t-s). \tag{1}$$

Let us now present the main network calculus result which allows, considering contracts, to compute bounds on delay.

▶ **Theorem 3** (Delay bound). *Let $S$ be a server transforming an arrival $A$ into a departure $D$. If $A$ has arrival curve $\alpha$ and $S$ offers a strict minimal service of curve $\beta$ then*

$$hDev(A,D) \leq hDev(\alpha,\beta). \tag{2}$$

A key point in network calculus is that arrival and service curves do not have to be tight. Mathematically they only have to be, respectively, upper and lower bounds (cf. eq. (1), eq. (1)). From a modelling point of view, they are not the exact behaviour, but only contracts. It has two complementary consequences. On one hand, if the contract is too far away from the real behaviour, the computed bounds will be large w.r.t. the real worst case. On the other hand, a complex contract can be approximated by a simpler one and all results still hold.

## 2.2 Sporadic workload, rate-latency servers and NP-SP policy

This paper focuses on periodic or sporadic flows and rate-latency servers.

Given a flow sending frames of maximal size or cost $C \in \mathbb{R}^+$ with a period or minimal inter-arrival time $T \in \mathbb{R}^+$ and a jitter $J \in \mathbb{R}^+$, it admits the arrival curve $\nu_{T,C,J} : \mathbb{R}^+ \to \mathbb{R}^+$, $t \mapsto C \left\lceil \frac{t+J}{T} \right\rceil$ but also $\gamma_{r,b} : \mathbb{R}^+ \to \mathbb{R}^+$, $t \mapsto rt + b$ ([1]), with $r = \frac{C}{T}$ and $b = r(J+T)$, as

[1] Readers with some background in network calculus may notice that in our definition, $\gamma_{r,b}(0) = b$ whereas the common practice is to set $\gamma_{r,b}(0) = 0$. But the results are simpler to prove with this definition, and can be easily extended to the case where the function is null at origin.

illustrated in Figure 4. Using $\nu_{T,C,J}$ is called “staircase modelling” while using $\gamma_{r,b}$ is called “fluid modelling”.

Servers often offer a rate-latency service, *i.e.* a constant rate $R$ (a data link bandwidth for example) after some latency $L$ (some switching delay for example), modelled by a function $\beta_{R,L} : t \mapsto R\,[t-L]^+$.

When several flows share a server, its capacity $\beta$ is shared between the flows, and to compute an upper bound on the delay for a flow of interest, network calculus offers to compute a residual service (aka. left-over service). The expression depends on the scheduling policy.

▶ **Theorem 4** (NP-SP residual service). *Let $S$ be a server shared by $n$ flows using a non-preemptive static priority scheduling policy. If $S$ offers a strict minimal service curve $\beta_{R,0}$, and each flow $i$ has arrival curve $\alpha_i$ and a maximal frame size $S_i$, then each flow $j$ receives a residual service*

$$\beta_j = \left[\beta_{R,S_j^{\max}/R} - \sum_{k \in hp(j)} \alpha_k\right]_{\uparrow}^{+} \tag{3}$$

*with $S_j^{\max} = \max_{k \in lp(j)} S_k$, and $hp(j)$ (resp. $lp(j)$) the set of flows with higher (resp. lower) priority than $j$.*

The same kind of result holds, with some variations, with other type of service curves, or preemptive static priority server, FIFO or even EDF [5][2].

## 2.3 Illustrative example

Consider a bus with a bandwidth of 125kb/s, a non-preemptive static priority arbitration rule, and a latency of 0.75ms. Three periodic flows, with period and packets sizes given in Table 1 are sharing this bus. Flow 3 has the lowest priority, then $hp(3) = \{1, 2\}$.

To compute the delay of this flow 3, one may choose to apply eq. (3). One may then either set $\alpha_i = \nu_{T_i,C_i,J_i}$ (staircase modelling) or $\alpha_i = \gamma_{r_i,b_i}$ (fluid modelling). In the top plot of Figure 5 are plotted the two staircase arrival curves, $\nu_1, \nu_2$ and the corresponding fluid arrival curves $\gamma_1, \gamma_2$. Eq. (3) involves the sum $\alpha_1 + \alpha_2$, being either $\nu_1 + \nu_2$ or $\gamma_1 + \gamma_2$ (both are in the second plot of Figure 5), and the two residual services $\beta_3^{stc} = [\beta - \nu_1 - \nu_2]_{\uparrow}^{+}$, $\beta_3^{fluid} = [\beta - \gamma_1 - \gamma_2]_{\uparrow}^{+}$ are also plotted (given in the third and fourth plots of Figure 5). The latency is then bounded either by $hDev(\alpha_3, \beta_3^{stc}) = h_1$ or $hDev(\alpha_3, \beta_3^{fluid}) = h_1 + h_2 + h_3$.

As expected, the staircase modelling, that captures in a more accurate way the behaviour of the flows, gives a smaller bound than the fluid modelling.

## 2.4 Problem statement

Whereas the staircase modelling computes better bounds, it has several drawbacks.

One problem is the cost of the addition (*i.e.* the term $\sum_{k \in hp(j)} \alpha_k$ in equation 3). With a fluid model (when it exists real values $r_k, b_k$ such that $\alpha_k = \gamma_{r_k,b_k}$) there exists a closed-form formula whose cost is linear w.r.t. the number of curves (it holds $\sum_{k \in hp(j)} \gamma_{r_k,b_k} = \gamma_{r,b}$ with

[2] Readers with a background in network calculus may have noticed only strict minimal service is presented, whereas applications of these results also involve min-plus minimal service. Since the contribution of this paper is independent of the service type, only one notion has been presented.

**Table 1** Flow parameters of illustrative example.

| $i$ | $T_i$ | $C_i$ | $r_i$ | $b_i$ |
|---|---|---|---|---|
| 1 | 2.5 ms | 125 b | 50 kb/s | 125 b |
| 2 | 3.5 ms | 125 b | 35.72kb/s | 125 b |
| 3 | 3 ms | 100 b | 33.33 kb/s | 100 b |

$r = \sum_{k \in hp(j)} r_k$ and $b = \sum_{k \in hp(j)} b_k$). On the contrary, the addition with a staircase model is hard: there exists no closed-form formula, only algorithms [6], and the computation requires to unroll the function up to the least common multiple of the periods[3], leading to exponential complexity.

Another problem is the absence of a closed-form formula. Closed formulae, and especially those involving linear terms, allow to perform explicit and efficient optimisations.

A last problem is related to the implementation: not all tools are able to handle staircase functions, and several only consider linear arrival curves, as presented in the next section.

The contribution of this paper is to give a rate-latency residual service that lies between the staircase and the fluid residual service curves, denoted $\beta_{R',C/R'}$ in Figure 5.

# 3 Related work

## 3.1 Implementation of algebraic operators for network calculus

Practical application of network calculus requires an implementation of algebraic operations on functions.

For years, work has concentrated exclusively on linear functions, using closed-form formulae [20], and some tools were even only using affine arrival curves and rate-latency service curves [3].

The subclass of concave or convex piecewise linear functions has also received some attention [25, 7] and is the class currently used in the DISCO tool [26, 4].

A big step was the development of the (min,plus) library for the RTC toolbox [29], representing piecewise linear functions (called VCCs) as a collection of segments [28, Sec. 7].

A major breakthrough has been achieved with the definition of the class of ultimately pseudo periodic functions, generalising VCCs, and the development of the algorithms allowing effective computation [6].

The problem of computation time has not yet received a lot of attention in academia.

In [9], the idea is to maintain a staircase arrival curve per flow, but to approximate it by a concave piecewise linear function of two segments before summing, to keep linear complexity.

The notion of a “container” is developed in [21], with $O(n \log n)$ complexity on operations.

Another line of work is based on the fact that the computation of the bounds (the horizontal deviation, *hDev*) is based only on the prefix of the involved functions, and that one can maintain only a prefix and approximate the remainder of the function by an affine segment [17, 18, 27].

Lastly, another way to reduce the computation time (at the price of getting larger upper bounds) is to replace some periods $T_i$ by a smaller value $T'_i$ but such that the lcm of the $T'_i$ is smaller than the lcm of the $T_i$ [23, 24].

[3] In practice, periods are often integers or rational numbers that can be mapped to integers once a common denominator is found.

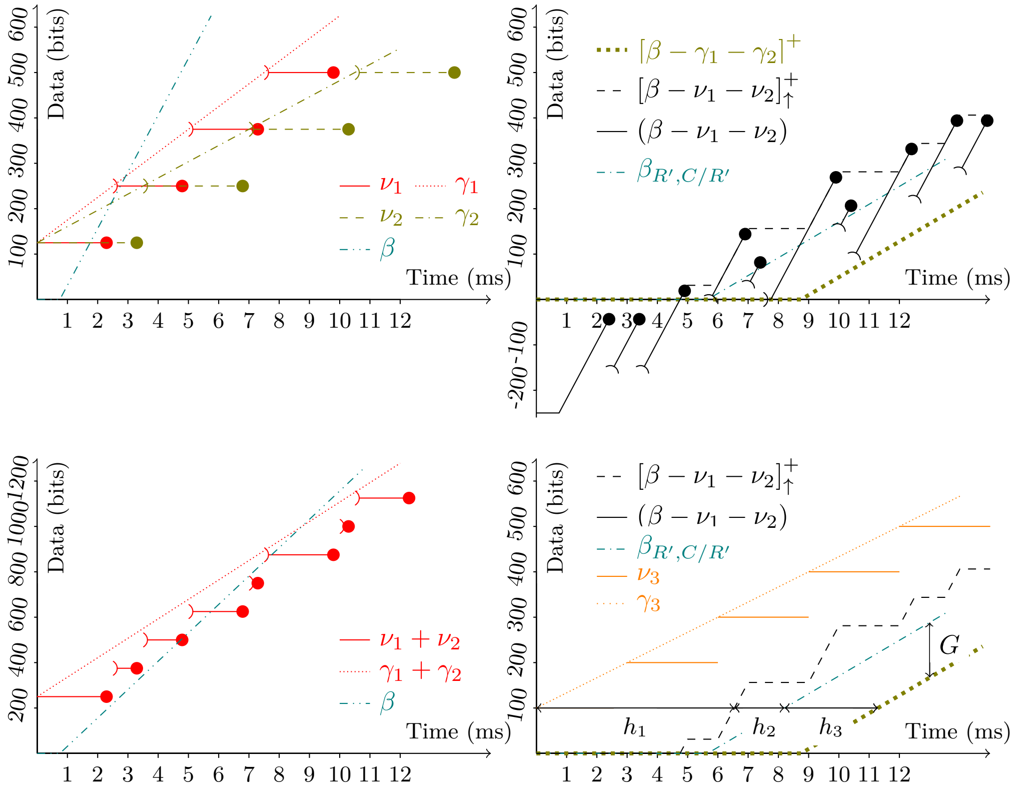


**Figure 5** Illustration of different curves, with $R = 125kb/s$, $L = .75ms$ and $\forall i \in \{1, 2, 3\}, \nu_i = \nu_{T_i,C_i,0}$, $\gamma_i = \gamma_{r_i,b_i}$, for the values of $T_i, C_i, r_i$ and $b_i$ given in Table 1, and $G$ as in Theorem 5.

## 3.2 Coq for real-time systems

Coq is a proof assistant [13], *i.e.*, a tool offering a language to state theorems and describe their proofs as well as a software[4] verifying the proofs. It can also be used to develop software whose execution is proved to be conform to their (formal) specification such as the CompCert C compiler [22] or the CertiKOS operating system [16]. When used as a proof checker, Coq will complain when attempting to use a lemma without providing a proof for one of its hypotheses or if the proved hypotheses do not match the expected ones.

Proving that a systems guarantees some real-time property is often a complex task, requiring long and complex proofs. One way to build correct analyses is to use a proof assistant, like Coq [11] or Isabelle/HOL.

# 4 Contribution

This section details the main contribution of the paper: given a rate-latency curve $\beta_{R,T}$ and a set of staircase functions $\nu_{T_i,C_i,J_i}$, there exists a rate-latency function $\beta_{R',C/R'}$ which is a lower bound, as shown in eq. (4), that can be used to compute residual service. The main results are presented in Section 4.1. Since the proof of the main theorem is quite long, it is presented in Appendix A. The statement of the theorem in Coq is presented in Section 4.2.

[4] Think of it as a compiler (in practice it is indeed a compiler for a very strongly typed language).

The problem is illustrated in Figure 5. It shows that each function $\gamma_i$ is a good fluid approximation of the function $\nu_i$ (e.g. $\gamma_1(t) = \nu_1(t)$ for $t = 2.5 \times k$, $k \in \mathbb{N}$) and even if there are less equality point between the two sums ($(\gamma_1 + \gamma_2)(t) = (\nu_1 + \nu_2)(t)$ for $t = 17.5 \times k$, $k \in \mathbb{N}$), $\gamma_1 + \gamma_2$ it is still the best possible affine upper approximation of $\nu_1 + \nu_2$. And the distance between $(\beta - \gamma_1 - \gamma_2)$ and $(\beta - \nu_1 - \nu_2)$ is exactly the same as the one between $\gamma_1 + \gamma_2$ and $\nu_1 + \nu_2$. But the non-decreasing closure has a major impact on the expression based on staircase, but none on the fluid one, creating a larger distance between both functions (e.g. at $t = 11$, $(\beta - \nu_1 - \nu_2)(11)$ is close to $(\beta - \gamma_1 - \gamma_2)(11)$ but far away from $[\beta - \nu_1 - \nu_2]_{\uparrow}^{+}(11)$).

## 4.1 A quadratic rate-latency bound

▶ **Theorem 5** (Quadratic rate-latency bound). *Let $R, L, C_1, \ldots, C_n, T_1, \ldots, T_n$ (resp. $J_1, \ldots, J_n$) be a set of positive real (resp. non negative real) values such that $\sum_{i=1}^{n} \frac{C_i}{T_i} < R$. Then*

$$\left[\beta_{R,L} - \sum_{i=1}^{n} \nu_{T_i,C_i,J_i}\right]_{\uparrow}^{+} \geq \beta_{R',C/R'} \tag{4}$$

*with*

$$R' = R - \sum_{i=1}^{n} \frac{C_i}{T_i}, \qquad C = RL + W - \frac{\max\left\{G^l, G^q\right\}}{R}, \qquad W = \sum_{i=1}^{n} \left(T_i + J_i - \frac{C_i}{R}\right) \frac{C_i}{T_i},$$

$$G^l = \min_{k \in [\![1,n]\!]} C_k \left(\sum_{i=1}^{n} \frac{C_i}{T_i} - \max_{i \in [\![1,n]\!]} \frac{C_i}{T_i}\right), \qquad G^q = \sum_{i=1}^{n} \sum_{j=1}^{i-1} \min\left\{T_i, T_j\right\} \frac{C_i C_j}{T_i T_j}.$$

In the context of a rate-latency server shared by several sporadic flows with a static priority policy, the term $R'$ represents the residual rate, made of the initial rate minus the utilisation of higher priority flow $\frac{C_i}{T_i}$, and the term $C$ represents an upper bound on the backlog of higher priority flows.

The expression of the function $\beta_{R',C/R'}$ involves only simple sums (sub-terms $R'$, $W$ and $G^l$) and one double sum (sub-term $G^q$) leading to quadratic complexity $O(n^2)$. To obtain a linear complexity, one may omit the term $G^q$, leading to a smaller curve (*i.e.* a worst service) but in a shorter time.

Two proofs are given. In Appendix A is given a "pen and paper" proof. This proof being non trivial, we chose to get a high level of confidence in its soundness by formalizing and verifiying it with Coq. A feedback of this use is given at the end of the current section and an overview of the Coq proof is given in Section 4.2.

The next theorem states that the previous result is an enhancement w.r.t. a fluid modelling.

▶ **Theorem 6.** *Let $R, L, C_1, \ldots, C_n, T_1, \ldots, T_n, J_1, \ldots, J_n, R', C$ be as in Theorem 5. Then*

$$\left[\beta_{R,L} - \sum_{i=1}^{n} \gamma_{r_i,b_i}\right]^{+} = \left[\beta_{R',C/R'} - G\right]^{+} \qquad \text{with } G = \left(\sum_{i=1}^{n} \frac{C_i^2}{T_i} + \max\left\{G^l, G^q\right\}\right) \tag{5}$$

*$r_i = \frac{C_i}{T_i}$, $b_i = r_i(J_i + T_i)$, and $R', C, G^l, G^q$ defined as in Theorem 5.*

The term $G$ is the global gain obtained with the new result from Theorem 5 w.r.t. a fluid modelling.

**Proof.** The first step consists in an expression of linear residual service. First, $\sum_{i=1}^n \gamma_{r_i,b_i} = \gamma_{\sum_{i=1}^n r_i, \sum_{i=1}^n b_i}$, then for any $t \in \mathbb{R}^+$ :

$$\left[\beta_{R,L}(t) - \sum_{i=1}^{n} \gamma_{r_i,b_i}(t)\right]^+ = \left[[R(t-L)]^+ - \left(\sum_{i=1}^{n} r_i\right) t - \sum_{i=1}^{n} b_i\right]^+ \tag{6}$$

$$= \left[\left(R - \sum_{i=1}^{n} r_i\right) t - \left(RL + \sum_{i=1}^{n} b_i\right)\right]^+ \tag{7}$$

$$= \beta_{R',C'/R'} \tag{8}$$

with $C' = RL + (\sum_{i=1}^n b_i)$. Let now compare $C$ and $C'$

$$C' = RL + \sum_{i=1}^{n} (J_i + T_i) \frac{C_i}{T_i} = C + \frac{1}{R} \left( \sum_{i=1}^{n} \frac{C_i^2}{T_i} + \max\left\{G^l, Gq\right\} \right). \tag{9}$$

◀

Figure 5 illustrates the differences between the functions and highlights the influence of the non-decreasing closure. As expected, since fluid modelling gives a larger arrival curve than staircase modelling ($\gamma_i \geq \nu_i$), then the fluid residual curve is less than or equal to the staircase one: $\beta - \sum_i \gamma_i \leq \beta - \sum_i \nu_i$. As stated by the Theorem, $\beta_{R',C/R'} \leq [\beta - \sum_i \gamma_i]_\uparrow^+$ but $\beta_{R',C/R'}$ is not smaller than $\beta - \sum_i \gamma_i$.

### Comparison with [2]

This result is of course closely related to the one in [2], and once the equation is given, the amount of generalisation can be detailed. Using network calculus, the response time of a task of execution time $C_0$ and period $T_0$ on a CPU with speed one ($R = 1$) and no latency ($L = 0$) can be bounded by $hDev(\gamma_{C_0/T_0,C_0}, \beta_{R',C/R'}) = C/R' + C_0/R' = \frac{C_0+W-\max\{L,Q\}}{R'}$ whereas the expression in [2, Thm. 1] is $\frac{C_0+W-Q}{R'}$. The contribution of this paper is then: the modelling of the speed $R$ and the latency $L$ of a server, the introduction of the linear term $G^l$ and the extraction of the residual curve, that can be used in more contexts than the fixed priority scheduling.

### Feedback on the use of Coq

The use of Coq gave us the opportunity to fix a few small mistakes in a preliminary version of the proof of Theorem 5 and one of its hypotheses.

One of the last steps of the proof consists is showing that a value $s$ is non-negative (step 11 in Appendix A). It was claimed as an evidence, even with negative values $J_i$ of the jitters. While trying to encode this "evidence" in Coq, we realised that the current proof holds only for non-negative $J_i$ values, and the hypotheses have been updated. We do not know currently whether the property holds with negative $J_i$ values.

One step of the proof (an index permutation, step 9.c in Appendix A) was using a wrong argument, doing a confusion between values and indexes. The proof has been corrected.

Regarding the cost of the development, it can be considered reasonable as only 1400 lines of Coq code were needed[5], requiring about two person×weeks of development[6], (including above

[5] 214 lines for statements, 989 lines for proofs and 49 lines of comment (the remaining being blank lines).

[6] For a developper with a few years of experience with the tool.

mentioned proof fixes). This was made possible thanks to the availability of a formalization of the real numbers in Coq's standard library as well as the nice Mathematical Components library [15] and particularly its big operators [1] to manipulate the Σ notation for sums.

## 4.2 Coq statement of Theorem 5

While the Coq compiler checks that a theorem is well formed and that its proof is correct, it can not check that the theorem conforms to the author or reader intuition. We will then describe the formal statement of Theorem 5 in Coq's language.

The full proof is available, along with instructions to automatically recheck it with Coq, at `http://doi.org/10.5281/zenodo.4518843` .

First comes the loading of the libraries,

```
Require Import mathcomp.(*...*).
Require Import Reals (*...*).
```

and Coq is instructed to interpret all standard notations, such as $+, -, \leq$, as real number ones

```
Local Open Scope R_scope.
```

We then give the hypotheses of the theorem

```
Section Theorem3.

Variable n’ : nat.
Notation n := n’.+1.  (* Be sure that n is non zero *)

Variable R T : R+*.
Variable tC tT : R+* ^ n.
Variable tJ : R+ ^ n.
```

For convenience, the `i`-th element `(tC i)` of the `n`-tuple `tC` will then be denoted `C‘_i`

```
Notation "’C‘_’ i" := (tC i).
Notation "’T‘_’ i" := (tT i).
Notation "’J‘_’ i" := (tJ i).

Hypothesis R_large_enough : \sum_i C‘_i / T‘_i < R.
```

And we define the various constants and functions

```
Definition R’ := R − \sum_i C‘_i / T‘_i.
Definition W := \sum_i (T‘_i + J‘_i − C‘_i / R) ∗ (C‘_i / T‘_i).
Definition L := (\min_k C‘_k) ∗ ((\sum_i C‘_i / T‘_i) − \max_i (C‘_i / T‘_i)).
Definition Q :=
  \sum_(i < n) \sum_(j < n | j < i) Rmin T‘_i T‘_j ∗ (C‘_i ∗ C‘_j) / (T‘_i ∗ T‘_j).
Definition C := R ∗ T + W − Rmax L Q / R.
Definition V t := \sum_i C‘_i ∗ IZR (Zceil ((t + J‘_i) / T‘_i)).
Definition beta R T := fun t : R+ ⇒ R ∗ ’[ t − T ]+.
```

Before finally stating the theorem itself

```
Theorem theorem3 : forall t, (beta R’ (C / R’) t ≤ ’[fun t ⇒ beta R T t − V t]+^ t)%Rbar.
```

where `%Rbar` tells Coq that $\leq$ is the one on $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ since the non decreasing closure contains a least upper bound that could be infinite.

**Table 2** Periods of flows (in ms).

| Set name | S1 | S2 | S3 |
|---|---|---|---|
| Period values | 2,5,10,20,25,40,50 | 2,3,4,5,6,7,8,9,10 | 2,3,5,7,11,13 |
| lcm | 200 | 2520 | 30030 |

**Table 3** Mean computed bounds and computing time.

| Configuration | | Method | | | |
|---|---|---|---|---|---|
| Periods | Jitters | Fluid | Linear | Quadratic | Staircase |
| Mean computed bounds, per flow, in ms, and gain w.r.t. fluid modelling | | | | | |
| S1 | Null | 12.3 | 12.0 (-2%) | 10.3 (-16%) | 6.1 (-50%) |
| S1 | Rand. | 17.6 | 17.2 (-2%) | 15.5 (-11%) | 6.1 (-65%) |
| S2 | Null | 7.7 | 7.4 (-3%) | 5.7 (-25%) | 3.4 (-56%) |
| S2 | Rand. | 10.6 | 10.2 (-3%) | 8.6 (-19%) | 3.4 (-68%) |
| S3 | Null | 7.2 | 6.9 (-3%) | 5.6 (-22%) | 3.3 (-54%) |
| S3 | Rand. | 9.9 | 9.5 (-3%) | 8.2 (-17%) | 3.3 (-66%) |
| Mean computing time, per configuration, in ms<br>(and ratio w.r.t. lcm for staircase) | | | | | |
| S1 | Null | 9 | 15 | 96 | 567 (2.8) |
| S1 | Rand. | 10 | 18 | 101 | 597 (3.0) |
| S2 | Null | 6 | 7 | 26 | 5239 (2.1) |
| S2 | Rand. | 6 | 7 | 24 | 4935 (2.0) |
| S3 | Null | 6 | 6 | 21 | 51657 (1.7) |
| S3 | Rand. | 6 | 6 | 21 | 50226 (1.7) |

## 5 Evaluation

This section evaluates the quality of the approximation provided in this paper, in terms of accuracy of the result and computational cost.

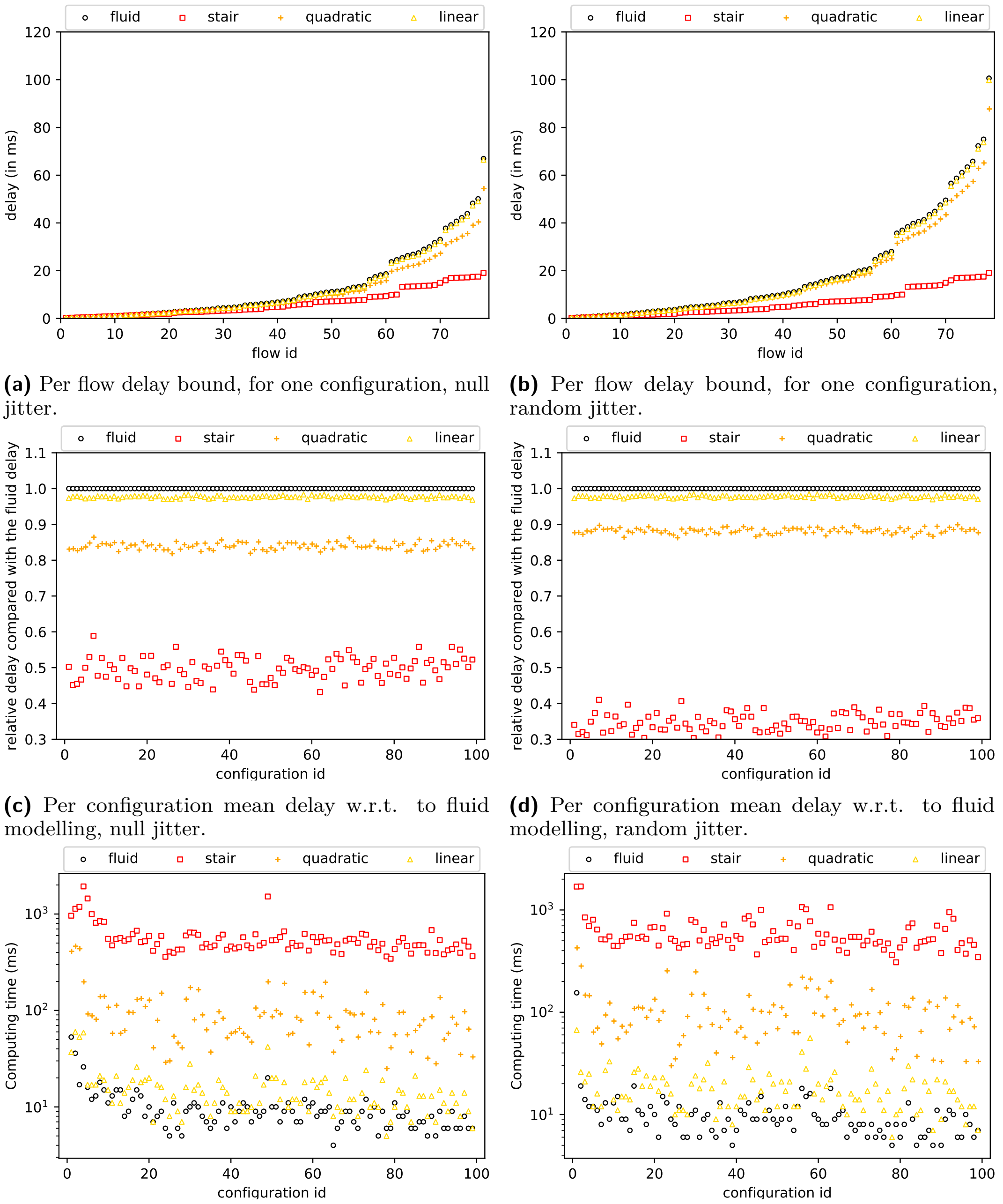
To do so, we test the expression on a large set of configurations. Each configuration represents a non-static priority server, with a constant rate of 1Mb/s, no latency, and a set of randomly generated sporadic flows. Let $c_i$ be a configuration, each flow $f_{i,j}$ has priority $j$, a fixed packet size $C_{i,j}$ chosen uniformly between 8 and 16 bytes, a period $T_{i,j}$ also randomly chosen in a subset of values, and a jitter $J_{i,j}$ also randomly chosen. New flows are added up to reaching a global load of 90%, and $n_i$ denotes the number of flows.

One hundred configurations are generated picking periods values from S1 of Table 2 and with no jitter, another hundred using set S2 and also with no jitter, and another hundred using set S3 of the same table and also no jitter. Three others sets are generated in a similar way, but with a jitter uniformly distributed between 0 and the flow period (excluded).

For each configuration $c_i$, let $f_{i,1}, \ldots, f_{i,n_i}$ be the set of flows. For each flow $f_{i,j}$, four bounds on the delay are computed using different methods. The two first have been used in the illustrative example in Section 2.3.

1. $d_{i,j}^{\mathit{fluid}} = hDev(\alpha_i, \beta_{i,j}^{\mathit{fluid}})$, where $\beta_{i,j}^{\mathit{fluid}}$ is computed using eq. (3) with $\alpha_k = \gamma_{C_k/T_k, C_k(1+J_k/T_k)}$. It is called the *fluid* modelling.
2. $d_{i,j}^{stc} = hDev(\alpha_i, \beta_{i,j}^{stc})$, where $\beta_{i,j}^{stc}$ is computed using eq. (3) with $\alpha_k = \nu_{T_k,C_k,J_k}$. It is called the *staircase* modelling.
3. $d_{i,j}^{lin} = hDev(\alpha_i, \beta_{i,j}^{lin})$ where $\beta_{i,j}^{lin}$ is computed using Theorem 5 but only with the linear term $G^l$ (*i.e.* setting $G^q = 0$). It is called the *linear* modelling.
4. $d_{i,j}^{quad} = hDev(\alpha_i, \beta_{i,j}^{quad})$ where $\beta_{i,j}^{quad}$ is computed using Theorem 5 but only with the quadratic term $G^q$ (*i.e.* setting $G^l = 0$). It is called the *quadratic* modelling.

Experiments have run on a laptop with 4GB of memory and a 2.7GHz Intel Core i5.



**(a)** Per flow delay bound, for one configuration, null jitter.

**(b)** Per flow delay bound, for one configuration, random jitter.

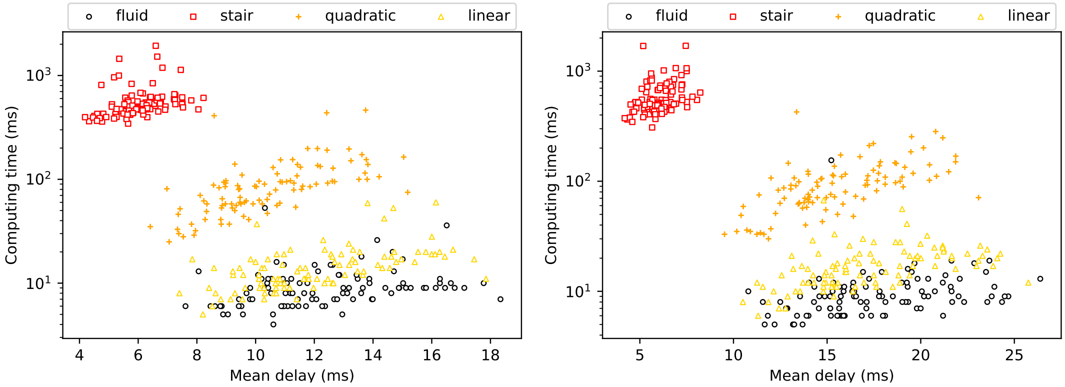
**(c)** Per configuration mean delay w.r.t. to fluid modelling, null jitter.

**(d)** Per configuration mean delay w.r.t. to fluid modelling, random jitter.

**(e)** Per configuration computing time, null jitter.

**(f)** Per configuration computing time, random jitter .

**Figure 6** Plots related to configurations with periods in S1 set, null w.r.t. random jitters.



**(a)** Null jitter.

**(b)** Random jitter.

**Figure 7** Computing time per mean delay, for configurations with periods in S1 set.

Figure 6a plots, for a given configuration $c_k$ with periods chosen in S1 (harmonic periods) and no jitter, the bounds $d^{fluid}_{k,j}$, $d^{stc}_{k,j}$, $d^{quad}_{k,j}$ $d^{lin}_{k,j}$ computed by the four methods for each flow. Since flows are sorted by priority, the plots are non decreasing. As expected, the fluid modelling gives the larger, *i.e.* worse, bounds, whereas the linear approximation is smaller, the quadratic approximation even smaller, and staircase modelling leads to the smallest bounds. Only one configuration is plotted, but they all have the same shape.

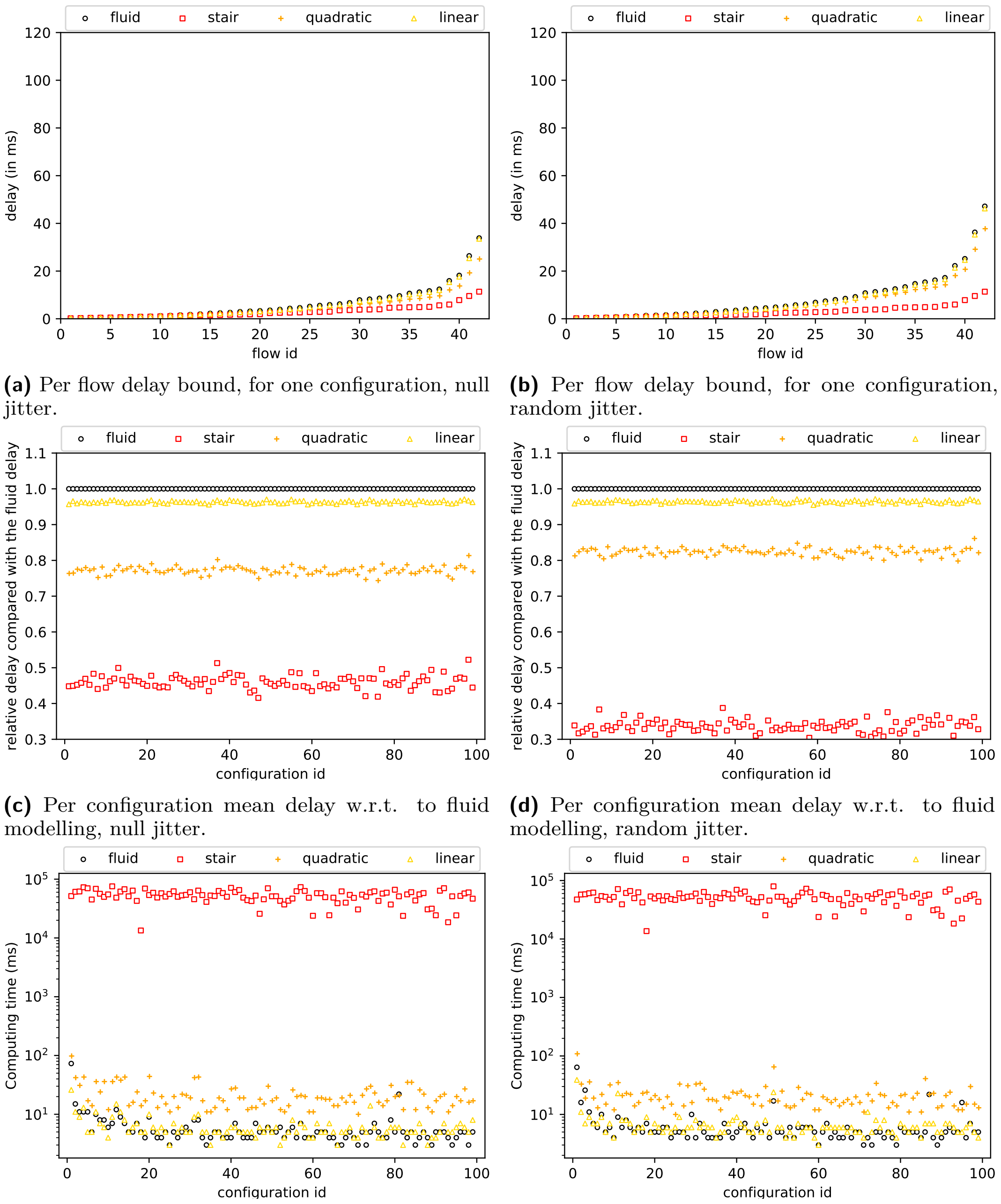
Now considering the fluid modelling as the reference value, Figure 6c plots, for each configuration with harmonic periods, the sum of all bounds computed by a method divided by the sum of all bounds computed by fluid modelling: $\frac{\sum_{j=0}^{n_i} d^X_{i,j}}{\sum_{j=0}^{n_i} d^{fluid}_{i,j}}$ with $X \in \{\mathit{fluid}, \mathit{stc}, \mathit{lin}, \mathit{quad}\}$. Figure 6e plots the computation time required to analyse each configuration, depending on the modelling, with a log-scale on time axis. Last, Figure 7 plots the computation time as a function of the mean delay, for each configuration and each method.

In the same figure group are also plotted the same graphs but considering the jitter of each flow picked up between 0 and the flow period. As expected, the jitter increases the delay of the affine models, but has no influence on the staircase one (cf. Figure 6b). Then, the gain obtained by the staircase model w.r.t. the fluid model increases, whereas the gain of the quadratic model is less (12% instead of 16%).

The Table 3 summarises, for each set of hundred configurations, the mean bound on delays for all flows, and the mean computing time for a single configuration. For the staircase modelling is added this computation time divided by the lcm of the periods, showing that this computation time is almost linear w.r.t. this lcm.

The same kind of information is plotted in the group of Figures 8 when the periods are taken from the set S3. The relations between the methods in terms of accuracy of results are in the same order of magnitude (from 16% to 22% without jitter, from 11% to 17% with jitter), but the computation time of the staircase methods is three orders of magnitude larger (50s vs. 21ms).

The results for the set S2 are not plotted but are summarised in Table 3.



**(a)** Per flow delay bound, for one configuration, null jitter.

**(b)** Per flow delay bound, for one configuration, random jitter.

**(c)** Per configuration mean delay w.r.t. to fluid modelling, null jitter.

**(d)** Per configuration mean delay w.r.t. to fluid modelling, random jitter.

**(e)** Per configuration computing time, null jitter.

**(f)** Per configuration computing time, random jitter .

**Figure 8** Plots related to configurations with periods in S3 set, null w.r.t. random jitters.

## 6 Conclusion

In network calculus, the computation of residual services with staircase arrival curves has exponential complexity, whereas fluid arrival curves offer a linear complexity but give larger, *i.e.* worse, upper bounds.

This paper generalises a result from [2], and develops a residual service curve with either linear or quadratic computational complexity. The correctness of the result is enforced by providing a formal Coq proof. The different approaches are evaluated on 600 systems with sporadic workload and non-preemptive static priority scheduling.

Whereas the staircase model computes bounds that are half of those of the fluid model[7], at the expense of a computation time from $10^2$ to $10^4$ times larger, the quadratic approach already enhances the results by about 20% while being only 10 times slower. The linear model offers a limited enhancement (2%-5%). Having accurate results in short computation times helps real-time system designers when exploring several configurations (in design space exploration). A comparison with prefix-based approach [17, 18, 27] is left to further studies.

Moreover, the analytic formula of the residual service curves opens some opportunities. First, having a residual curve allows to use the Pay Burst Only Once principle, to compute an end-to-end network delay smaller than the sum of per switch delays. Second, a closed form formula gives opportunities for optimisation. Third, getting rid of least common multiple allows the use of directed rounding floating-point arithmetic that could lower the computation cost by one or two additional orders of magnitude.

The formalization of the main result of the paper using Coq enabled to fix some glitches and reach a very high level of confidence in this result. This was done at a moderate extra cost and follows the direction impulsed by the call for action[8] “Real Proofs for Real Time: Let’s do better than “almost right” at ECRTS 2016 [14].

## References

1 Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

2 Enrico Bini, Andrea Parri, and Giacomo Dossena. A quadratic-time response time upper bound with a tightness property. In *Proc. of the 36th IEEE Real-Time Systems Symposium (RTSS 2015)*, San Antonio, USA, December 2015.

3 Luca Bisti, Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. DEBORAH: a tool for worst-case analysis of FIFO tandems. In *Proc. of the 4th Int. Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010)*, LNCS. Springer, 2010.

4 Steffen Bondorf and Jens B. Schmitt. The DiscoDNC v2 – a comprehensive tool for deterministic network calculus. In *Proc. of the 8th Int. Conf. on Performance Evaluation Methodologies and Tools (VALUETOOLS 2014)*, 2014.

5 Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus – From theory to practical implementation.* Wiley, 2018.

6 Anne Bouillard and Éric Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 18(1):3–49, October 2008.

7 Marc Boyer and Christian Fraboul. Tightening end to end delay upper bound for AFDX network with rate latency FCFS servers using network calculus. In *Proc. of the 7th IEEE Int. Workshop on Factory Communication Systems Communication in Automation (WFCS 2008)*, pages 11–20. IEEE industrial Electrony Society, May 21–23 2008.

[7] It must be mentioned that a previous study on a realistic avionic configuration, based on Ethernet and 2 priority levels, has shown a gain related to staircase of only 6% [8] and another on a more loaded configuration gave a gain of 18% [10]. But these realistic configurations had lower load.

[8] A similar impulsion was given a decade ago in the programming language community and a number of mechanized formalisations (using either Coq or other tools) now appear each year at their main conference POPL.

8 Marc Boyer, Jörn Migge, and Marc Fumey. PEGASE, a robust and efficient tool for worst case network traversal time. In *Proc. of the SAE 2011 AeroTech Congress & Exhibition*, Toulouse, France, 2011.

9 Marc Boyer, Jörn Migge, and Nicolas Navet. An efficient and simple class of functions to model arrival curve of packetised flows. In *Proc. of the 1st Int. Workshop on Worst-Case Traversal Time (WCTT'2011)*, pages 43–50. ACM, 2011.

10 Marc Boyer, Nicolas Navet, and Marc Fumey. Experimental assessment of timing verification techniques for AFDX. In *Proc. of the 6th Int. Congress on Embedded Real Time Software and Systems*, Toulouse, France, 2012.

11 Felipe Cerqueira, Felix Stutz, and Björn B. Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *Proc. of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, pages 273–284, Toulouse, France, July 5–8 2016. doi:10.1109/ECRTS.2016.28.

12 Cheng-Shang Chang. *Performance Guarantees in communication networks.* Telecommunication Networks and Computer Systems. Springer, 2000.

13 The Coq development team. *The Coq proof assistant reference manual*, 2019. Version 8.11. URL: https://coq.inria.fr.

14 Christian Fraboul and Nathan Fisher, editors. *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016.* IEEE Computer Society, 2016. URL: https://ieeexplore.ieee.org/xpl/conhome/7557819/proceeding.

15 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008. URL: http://hal.inria.fr/inria-00258384.

16 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

17 Nan Guan and Wang Yi. Finitary real-time calculus: Efficient performance analysis of distributed embedded systems. In *Proc. or the IEEE 34th Real-Time Systems Symposium (RTSS'2013)*, pages 330–339. IEEE, 2013.

18 Kai Lampka, Steffen Bondorf, and Jens Schmitt. Achieving efficiency without sacrificing model accuracy: Network calculus on compact domains. In *Proc. of the 24th IEEE Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2016)*, 2016.

19 Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, March 2012.

20 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus*, volume 2050 of *LNCS*. Springer Verlag, 2001. http://lrcwww.epfl.ch/PS_files/NetCal.htm.

21 E. Le Corronc, B. Cottenceau, and L. Hardouin. Container of (min,+)-linear systems. *Journal of Discrete Event Dynamic Systems*, 14(1):15–52, March 2014.

22 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

23 Nicolas Navet, Tieu Long Mai, and Jörn Migge. Using machine learning to speed up the design space exploration of Ethernet TSN networks. Technical Report 10993/38604, University of Luxembourg, January 2019.

24 Nicolas Navet, Jörn Migge, Josetxo Villanueva, and Marc Boyer. Pre-shaping bursty transmissions under IEEE802.1Q as a simple and efficient QoS mechanism. *SAE Int. Journal of Passenger Cars—Electronic and Electrical System*, 11(3), 2018.

25 Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. SCED: A generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM transactions on networking*, 7(5):669–684, October 1999.

26 Jens B. Schmitt and Frank A. Zdarsky. The DISCO network calculator - a toolbox for worst case analysis. In *Proc. of the First International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'06), Pisa, Italy*. ACM, 2006.

27 Urban Suppiger, Simon Perathoner, Kai Lampka, and Lothar Thiele. A simple approximation method for reducing the complexity of modular performance analysis. TIK-Report 329, Computer Engineering and Networks Laboratory – Swiss Federal Institute of Technology (ETH), August 2010.

28 Ernesto Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, ETH Zurich, September 2006.

29 Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006. URL: `http://www.mpa.ethz.ch/Rtctoolbox`.

## A Proof of Theorem 5

Regarding only correctness issues, we may have omitted this section since the Coq proof already provides a formal correctness insurance. Nevertheless, this section can be considered as the documentation of the Coq proof. But the main justification of this section relies in the opportunity to adapt or generalise the results. The same way as we have converted and extended the result on response time presented in [2] by a study of its proof, we provide a human-oriented proof, as a complement of the formal Coq proof.

The proof presentation is inspired by [19]. Each sub-step of the proof will start with some ordering value, followed by the statement of the sub-step, using bold font. Thereafter will come the proof of the sub-step itself.

For the proof, let $V \overset{def}{=} \sum_{i=1}^{n} \nu_{T_i,C_i,J_i}$ *i.e.* $V(t) = \sum_{i=1}^{n} C_i \left\lceil \frac{t+J_i}{T_i} \right\rceil$, and $\rho = \sum_{i=1}^{n} \frac{C_i}{T_i}$ the long term rate of $V$, and recall that $\rho < R$.

1. **Definitions of $s^M$ and first properties**: For any $M \in \mathbb{R}$, let

$$s^M \overset{def}{=} \min \left\{ t \in \mathbb{R} \mid V(t) + M = R(t - L) \right\}. \tag{10}$$

This $s^M$ is the minimal solution to $V(t) + M = R(t - L)$. The first step consists in showing that $s^M$ exists (there are solutions, and there exists a minimal one), and the second on their relative positions (cf. Figure 9).

a. **The minimum exists:** By definition of the ceiling function, $x \le \lceil x \rceil < x + 1$. Then, for any $i$, $t\frac{C_i}{T_i} + J_i\frac{C_i}{T_i} \le C_i \left\lceil \frac{t+J_i}{T_i} \right\rceil < t\frac{C_i}{T_i} + J_i\frac{C_i}{T_i} + C_i$. Making the sum for all $i \in [1, n]$

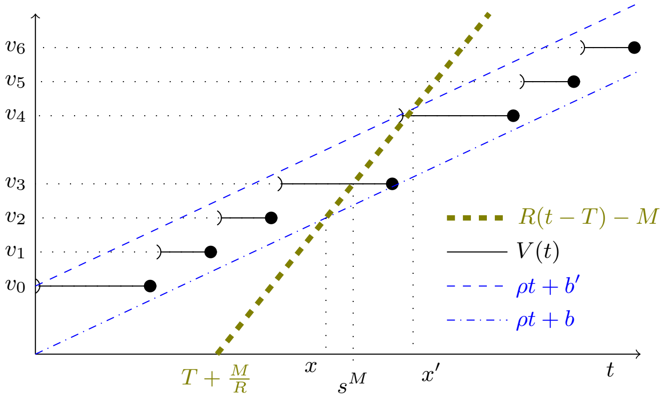


**Figure 9** Illustration of $s^M$ definition, with $n = 2$, $C_1 = C_2 = \frac{1}{2}$, $T_1 = 2$, $T_2 = 3$, $J_1 = J_2 = 0$.

leads to

$$\forall t \in \mathbb{R} : \rho t + b \leq V(t) < \rho t + b', \tag{11}$$

with $b = \sum_{i=1}^{n} J_i \frac{C_i}{T_i}$, $b' = b + \sum_{i=1}^{n} C_i$.
These are affine functions, and since $R > \rho$, there exists $x < x'$ such that $\rho x + b = R(x - L) - M$ and $\rho x' + b' = R(x' - L) - M$ (cf. Figure 9). Set $y = \rho x + b$, $y' = \rho x' + b'$. From eq. 11, for any $t \in [x, x'] : y \leq V(t) \leq y'$. Set $Y = \{V(t) \mid t \in [x, x']\}$ the set of values of $V$ on $[x, x']$. This set is non-empty and finite. If $(v_i)_{i\in\mathbb{N}}$ is the ordered set of values of the function $V$, there exists $k \leq k'$ such that $Y = \{v_k, v_{k+1}, \ldots, v_{k'}\}$ ($Y = \{v_3, v_4\}$ on the example in Figure 9), and to each one corresponds one $s_k^M$ such that $v_k = R(s_k^M - L) - M$. Then the set $\{t \in \mathbb{R} \mid V(t) + M = R(t - T)\} = \{s_k^M, \ldots, s_{k'}^M\}$ is non empty and finite, and its minimum, $s^M$ exists.

b. **Before $s^M$, $R(t - L) - M$ is below $V(t)$ *i.e.***

$$\forall t < s^M : R(t - L) - M < V(t)\textbf{:} \tag{12}$$

By contradiction, assume there exists $t < s^M$ such that $R(t-L)-M \geq V(t)$. The case $R(t-L)-M = V(t)$ leads to $t \geq s^M$ by definition of $s^M$. In case of $R(t-L)-M > V(t)$, then $R(t - L) - M > \rho t$, so $t > x$.

c. **A lower bound on $s^M$:** In step 1a, $x$ has been defined such that $s^M \in [x, x']$, with $\rho x + b = R(x - L) - M$, then

$$s^M \geq x = \frac{M + RL + \sum_{i=1}^{n} J_i \frac{C_i}{T_i}}{R'}. \tag{13}$$

This relation will be used in one of the last step of the proof.

2. **Definitions of $q_i^M, r_i^M$ and first properties:** Let introduce for any $i \in [\![1, n]\!]$,

$$q_i^M \overset{def}{=} \left\lceil \frac{s^M + J_i}{T_i} \right\rceil, \qquad r_i^M \overset{def}{=} T_i q_i^M - (s^M + J_i) \tag{14}$$

keep in mind that $q_i^M = \frac{s^M + J_i + r_i^M}{T_i}$ and that $T_i > r_i^M \geq 0$ (from $\frac{x}{L} \leq \left\lceil \frac{x}{L} \right\rceil < \frac{x}{L} + 1$ comes $0 \leq L\left\lceil \frac{x}{L} \right\rceil - x < L$, and setting $x = s^M + J_i$).

3. **Expression of $s^M$ in terms of $r_i^M$:** Since $s^M$ is a minimum, it satisfies $R(s^M - L) = V(s^M) + M$ *i.e.*

$$R(s^M - L) = \sum_{i=1}^{n} C_i \left\lceil \frac{s^M + J_i}{T_i} \right\rceil + M = \sum_{i=1}^{n} C_i q_i^M + M \tag{15}$$

$$= \sum_{i=1}^{n} \frac{C_i}{T_i}(s^M + J_i + r_i^M) + M \tag{16}$$

$$\iff s^M(R - \sum_{i=1}^{n} \frac{C_i}{T_i}) = M + RL + \sum_{i=1}^{n} \frac{C_i}{T_i}(J_i + r_i^M) \tag{17}$$

$$\iff R' s^M = M + RL + \sum_{i=1}^{n} \frac{C_i}{T_i} J_i + \sum_{i=1}^{n} \frac{C_i}{T_i} r_i^M \tag{18}$$

4. **Two definitions for a reordering $l_i^M$ and $\sigma$:**

$$\forall i \in [\![1, n]\!] : l_i^M \overset{def}{=} (q_i^M - 1)T_i - J_i. \tag{19}$$

Remark that $s^M > l_i^M$ (from $0 \leq r_i^M < T_i$ comes $0 \leq T_i q_i^M - (s^M + J_i) < T_i$ and $T_i(q_i^M - 1) - J_i < s^M$).

Now, let $\sigma : [\![1, n]\!] \to [\![1, n]\!]$ be a permutation such that the sequence $l_{\sigma(i)}^M$ is non-increasing, *i.e.* $s^M > l_{\sigma(1)}^M \geq l_{\sigma(2)}^M \geq \cdots \geq l_{\sigma(n)}^M$.

5. **Forall $\boldsymbol{k \in [\![1, n]\!]}$ it holds**

$$C_k \left\lceil \frac{s^M + J_k}{T_k} \right\rceil = C_k \left\lceil \frac{l_k^M + J_k}{T_k} \right\rceil + C_k \tag{20}$$

By definition $q_k^M$ is an integer, so $q_k^M - 1$ also is and $q_k^M = \lceil q_k^M - 1 \rceil + 1$. By definition of $l_k^M$, we then get $q_k^M = \left\lceil \frac{l_k^M + J_k}{T_k} \right\rceil + 1$ hence the result by definition of $q_k^M$.

6. **Forall** $i \in [\![1, n]\!] : s^M \geq l_{\sigma(i)}^M + \sum_{k=1}^{i} \frac{C_{\sigma(k)}}{R}$ **:**

Let $i \in [\![1, n]\!]$, $\mathbf{S}_i = \{\sigma(1), \ldots, \sigma(i)\}$ and $\bar{\mathbf{S}}_i = [\![1, n]\!] \setminus \mathbf{S}_i$.

From previous relation, for any $k$ in $\mathbf{S}_i$, $C_k \left\lceil \frac{s^M + J_k}{T_k} \right\rceil \geq C_k \left\lceil \frac{l_k^M + J_k}{T_k} \right\rceil + C_k$. But by definition, $(l_{\sigma(m)}^M)_{m \in [\![1,n]\!]}$ is non-increasing sequence, so for all $k \in \mathbf{S}_i$, $l_k^M \geq l_{\sigma(i)}^M$, which yields $C_k \left\lceil \frac{l_k^M + J_k}{T_k} \right\rceil + C_k \geq C_k \left\lceil \frac{l_{\sigma(i)}^M + J_k}{T_k} \right\rceil + C_k$. To conclude

$$\forall k \in \mathbf{S}_i : C_k \left\lceil \frac{s^M + J_k}{T_k} \right\rceil \geq C_k \left\lceil \frac{l_{\sigma(i)}^M + J_k}{T_k} \right\rceil + C_k \tag{21}$$

Now, consider $k \in \bar{\mathbf{S}}_i$. By the definition of $l_j^M$ (cf. proof step 4), $\forall j \in [\![1, n]\!] : s^M > l_j^M$, and in particular, for $j = \sigma(i)$. Then, it holds

$$\forall k \in \bar{\mathbf{S}}_i : C_k \left\lceil \frac{s^M + J_k}{T_k} \right\rceil \geq C_k \left\lceil \frac{l_{\sigma(i)}^M + J_k}{T_k} \right\rceil \tag{22}$$

Summing over eq. (21) and eq. (22), it comes

$$\sum_{k \in \mathbf{S}_i \cup \bar{\mathbf{S}}_i} C_k \left\lceil \frac{s^M + J_k}{T_k} \right\rceil \geq \sum_{k \in \mathbf{S}_i \cup \bar{\mathbf{S}}_i} C_k \left\lceil \frac{l_{\sigma(i)}^M + J_k}{T_k} \right\rceil + \sum_{k \in \mathbf{S}_i} C_k$$

$$\textit{i.e. } V(s^M) \geq V(l_{\sigma(i)}^M) + \sum_{j=1}^{i} C_{\sigma(j)}$$

By the definition of $s^M$, one has $V(s^M) = R(s^M - L) - M$. Conversely, since $s^M \geq l_{\sigma(i)}^M$, from eq. 12, it comes $V(l_{\sigma(i)}^M) \geq R(l_{\sigma(i)}^M - L) - M$, so

$$R(s^M - L) - M \geq R(l_{\sigma(i)}^M - L) - M + \sum_{j=1}^{i} C_{\sigma(j)} \tag{23}$$

$$\iff s^M \geq l_{\sigma(i)}^M + \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R} \tag{24}$$

7. **Forall** $i \in [\![1, n]\!] : r_{\sigma(i)}^M \leq T_{\sigma(i)} - \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R}$ **:** This is a direct consequence of definition

of $l_i^M$, $q_i^M$ and previous relation.

$$s^M \geq l^M_{\sigma(i)} + \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R} \tag{25}$$

$$\iff s^M \geq (q^M_{\sigma(i)} - 1)T_{\sigma(i)} - J_{\sigma(i)} + \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R} \tag{26}$$

$$\iff s^M + J_{\sigma(i)} - T_{\sigma(i)} q^M_{\sigma(i)} \geq -T_{\sigma(i)} + \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R} \tag{27}$$

$$\iff r^M_{\sigma(i)} \leq T_{\sigma(i)} - \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R} \tag{28}$$

**8.** $\sum_{i=1}^{n} r_i^M \frac{C_i}{T_i} \leq \sum_{i=1}^{n} \left( T_i - \frac{C_i}{R}\right) \frac{C_i}{T_i} - \frac{1}{R} \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(i)} C_{\sigma(j)}}{T_{\sigma(i)}}$ **:**

$$\sum_{i=1}^{n} r_i^M \frac{C_i}{T_i} = \sum_{i=1}^{n} r^M_{\sigma(i)} \frac{C_{\sigma(i)}}{T_{\sigma(i)}} \qquad \text{since } \sigma \text{ is a permutation} \tag{29}$$

$$\leq \sum_{i=1}^{n} \left( T_{\sigma(i)} - \sum_{j=1}^{i} \frac{C_{\sigma(j)}}{R} \right) \frac{C_{\sigma(i)}}{T_{\sigma(i)}} \tag{30}$$

$$= \sum_{i=1}^{n} \left( T_{\sigma(i)} - \frac{C_{\sigma(i)}}{R} - \sum_{j=1}^{i-1} \frac{C_{\sigma(j)}}{R} \right) \frac{C_{\sigma(i)}}{T_{\sigma(i)}} \tag{31}$$

$$= \sum_{i=1}^{n} \left( T_{\sigma(i)} - \frac{C_{\sigma(i)}}{R} \right) \frac{C_{\sigma(i)}}{T_{\sigma(i)}} - \frac{1}{R} \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(j)} C_{\sigma(i)}}{T_{\sigma(i)}} \tag{32}$$

$$= \sum_{i=1}^{n} \left( T_i - \frac{C_i}{R} \right) \frac{C_i}{T_i} - \frac{1}{R} \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(j)} C_{\sigma(i)}}{T_{\sigma(i)}} \tag{33}$$

The next step consists in having a lower bound on $\sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(j)} C_{\sigma(i)}}{T_{\sigma(i)}}$.

**9.** $\sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(j)} C_{\sigma(i)}}{T_{\sigma(i)}} \geq \max\left\{G^q, G^l\right\}$ **The goal in this step is to get rid of the $\sigma$ permutation, since it depends on $M$. :**

**a.** $\sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(j)} C_{\sigma(i)}}{T_{\sigma(i)}} \geq G^l$ **:**

$$\sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(i)} C_{\sigma(j)}}{T_{\sigma(i)}} \geq \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{C_{\sigma(i)} \min_{k \in [\![1,n]\!]} C_k}{T_{\sigma(i)}} \tag{34}$$

$$= \min_{k \in [\![1,n]\!]} C_k \sum_{i=1}^{n} \frac{C_{\sigma(i)}}{T_{\sigma(i)}} \times (i-1) \tag{35}$$

$$\geq \min_{k \in [\![1,n]\!]} C_k \sum_{i=1}^{n} \frac{C_{\sigma(i)}}{T_{\sigma(i)}} \times \min(i-1, 1) \tag{36}$$

and since one does not know the value of $\sigma(1)$ (*i.e.* when $i - 1 = 0$)

$$\geq \min_{k \in [\![1,n]\!]} C_k \left( \sum_{i=1}^{n} \frac{C_i}{T_i} - \max_{i \in [\![1,n]\!]} \frac{C_i}{T_i} \right) = G^l \tag{37}$$

**b.** $\sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}} \geq \sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}T_{\sigma(j)}} \min\left\{T_{\sigma(i)}, T_{\sigma(j)}\right\}$ **:**

$$\sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}} = \sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}T_{\sigma(j)}}T_{\sigma(j)} \tag{38}$$

$$\geq \sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}T_{\sigma(j)}} \min\left\{T_{\sigma(i)}, T_{\sigma(j)}\right\} \tag{39}$$

$$\tag{40}$$

**c.** $\sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}T_{\sigma(j)}} \min\left\{T_{\sigma(i)}, T_{\sigma(j)}\right\} = \sum_{p=1}^{n}\sum_{q=1}^{p-1}\frac{C_pC_q}{T_pT_q}\min\{T_p, T_q\} = G^q$**:**

Let $x_{i,j} \overset{def}{=} \frac{C_iC_j}{T_iT_j}\min\{T_i, T_i\}$, and $X \overset{def}{=} \left\{(i,j) \in [\![1,n]\!]^2 \mid i > j\right\}$, and also

$$\begin{aligned} h : [\![1,n]\!]^2 &\to [\![1,n]\!]^2 \\ (i,j) &\mapsto \begin{cases} (\sigma(i), \sigma(j)) \text{ when } (i > j) = (\sigma(i) > \sigma(j)) \\ (\sigma(j), \sigma(i)) \text{ otherwise.} \end{cases} \end{aligned}$$

Note that for all $i, j$, we have $(i,j) \in X$ if and only if $h(i,j) \in X$ and $x_{\sigma(i),\sigma(j)} = x_{h(i,j)}$ since $x$ is symmetric. We thus have

$$\sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}T_{\sigma(j)}} \min\left\{T_{\sigma(i)}, T_{\sigma(j)}\right\} = \sum_{(i,j)\in X} x_{(\sigma(i),\sigma(j))} = \sum_{h(i,j)\in X} x_{h(i,j)} \tag{41}$$

One can then prove that $h$ is injective, meaning it is bijective, which enables the following reindexing

$$\sum_{h(i,j)\in X} x_{h(i,j)} = \sum_{(i,j)\in X} x_{(i,j)} = G^q \tag{42}$$

**10.** $\boldsymbol{s^M \leq \frac{M+C}{R'}}$**:** This is just, going from equations (18), application of steps 8 and 9.

$$\begin{aligned} R's^M &= M + RL + \sum_{i=1}^{n}\frac{C_i}{T_i}J_i + \sum_{i=1}^{n}\frac{C_i}{T_i}r_i^M \\ &\leq M + RL + \sum_{i=1}^{n}\frac{C_i}{T_i}J_i + \sum_{i=1}^{n}\left(T_i - \frac{C_i}{R}\right)\frac{C_i}{T_i} - \frac{1}{R}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\frac{C_{\sigma(i)}C_{\sigma(j)}}{T_{\sigma(i)}} \\ &\leq M + RL + \sum_{i=1}^{n}\left(J_i + T_i - \frac{C_i}{R}\right)\frac{C_i}{T_i} - \frac{\max\left\{G^q, G^l\right\}}{R} \\ \implies s^M &\leq \frac{M+C}{R'} \end{aligned}$$

**11. Here comes the $\boldsymbol{M}$ elimination:** Let $t \in \mathbb{R}^+$.
If $t \leq \frac{C}{R'}$, $\beta_{R',\frac{C}{R'}}(t) = 0$, so $\beta_{R',\frac{C}{R'}}(t) \leq [\beta_{R,L} - V]_{\uparrow}^{+}(t)$ trivially holds.
If $t \geq \frac{C}{R'}$. By definition of $s^M$, for any $M \in \mathbb{R}$,

$$M = R(s^M - L) - V(s^M) \tag{43}$$

so, for any interval $I^M$ such that $s^M \in I^M$

$$M \leq \sup_{u\in I^M}\left\{R(u-L) - V(u)\right\}. \tag{44}$$

Set $M = R't - C$ (this can be done safely since there is no hidden $M$ in $R, R', L, V(\cdot)$). From step 10, $s^M \leq \frac{M+C}{R'} = t$, so

$$R't - C \leq \sup_{s^M \leq u \leq t} \left\{ R(u - L) - V(u) \right\}. \tag{45}$$

But from $t \geq \frac{C}{R'}$ and $M = R't - C$ comes $M \geq 0$, and introducing it in eq. 13 yields $s^M \geq 0$, so

$$R't - C \leq \sup_{0 \leq u \leq t} \left\{ R(u - L) - V(u) \right\}, \tag{46}$$

and by doing the maximum with 0

$$R' \left[ t - \frac{C}{R'} \right]^+ \leq \sup_{0 \leq u \leq t} \left[ R(u - L) - V(u) \right]^+ \tag{47}$$

$$\iff \beta_{R', \frac{C}{R'}}(t) \leq \left[ \beta_{R,L} - V \right]^+_{\uparrow}(t) \tag{48}$$

# Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses

Nils Vreman
Lund University, Department of Automatic Control, Sweden

Anton Cervin
Lund University, Department of Automatic Control, Sweden

Martina Maggio
Universität des Saarlandes, Department of Computer Science, Saarbrücken, Germany
Lund University, Department of Automatic Control, Sweden

## Abstract

Control systems are by design robust to various disturbances, ranging from noise to unmodelled dynamics. Recent work on the weakly hard model – applied to controllers – has shown that control tasks can also be inherently robust to deadline misses. However, existing exact analyses are limited to the stability of the closed-loop system. In this paper we show that stability is important but cannot be the only factor to determine whether the behaviour of a system is acceptable also under deadline misses. We focus on systems that experience bursts of deadline misses and on their recovery to normal operation. We apply the resulting comprehensive analysis (that includes both stability and performance) to a Furuta pendulum, comparing simulated data and data obtained with the real plant. We further evaluate our analysis using a benchmark set composed of 133 systems, which is considered representative of industrial control plants. Our results show the handling of the control signal is an extremely important factor in the performance degradation that the controller experiences – a clear indication that only a stability test does not give enough indication about the robustness to deadline misses.

**2012 ACM Subject Classification** Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time systems; Computer systems organization → Dependable and fault-tolerant systems and networks

**Keywords and phrases** Fault-Tolerant Control Systems, Weakly Hard Task Model

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.15

**Funding** The authors are members of the ELLIIT Strategic Research Area at Lund University. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Number 871259 (ADMORPH project). This publication reflects only the authors' view and the European Commission is not responsible for any use that may be made of the information it contains.

## 1 Introduction

Feedback control systems have been used as prime examples of hard real-time systems ever since the term was coined. However, in the past twenty years, it has become increasingly clear that the hard real-time task model is overly strict for most control systems. Requiring that *all* deadlines of a periodic control task must be met can lead to very conservative designs with low utilisation, low sampling rates, and – in the end – worse than necessary control performance. Following this line of reasoning, researchers started looking into task models in which tasks can sporadically miss some deadlines, and defined concepts like the "skip factor" [45], i.e., the number of correctly executed jobs that must occur between two failed instances. Task models with failed jobs eventually led to the definition of the weakly hard task model [11], that specify constraints on the sequence of jobs that complete their

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 15; pp. 15:1–15:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

execution correctly and the ones that miss their deadlines. Adopting the weakly hard model allows a control task to opportunistically execute more frequently, which in general improves reference tracking and disturbance rejection [41, 46, 55].

A recent industrial survey has shown that practitioners are used to work with systems that experience deadline misses [5, Questions 14 and 15]. In a significant percentage of cases, these systems are subject to blackout events that can persist for more than ten consecutive task periods. Examples of such events are mode switches in mixed-criticality systems, resets due to hardware faults, security attacks, specific types of cache misses, and connectivity issues in networked control systems. Handling all of these situations by design could require extreme resource over-provisioning.

In this paper we focus precisely on these sporadic system events, which may cause a control task to stall for one or several cycles. To determine the effect of deadline misses on the control system, it is of utmost importance to analyse the physics of the plant and the effect of control signals not being delivered to it. For these systems, stability guarantees have been given on the maximum number of tolerable consecutive deadline misses [48]. These guarantees only consider *stability* of the closed-loop system as the property to be preserved. In this paper, we demonstrate that while stability may be preserved, the control system *performance* may be severely affected by the burst of misses. Performance and stability have been considered simultaneously in the literature. For example, in [30] a controller is developed that guarantees stability, accepting some level of performance degradation for a given plant. However, we believe that a lot is left open to investigate, especially with respect to general guarantees. In particular, in this paper we aim to understand the effect that the deadline handling strategies jointly have on performance and stability, providing a holistic evaluation. Furthermore, we evaluate our results on both simulated platforms and real control plants. More precisely, we offer the following contributions:

- We propose a new type of weakly hard task model, which specifies a *consecutive* deadline miss interval followed by a minimum *consecutive* deadline hit (recovery) interval. This model is crucial to properly assess the performance effect of a burst of deadline misses, as the ones reported by practitioners [5].
- We provide an analysis methodology for stability and performance of control tasks executing under this task model using a variety of implementation choices to handle deadline misses (Kill vs. Skip-Next, Zero vs. Hold). In particular, we separately consider the two cases in which a miss pattern is repeated (which fits an increased workload situation – for example due to a different mode of execution), and in which it is not possible to specify constraints on the repetition of the miss pattern.
- We compare experimental results obtained with a real process – a Furuta pendulum that is stabilised in the upright position – with simulation results based on a linear model of the same process, using the same controller. This shows that simulated data is representative enough to draw conclusions on the controller performance, despite unmodelled nonlinear dynamics and noise.
- We present the result of a large scale evaluation campaign of commonly used controllers on a benchmark of 133 industrial plants. From this evaluation we conclude that the choice of actuation strategy (i.e., what to do with the control signal when a miss occurs) affects control performance significantly more than the choice of deadline handling strategy (i.e., what to do with the control task when a miss occurs).

The rest of this paper is outlined as follows. In Section 2 we give a brief overview of related work. In Section 3 we present relevant control theory and introduce the stability and performance concepts. Section 4 describes the weakly hard task models and the strategies that are commonly used to handle deadline misses. Section 5 presents our extension to the weakly hard task model, and the corresponding stability and performance analysis. Section 6 presents our experimental results, and Section 7 concludes the paper.

## 2 Related Work

The work presented in this paper is closely related to two broad research areas, namely, the analysis of

**(i)** weakly hard systems and

**(ii)** fault-tolerant control systems.

**Weakly Hard Systems.** Deadline misses can be seen as sporadic events caused by unforeseen delays in the system. Such delays could for instance be induced by overload activations [36,64] or cache misses [6,22]. The idea behind weakly hard analysis is that deadline misses are permitted under predefined constraints. Such systems have been analysed extensively from a real-time scheduling perspective [10,15,21,37]. The weakly hard models have gained traction in the research community as a tool to understand and analyse systems with sporadic faults [4,12,13,26,29,35,38,55,59–61]. In a recent paper, Gujarati et al. [33] analysed and compared different methods for estimating the overall reliability of control systems using the weakly hard task model. Furthermore, the authors of [50] proposed a toolchain for analysing the strongest, satisfied weakly hard constraints as a function of the worst-case execution time.

**Fault-Tolerant Control Systems.** Real-time systems are sensitive to faults. Due to their safety-critical nature, it is arguably more important to guarantee fault-tolerance with respect to other classes of systems. Some of these faults can be described using the weakly hard model. Due to the nature of control systems, special analysis techniques can combine fault models and the physical characteristics of systems.

Fault-tolerance has been investigated in many of its aspects, e.g., fault-aware scheduling algorithms [16,23] and the analysis of systems with unreliable components [43]. Furthermore, restart-based design [1,2] has been used as a technique to guarantee resilience. The fault models are frequently assumed to target overload-prone systems, or systems with components subject to sporadic failures. Bursts of faults have been observed to affect real systems [20,63]. Gujarati et al. [32] proposed an analysis method for networked control systems that uses active replication and quantifies the resilience of the control system to stochastic errors. Maggio et al. [48] developed a tool for determining the stability of a control system where the control task behaves according to the weakly hard model. From the control perspective, there has been extensive research into both analysis and mitigation of real-time faults in feedback systems [30,31,57]. Very often, this research produced tools to analyse the effect of computational delays [19] and of choosing specific scheduling policies or parameters [18,52], possibly including deadline misses. In a few instances, researchers looked at how to improve the performance of control systems in conjunction with scheduling information [14]. One such effort analyses modifications to the code of classic and simple control systems to handle overruns that reset the period of execution of the control task [53]. Abdi et al. [3] proposed a control design method for safe system-level restart, mitigating unknown faults during runtime execution, while keeping the system inside a safe operating space. Pazzaglia et al. [54] used the scenario theory to derive a control design method accounting for potential deadline misses, and discussed the effect of different deadline handling strategies. Linsenmayer et al. [47] worked on the stabilisation of weakly-hard linear control systems for networked control

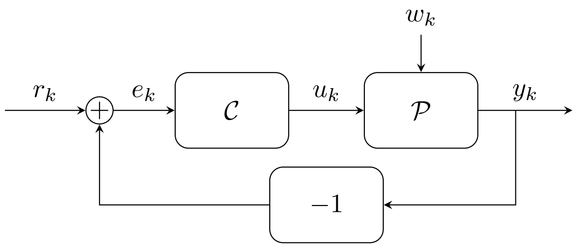


**Figure 1** Control loop: The reference value $r_k$ is compared with the output $y_k$ of the plant $\mathcal{P}$. The control error $e_k = r_k - y_k$ is used by the controller $\mathcal{C}$ to compute the value of the control signal $u_k$. The plant is disturbed by the stochastic process $w_k$.

systems, with some extension for nonlinear systems [39]. In the considered setup, faults compromise network transmissions, but do not interfere with the controller computation (assuming that the computation is triggered). The work also focused on stability, with no control performance evaluation.

To the best of our knowledge, no previous work has devised a combined stability and performance analysis to understand how faults (even when they can be tolerated) affect the plant that should be controlled when different deadline handling strategies are used.

## 3 System Behaviour in Nominal Conditions

In this section, we introduce the relevant control background needed for the remainder of the paper, and we detail how the controller and the system behave under normal operation.

### 3.1 Plant Model

We first describe the model we use for the object we are trying to control. In control terms – mostly due to historical reasons – this object is called a *plant*. Examples range from a pendulum that we would like to stabilise in the upward position, to a chemical dilution process, to the distribution of workload in a datacenter.

Plants are usually modelled as continuous- or discrete-time dynamical systems. All real-world plants are nonlinear, but for control design purposes they are often linearised around their operating points. Around such a point, the resulting model becomes a Linear Time-Invariant (LTI) system. In this paper, we restrict our analysis to discrete-time LTI systems, because we investigate controllers implemented with fixed-rate sampling and actuation in digital electronics. To design and analyse these systems, we use the discrete-time counterpart of the continuous-time physical model, which can be obtained with standard techniques [9].

We consider a plant $\mathcal{P}$ described in state-space form:

$$\mathcal{P} : \begin{cases} x_{k+1} &= A_p\, x_k + B_p\, u_k + G_p\, w_k \\ \quad y_k &= C_p\, x_k + D_p\, u_k \end{cases} \tag{1}$$

In (1), $k$ counts the discrete instants that represent the plant's sampling points. We assume periodic sampling; the time between two consecutive samples $k$ and $k+1$ is fixed and equal to sampling period $t_s$. In the equation, $x_k$ is a column vector with $d_p$ elements. These elements represent the state variables that account for, e.g., the storage of mass, momentum, and energy. Similarly, $u_k$ is a column vector with $i_p$ elements. These values represent the inputs that affect the dynamics of the plant. We also consider $w_k$, a column vector with $i_p$ elements. The term $w_k$ represents an unknown load disturbance, modelled as a stationary stochastic process with known properties. Finally, $y_k$ is a column vector with $o_p$ elements, that represents the measurements that are taken from our plant. The matrices $A_p$ (size $d_p \times d_p$), $B_p$ (size $d_p \times i_p$), $C_p$ (size $o_p \times d_p$), $D_p$ (size $o_p \times i_p$), and $G_p$ (size $d_p \times i_p$) characterise the dynamics of the plant.

## 3.2 Controller Model

The plant $\mathcal{P}$ is controlled by a periodically executing controller $\mathcal{C}$ with implicit deadlines, i.e., the deadline of each task instance (job) coincides with the next task activation. We consider the class of all linear controllers with a one-step delay between sampling and actuation.[1] In other words, we consider all the controllers that can be written as linear systems, according to the following state-space equation:

$$\mathcal{C}: \begin{cases} z_{k+1} &= A_c\, z_k + B_c\, e_k \\ u_{k+1} &= C_c\, z_k + D_c\, e_k \end{cases} \tag{2}$$

Here, $z_k$ is a column vector with $d_c$ elements that represents the state of the controller. The input of the controller is $e_k$, a vector of $i_c = o_p$ elements. Each element in the vector is the error between the corresponding plant output and its reference value ($e_k = r_k - y_k$, where $r_k$ represents the reference values for the plant outputs). Finally, $u_k$ is a vector of $o_c = i_p$ elements, that encodes the output of the controller, which is connected to the plant input vector. The matrices $A_c$ (size $d_c \times d_c$), $B_c$ (size $d_c \times i_c$), $C_c$ (size $o_c \times d_c$), and $D_c$ (size $o_c \times i_c$) characterise the dynamics of the controller. For every task activation, the controller first applies the value of $u_k$ that was computed by the previous job and then reads the inputs $r_k$ and $y_k$. It then calculates the values of $z_{k+1}$ and $u_{k+1}$ that will be used in the next iteration.

The analysis methodology presented in the remainder of this paper is valid for *all* linear controllers. The class of linear controllers includes some of the most frequently used controllers in industry, in particular proportional and integral (PI), proportional, integral, and derivative (PID), lead–lag compensators, and linear-quadratic-Gaussian (LQG) controllers. Although the performance analysis is presented for the time-invariant case, the formulas are valid also for systems with time-varying matrices. Hence, it is possible to analyse plants and controllers that transition between different local linear models.

## 3.3 Closed-Loop System Dynamics

We now analyse the closed-loop system shown in Figure 1. Combining the dynamical models from (1) and (2), we obtain matrices that represent the closed-loop system. We denote the state vector of the closed-loop system with $\tilde{x}_k = \left[x_k^T,\, z_k^T,\, u_k^T\right]^T$, where $^T$ is the transpose operator. In this way, we obtain a system that has the vectors $r_k$ and $w_k$ as input, and is described by

$$\mathcal{S}: \begin{cases} \tilde{x}_{k+1} &= A\, \tilde{x}_k + B_r\, r_k + B_w\, w_k \\ y_k &= C\, \tilde{x}_k, \end{cases} \tag{3}$$

[1] One-step delay controllers are controllers in which a control signal is computed in the $k$-th interval and actuated at the beginning of the $k+1$-th period. In the real-time systems jargon, one-step delay controllers are often referred to as controllers that follow the Logical Execution Time (LET) paradigm [25, 44]. From the real-time perspective, implementing the controller following the LET paradigm improves the timing predictability. From the control perspective, one-step delay controllers reduce activation jitter and allows the engineer to neglect time-varying computational delays.

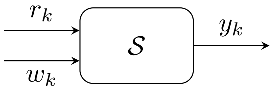


**Figure 2** Closed-loop system rewritten as a new linear system $\mathcal{S}$. The resulting system has two inputs, $r_k$ and $w_k$ and one output. The feedback loop shown in Figure 1 is hidden inside $\mathcal{S}$.

where the closed-loop state matrix $A$ is

$$A = \begin{bmatrix} A_p & 0_{d_p \times d_c} & B_p \\ -B_c\, C_p & A_c & -B_c\, D_p \\ -D_c\, C_p & C_c & -D_c\, D_p \end{bmatrix}, \tag{4}$$

the input matrices $B_r$ and $B_w$ are

$$B_r = \begin{bmatrix} 0_{d_p \times i_c} \\ B_c \\ D_c \end{bmatrix}, \quad B_w = \begin{bmatrix} G_p \\ 0_{d_c \times i_p} \\ 0_{i_p \times i_p} \end{bmatrix}, \tag{5}$$

and the output matrix $C$ is

$$C = \begin{bmatrix} C_p & 0_{d_p \times d_c} & D_p \end{bmatrix}. \tag{6}$$

Figure 2 shows the graphical representation of the closed-loop system $\mathcal{S}$, with input and output signals.

### Stability

To assess the stability of the closed-loop system under normal operation, it is sufficient to check the eigenvalues of the state matrix. According to the Schur stability criterion [9], if the eigenvalues of $A$ lie within the unit disc, then the system is asymptotically stable. Formally, a closed-loop system is Schur stable if and only if

$$\max_i |\lambda_i(A)| < 1, \tag{7}$$

where $\lambda_i(A)$ is a function that returns the $i$-th eigenvalue of $A$.

If the system dynamics change at runtime (e.g., in the case of a lost sample, unexpected delay, or computational problem), Schur stability is no longer a sufficient stability criterion. Instead, *switching stability analysis* can be employed to check the stability of a system with alternating dynamics [40]. There has been a lot of research on the switching stability analysis, with multiple tools developed in order to simplify the analysis. Two main methods are employed: (i) the search for a common Lyapunov function, e.g., as done in [46], (ii) the computation of the Joint Spectral Radius (JSR), e.g., as done in [48, 62].

### Performance

Alongside stability, it is important to look at the *performance* of the closed-loop system. Performance can be defined in different ways, often depending on the application [8]. Whichever way is chosen, a common way to quantify performance is to define a cost function and evaluate the cost function during the execution of the controller. In our work, we use a quadratic cost function

$$J_k = \mathbb{E}\left(e_k^T Q_e e_k + u_k^T Q_u u_k\right). \tag{8}$$

The cost function penalises deviations from the reference value as well as usage of the control signal. $\mathbb{E}$ denotes expected value, and the positive semidefinite weighting matrices $Q_e$ (size $o_p \times o_p$) and $Q_u$ (size $o_c \times o_c$) weigh the different terms against each other. A small cost value means that the controller successfully makes the error approach zero, using a small control signal.

If the stochastic properties of the external signals $r_k$ and $w_k$ are known, it is possible to calculate the value of the cost function analytically. For simplicity and without loss of generality, we will henceforth assume that $r_k = 0$ (i.e., we want to regulate the output to zero) and that $w_k$ is a zero-mean Gaussian white noise process with variance $R = \mathbb{E}\left(w_k w_k^T\right)$. More elaborate disturbance models can be realised by adding extra states in the plant model.

We now detail how to evaluate (8). Let $P_k$ denote the covariance of the closed-loop state vector at time $k$,

$$P_k = \mathbb{E}\left(\tilde{x}_k \tilde{x}_k^T\right). \tag{9}$$

The state covariance evolves according to

$$P_{k+1} = A\, P_k\, A^T + B_w\, R\, B_w^T. \tag{10}$$

Given $P_k$, we can calculate the cost for time step $k$ as

$$J_k = \mathbb{E}\left(\tilde{x}_k^T\, Q\, \tilde{x}_k\right) = \operatorname{tr}\left(P_k\, Q\right), \tag{11}$$

where tr computes the trace of the matrix, and

$$Q = \begin{bmatrix} C_p^T\, Q_e\, C_p & 0_{d_p \times d_c} & 0_{d_p \times i_p} \\ 0_{d_c \times d_p} & 0_{d_c \times d_c} & 0_{d_c \times i_p} \\ 0_{i_p \times d_p} & 0_{i_p \times d_c} & Q_u \end{bmatrix} \tag{12}$$

is the total cost matrix. The stationary cost of the system is defined as $J_\infty$. This is the cost that the system converges to when operating under normal conditions:

$$J_\infty = \lim_{k \to \infty} J_k. \tag{13}$$

This means that there exists an instant $\bar{k}$ for which $J_k$ reaches a value arbitrarily close to the steady-state value $J_\infty$, or $\forall \varepsilon,\ \exists \bar{k}$ s.t. $\forall k > \bar{k},\ |(J_k - J_\infty)/J_\infty| < \varepsilon$.

## 4 System Behaviour with Deadline Misses

The analysis above holds when the control task meets all its deadlines. However, the presence of deadline misses changes the behaviour of the system. The stability of controllers with a number of consecutive deadline misses has been investigated in [48]. The results of this investigation attested that, due to their inherent robustness, many control systems can withstand at least a small number of consecutive misses.

To analyse the system, we need to clarify three aspects about the miss behaviour:

**(i)** What happens to the control signal.
**(ii)** What happens to the control task.
**(iii)** The computational model used for the analysis (how many deadlines can we miss, and in what pattern).

For the first item, the actuator can either output a *zero* ($u_k = 0_{o_c \times 1}$), or *hold* the previous value ($u_k = u_{k-1}$). The choice depends on both the plant dynamics and on the controller, as no strategy in general dominates the other one [58]. For controllers with integral action, it makes sense to hold the previous control value, under the presumption that the system is still disturbed and that a non-zero control signal is needed to keep the plant close to its operating point. On the other hand, the zero strategy may be preferred for plants with unstable or integrator dynamics, where outputting a zero control action may be the safer option.

Considering the second item, at least three different strategies can be employed to deal with a control task that misses its deadline [18]:

**(i)** *Kill*,
**(ii)** *Skip-Next*,
**(iii)** and *Queue*($\lambda$) ($\lambda \in \{1, 2, 3, \ldots\}$).

When the Kill strategy is used, the job that missed its deadline is terminated, its changes are rolled back, and the next job is released. Following the Skip-Next strategy, the job that missed its deadline continues its execution. No new control task jobs are released until the currently running one completes its execution. Queue($\lambda$) behaves similarly to Skip-Next in allowing the current job to complete execution, but also allows the activation of new jobs (the queue of active jobs holds up to the most recent $\lambda$ instances of the control task). In this paper we only analyse Kill and Skip-Next. In fact, the results presented in [18, 48] suggest that Queue($\lambda$) is not a feasible strategy to handle misses. The presence of two or more active jobs in the same period creates a chain effect that is hard to recover from and that deteriorates stability and performance.

The last item refers to models of computation. The weakly hard task model [11, 34] is usually considered expressive enough to analyse the behaviour of tasks that miss their deadlines. The authors of [11] propose four definitions for a weakly hard real-time task $\tau$:

▶ **Definition 1** (Weakly Hard Task Models [11]). *A task $\tau$ may satisfy any of these four weakly hard constraints:*

**(i)** $\tau \vdash \binom{n}{\ell}$*: there are at least $n$ hits for every $\ell$ jobs,*
**(ii)** $\tau \vdash \binom{\overline{m}}{\ell}$*: there are at most $m$ misses for every $\ell$ jobs,*
**(iii)** $\tau \vdash \left\langle{n \atop \ell}\right\rangle$*: there are at least $n$ consecutive hits for every $\ell$ jobs,*
**(iv)** $\tau \vdash \left\langle{\overline{m} \atop \ell}\right\rangle$*: there are at most $m$ consecutive misses for every $\ell$ jobs.*

There has been a lot of research on the second model, often also called $m$-$K$ model [4, 12, 13, 26, 29, 35, 38, 45, 54, 55, 57, 59–61] (with $m$ being the maximum number of misses in a window of $K$ activations). Recently there has also been an analysis of the stability of control systems when the control task behaves according to the fourth model [48].

If the misses are due to faults or security attacks, usually the control task experiences an interval of consecutive misses. When the fault is resolved, the control task starts hitting its deadlines again. From the performance standpoint, a consecutive number of misses degrades the control quality. We are interested in what degradation is acceptable and how much time should occur between two potential failures. Specifically, we look at how many deadline hits should follow a given number of consecutive misses for the system to *recover*. None of the four models above allow us to formulate this requirement (as they specify either consecutive hits or misses but not both), which leads us to introduce a different weakly hard model of computation, together with its analysis, in Section 5.

# 5 Burst Interval Analysis

In this section, we analyse the stability and performance of a real-time control system that experiences bursts of deadline misses. Section 5.1 introduces the fault model, Section 5.2 derives the control system behaviour subject to different real-time policies and delves into both the stability and performance analysis.

## 5.1 Fault Model

Faults can happen during the normal execution of tasks on a platform. Informally, as a result of a fault, tasks miss their deadlines. When the fault is resolved, then the original situation is recovered (possibly after a transient initial phase).

Specifically, given a system $\mathcal{S}$, we define a *burst interval* $\mathcal{M}$ as an interval of controller activations in which the control task executing $\mathcal{C}$ consecutively misses $m$ deadlines, regardless of the strategy used to handle the misses. We assume that the burst interval $\mathcal{M}$ is followed by a *recovery interval* $\mathcal{R}$, defined as an interval in which the control task consecutively hits $n$ deadlines.

During the burst interval, the deadline misses of the control task are handled using a *deadline handling strategy* $\mathcal{D}$ (Kill, $K$, or Skip-Next, $S$). The control signal $u_k$ is selected in accordance with the *actuation strategy* $\mathcal{A}$ (Zero, $Z$, or Hold, $H$). We denote the combination of $\mathcal{D}$ and $\mathcal{A}$ with $\mathcal{H} = (\mathcal{D}, \mathcal{A})$. For example $\mathcal{H}$ could be $SZ$ to indicate that the Skip-Next deadline handling strategy is paired with the Zero actuation strategy. The system *recovers* once it operates close to steady-state.

From an industrial viewpoint, the proposed fault model is highly relevant. The common approach is to treat faults as pseudo-independent events adhering to predefined constraints on their incidence rate [42, 49, 51]. However, during the operation of a control system, faults can be caused by events like network connection problems (e.g., cutting the connection between the sensor and the controller), security attacks, contention on resources. Studies in the automotive sector, for example, indicate that deadline misses can occur in bursts [56, 64]. In these cases, the controller does not execute properly for a given amount of time (e.g., until the connection is restored, the attack is terminated, or the resource contention is reduced). The analysis methods we propose allow us to address such situations and to provide tighter bounds on the closed-loop stability and performance than under the previously proposed weakly hard models. Moreover, following a burst interval, we are interested in analysing the length of the recovery interval $\mathcal{R}$ that is needed to return to normal operation under each implementation strategy $\mathcal{H}$. Hence, we here extend the weakly hard models of computation with a fifth alternative and then devote the remainder of the paper to its analysis.

▶ **Definition 2** (Weakly Hard Fault Model With Burst Of Misses). *A real-time task $\tau$ may satisfy the weakly hard task model*

**(v)** $\tau \vdash \left\{ {m \atop \ell} \right\}$*: there are at most $m$ consecutive misses, followed by $\ell - m$ consecutive hits for every $\ell$ jobs.*

*This means that a real-time task $\tau$ behaves according to the model $\tau \vdash \left\{ {m \atop \ell} \right\}$, if, whenever $\tau$ experiences a burst interval $\mathcal{M}$ consisting of $m$ consecutive deadline misses, it is always followed by a recovery interval $\mathcal{R}$ consisting of $n = \ell - m$ consecutive deadline hits.*

## 5.2 Closed-Loop System Dynamics

In this section we derive the system dynamics for a closed-loop control system under the assumption that we enter a burst interval of length $m$ after time instant $k$, and after $m$ deadline misses we start completing the control job in time.

**Normal Operation.** Under *normal operating conditions* the system is not experiencing any deadline misses. In other words, the system evolves according to the closed-loop system dynamics (3).

**Kill&Zero.** If a control task deadline miss occurs at time instant $k$, the plant states $x_k$ still evolve as normal. However, the controller terminates its execution prematurely by killing the job, thus not updating its states ($z_{k+1} = z_k$). The controller output is determined by the actuation strategy and is here zero ($u_{k+1} = 0$). Now, consider a burst interval of length $m$ after time instant $k$. Recalling that $\tilde{x}_k = \left[{x_k}^T {z_k}^T {u_k}^T\right]^T$, we can write the evolution of the closed-loop system for the sequence of $m$ deadline misses followed by a single deadline hit as the product of a matrix representing the behaviour of the system for a hit and a matrix representing the behaviour in case of miss elevated to the power of $m$ to indicate $m$ steps of the system evolution.

The resulting closed-loop system in state-space form is

$$\begin{bmatrix} x_{k+m+1} \\ z_{k+m+1} \\ u_{k+m+1} \end{bmatrix} = A \underbrace{\begin{bmatrix} A_p & 0_{d_p \times d_c} & B_p \\ 0_{d_c \times d_p} & I & 0_{d_c \times i_p} \\ 0_{i_p \times d_p} & 0_{i_p \times d_c} & 0_{i_p \times i_p} \end{bmatrix}^m}_{A_{KZ}(m)} \begin{bmatrix} x_k \\ z_k \\ u_k \end{bmatrix}, \tag{14}$$

where $A_{KZ}(m)$ represents the system matrix for $m$ misses under the Kill&Zero strategy, followed by a single hit (the matrix $A$ that is multiplied to the left of the equation). The matrix $A$ is the same specified in (4), and represents the first hit that follows the $m$ misses, hence, we determine how $\tilde{x}_k$ influences $\tilde{x}_{k+m+1}$ ($m$ misses and one hit).

**Kill&Hold.** Changing the actuation strategy to Hold, slightly alters the system matrix we derived for the Kill&Zero case. The plant states $x_k$ evolve as normal and the control states $z_k$ are still not updated ($z_{k+1} = z_k$). However, due to the change in actuation strategy, the last actuated value is instead held ($u_{k+1} = u_k$). The resulting closed-loop state-space form can be seen in (15), where $A_{KH}(m)$ is used to represent the system matrix for $m$ misses under the Kill&Hold strategy and matrix $A$ is specified in (4).

$$\begin{bmatrix} x_{k+m+1} \\ z_{k+m+1} \\ u_{k+m+1} \end{bmatrix} = A \underbrace{\begin{bmatrix} A_p & 0_{d_p \times d_c} & B_p \\ 0_{d_c \times d_p} & I & 0_{d_c \times i_p} \\ 0_{i_p \times d_p} & 0_{i_p \times d_c} & I \end{bmatrix}^m}_{A_{KH}(m)} \begin{bmatrix} x_k \\ z_k \\ u_k \end{bmatrix} \tag{15}$$

**Skip-Next&Zero.** When the control task misses a deadline under the Skip-Next strategy, the job missing the deadline is allowed to continue its execution until completion. However, no subsequent job of the control task is released until the current job has finished executing. If the currently active job terminates during period $k$, the next control job is released at the start of the $k+1$-th period. We can then write the evolution of the system where the control job experiences $m$ misses before completing its execution, meaning that there is a subsequent hit that uses old information for the error measurements. While the controller executed only once to completion, the plant evolved for $m+1$ steps. The resulting closed-loop state-space form can be seen in (16), where $A_{SZ}(m)$ is used to represent the system matrix under the Skip-Next&Zero strategy for $m$ misses and one completion using old measurements.

$$\begin{bmatrix} x_{k+m+1} \\ z_{k+m+1} \\ u_{k+m+1} \end{bmatrix} = \underbrace{\begin{bmatrix} A_p^{m+1} & 0_{d_p \times d_c} & A_p^m B_p \\ -B_c C_p & A_c & -B_c D_p \\ -D_c C_p & C_c & -D_c D_p \end{bmatrix}}_{A_{SZ}(m)} \begin{bmatrix} x_k \\ z_k \\ u_k \end{bmatrix} \tag{16}$$

**Skip-Next&Hold.** Similar to Skip-Next&Zero, one job finishes execution after $m$ consecutive misses. However, the actuation strategy holds the previous control value during the entire burst interval. Therefore, the plant evolution is affected by a cumulative sum over the prior control values. The resulting closed-loop state-space form can be seen in (17), where $A_{SH}(m)$ is used to represent the system matrix for $m$ misses under the Skip-Next&Hold strategy.

$$\begin{bmatrix} x_{k+m+1} \\ z_{k+m+1} \\ u_{k+m+1} \end{bmatrix} = \underbrace{\begin{bmatrix} A_p^{m+1} & 0_{d_p \times d_c} & \sum_{i=0}^{m} A_p^i B_p \\ -B_c C_p & A_c & -B_c D_p \\ -D_c C_p & C_c & -D_c D_p \end{bmatrix}}_{A_{SH}(m)} \begin{bmatrix} x_k \\ z_k \\ u_k \end{bmatrix} \tag{17}$$

Equations (14)–(17) are inspired by the analysis in [48], but we have we introduced two generalisations. The first one is that our controller is specified as a general state-space system; therefore our method is able to address *all* linear controllers. The second generalisation is that we could include estimates of the plant states in the controller. We can thus properly handle the presence of an observer.[2] Furthermore, we simplify the calculations by reducing the number of states $\tilde{x}_k$ of the closed-loop matrices.

### Stability

We now describe how the system matrices above can be used to analyse stability. Recall that a closed-loop control system is stable if and only if the (fixed) system matrix $A$ is Schur stable. This criterion is also valid for cyclic patterns, where $A$ represents the product of all closed-loop state matrices experienced in a full burst–recovery cycle. Hence, we can search for the shortest recovery interval length $n$ such that

$$\max_i \left| \lambda_i \left( A^{n-1} A_{\mathcal{H}}(m) \right) \right| < 1, \quad \mathcal{H} \in \{KZ, KH, SZ, SH\}. \tag{18}$$

Recall that $A_{\mathcal{H}}(m)$ already includes one hit, thus the left multiplication with $A^{n-1}$. This is a sufficient condition and not necessary, meaning that a miss occurring during the recovery interval does not immediately imply that the closed-loop system is destabilised. We summarise the analysis in the following definition.

▶ **Definition 3** (Static-Cyclic Stability Analysis)**.** *We denote the stability analysis from* (18) *with the term* static-cyclic stability analysis*. The system under analysis cycles through a sequence of $m$ misses followed by a sequence of $n$ hits, indefinitely.*

The static-cyclic analysis assumes a repeating burst–recovery cycle with no interruptions. This works well for instance in case the misses are due to a permanent overload condition caused by a mode switch (for example from low to high criticality mode in mixed-critical systems). However, the setting is not very general. To foster generality, we complement the stability evaluation with a less restrictive stability analysis, based on the proposed task model in Definition 2.

▶ **Definition 4** (Miss-Constrained Stability Analysis)**.** *To guarantee* miss-constrained stability*, a system has to be stable under arbitrary switching between all the possible $m$ realisations (i.e., closed-loop matrices) that comply with all task models $\tau \vdash \left\{ {m_\subset \atop \ell} \right\}, m_\subset \in \{1, \ldots, m\}$ and also include the case in which the system does not miss deadlines.*

[2] In [48] the controller state is specified as part of the plant (e.g., when the proportional and integral controller is introduced). This implies that the state is computed although the controller did not execute. Our formulation fixes this by separating the plant execution and the controller states.

In other words, a system is miss-constrained stable if and only if it is stable under arbitrary switching of the closed-loop matrices in the set

$$\left\{A^{\ell-1}A_{\mathcal{H}}\left(1\right),\, A^{\ell-2}A_{\mathcal{H}}\left(2\right),\, \ldots,\, A^{\ell-m}A_{\mathcal{H}}\left(m\right),\, A\right\}. \tag{19}$$

Switching stability is unfortunately quite involved.[3] However, many excellent tools have been developed to simplify this analysis (e.g., `MJSR` [48] or the `JSR toolbox` [62] for MATLAB).

## Performance

We now show how the cost function in Equation (11) can be used as a time-varying performance metric. Before a burst interval, we assume that the system is in the neighbourhood of its steady-state covariance $P_\infty$ and performance $J_\infty$.

When a burst interval of $m$ missed deadlines occurs, the system will be disrupted and its covariance matrix will evolve according to

$$P_{k+m+1} = A_{\mathcal{H}}\left(m\right)P_k\left(A_{\mathcal{H}}\left(m\right)\right)^T + A^{j_n}R_w\left(A^{j_n}\right)^T, \tag{20}$$

where

$$\begin{aligned} R_w &= \begin{bmatrix} \sum_{i=0}^{j_m} A_p^i\, G_p\, R\, G_p^T\, (A_p^i)^T & 0_{d_p\times d_c+i_p} \\ 0_{d_c+i_p\times d_p} & 0_{d_c+i_p\times d_c+i_p} \end{bmatrix}, \\ j_m &= \begin{cases} m-1 & \text{if } \mathcal{D}=K \text{ (Kill)}, \\ m & \text{if } \mathcal{D}=S \text{ (Skip-Next)}, \end{cases} \\ j_n &= \begin{cases} 1 & \text{if } \mathcal{D}=K \text{ (Kill)}, \\ 0 & \text{if } \mathcal{D}=S \text{ (Skip-Next)}. \end{cases} \end{aligned} \tag{21}$$

$A_p$ and $G_p$ are matrices from the plant evolution in (1), $R$ is the noise intensity from (10), and $A$ is the closed-loop matrix from (4). The cost will simultaneously change following (11). In the recovery interval, the covariance is again governed by the normal closed-loop evolution described in (10). The system is said to have recovered once the cost is arbitrarily close to the steady-state cost. We evaluate this as

$$\left|\frac{J_\infty - J_k}{J_\infty}\right| < \varepsilon, \tag{22}$$

where $\varepsilon > 0$ is the *recovery threshold.*

▶ **Definition 5** (Performance Recovery Interval). *We define the recovery length interval $n^*_{\mathcal{H}}$ as the smallest $n$ such that* (22) *is satisfied for all $k \geq n$ when using $\mathcal{H}$ to handle deadline misses.*

▶ **Definition 6** (Maximum Normalised Cost). *We denote the maximum normalised cost of the system by*

$$J_{M,\mathcal{H}} = \max_k \frac{J_{k,\mathcal{H}}}{J_\infty}, \tag{23}$$

*where $J_{k,\mathcal{H}}$ is the cost computed according to* (11) *when using $\mathcal{H}$ to handle the deadline misses.*

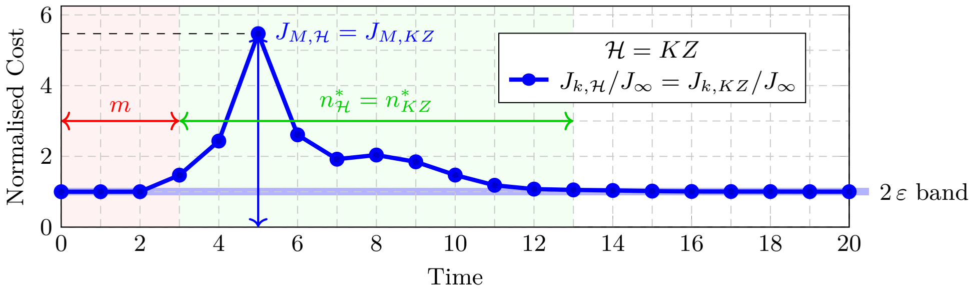


**Figure 3** Illustration of normalised cost ($J_k/J_\infty$), performance recovery interval $n^*_{\mathcal{H}}$ and maximum normalised cost $J_{M,\mathcal{H}}$ on a data trace. The example uses $\mathcal{H}$ = Kill&Zero and $\varepsilon = 0.1$.

Figure 3 gives a graphical representation of $n^*_{\mathcal{H}}$ and $J_{M,\mathcal{H}}$ for an execution trace in which the controller experiences 3 misses and uses Kill&Zero as strategy $\mathcal{H}$.

Compared to the stability analysis, the performance analysis also takes into account state deviations and uncertainty due to disturbances. In Section 5.2 we used the system dynamics to analyse the stability of the system. The disturbance term $w_k$ was neglected as it does not influence the system stability. However, its presence (as the presence of any disturbance) changes the dynamic behaviour of the system. For the performance metric, the state covariance matrix $P_k$ evolves according to both the noise intensity and the system dynamics (20). The result is that the performance analysis provides us with a conservative (but more realistic) recovery interval, that takes system uncertainties into consideration.

To find the length of the recovery interval, we evolve the state covariance during a burst interval, using a specific strategy $\mathcal{H}$ according to (20). Thereafter, the state covariance is evolved under normal operation, according to (10), until (22) is satisfied, allowing us to find the performance recovery interval $n^*_{\mathcal{H}}$.

## 6 Experimental Results

In this section, we apply the analysis presented in Section 5 to a set of case studies, analysing stability and performance. We first present detailed results with a Furuta pendulum, both in simulation and with real hardware, using the same controller. The simulated results are compared to the real physical plant. This shows that the performance analysis does capture the important trends for real control systems. We then present some aggregate results obtained with a set of 133 different plants from a control benchmark. One noteworthy aspect is that the Furuta pendulum model is linearised for the control design and the pendulum stabilised around an unstable equilibrium – the top position – while the control benchmark includes (by design) stable systems. The difference between simulation results and real experiments for stable linear systems should in principle be smaller than for unstable nonlinear systems, making our pendulum the ideal stress test for the similarity of simulated and real data.

[3] We have devoted some research effort into the investigation of a suitable stability analysis for control tasks subject to a set of weakly-hard constraints (of the type presented in Defintion 1). A summary of our findings can be found at https://arxiv.org/abs/2101.11312.

## 6.1 Furuta Pendulum

We here analyse the behaviour of a Furuta pendulum [27], a rotational inverted pendulum in which a rotating arm is connected to a pendulum. The rotation of the arm induces a swing movement on the pendulum. The pendulum has two equilibria: a stable position in which the pendulum is downright, and an unstable position in which the pendulum is upright. Our objective is to keep the pendulum in the up position, by moving the rotating arm.

The Furuta pendulum is a highly nonlinear process. In order to design a control strategy to keep the pendulum in the top position, it is necessary to linearise the dynamics of the system around the desired equilibrium point. We consider this as a stress test to check the divergence between simulation results and real hardware results, because of the instability of the equilibrium and the nonlinearity of the dynamics. In fact, the controller necessarily acts with information that is valid only around the upright position, and there is only a range of states in which the linearised model closely describes the behaviour of the physical plant.

We design a linear-quadratic regulator (LQR) to control the plant. Every $t_s = 10\,\text{ms}$ the plant is sampled and the control signal is actuated. Based on state-of-the-art models [17] and on our control design, the plant model $\mathcal{P}$ is

$$\mathcal{P} \ : \ \begin{cases} x_{k+1} = \begin{bmatrix} 1.002 & 0.0100 & 0 & 0 \\ 0.3133 & 1.002 & 0 & 0 \\ -2.943 \cdot 10^{-5} & -9.808 \cdot 10^{-8} & 1 & 0.01 \\ -0.0059 & -2.943 \cdot 10^{-5} & 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} -0.0036 \\ -0.7127 \\ 0.0096 \\ 1.9120 \end{bmatrix} u_k + I w_k, \\ y_k = I x_k, \end{cases} \tag{24}$$

the controller $\mathcal{C}$ takes the form

$$\mathcal{C} \ : \ u_{k+1} = \begin{bmatrix} 8.8349 & 1.5804 & 0.2205 & 0.3049 \end{bmatrix} x_k \tag{25}$$

and is designed and analysed using the following parameters (see Section 3.3):

$$Q_e = \operatorname{diag}(100, 1, 10, 10), \ \ Q_u = 100, \ \ R = \operatorname{diag}(0, 0, 10, 1). \tag{26}$$

We first apply the stability analyses presented in Section 5.2 to our model. Figure 4 shows the results. Each square in the figure represents a combination of (at most) $m$ deadline misses (on the vertical axis) and (at least) $n$ deadline hits (on the horizontal axis). If a square is coloured with a dark colour, the corresponding combination of misses and hits is both static-cyclic and miss-constrained stable, found using the `JSR Toolbox` [62]. The light squares in the figure show combinations for which the system only satisfies the static-cyclic stability condition. The white squares mark configurations for which stability cannot be guaranteed.

We remark on the presence of peaks in the static-cyclic stability region of $\mathcal{H} = KH$ at $n = \{1, 5, 9, 13, 19\}$. Similar peaks are also found for the other strategies, but for different values of $n$. These peaks indicate that the system would be stable if that particular burst and recovery interval length would be repeated indefinitely. However, this assumption is not robust to variations in the burst or recovery interval lengths as can be seen from the miss-constrained stability region being more conservative with its guarantees. Instead, the peaks in the static-cyclic region can be explained by stable modes occurring due to the natural frequencies of the open-loop (for the Zero actuation mode) and closed-loop (for the Hold actuation mode) systems. It is also interesting to note that Kill seems to consistently yield a larger stability region than Skip-Next, while neither Zero nor Hold dominate each other in terms of stability guarantees. An example of the latter fact was given already in [58].

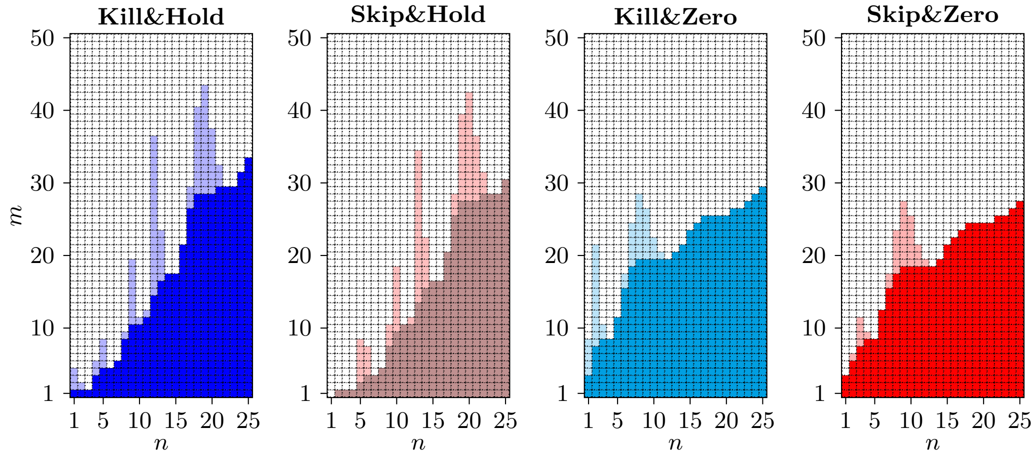


**Figure 4** Miss-constrained stability (dark coloured area) and static-cyclic stability (light coloured area) when different strategies $\mathcal{H}$ are used in the example and the weakly hard model in Definition 2 is considered. Each square represents a window of size $\ell = m + n$. The dark area satisfies both the miss-constrained and static-cyclic stability whilst the light area only provides static-cyclic stability. The white squares denote potentially unstable combinations of $m$ and $n$.

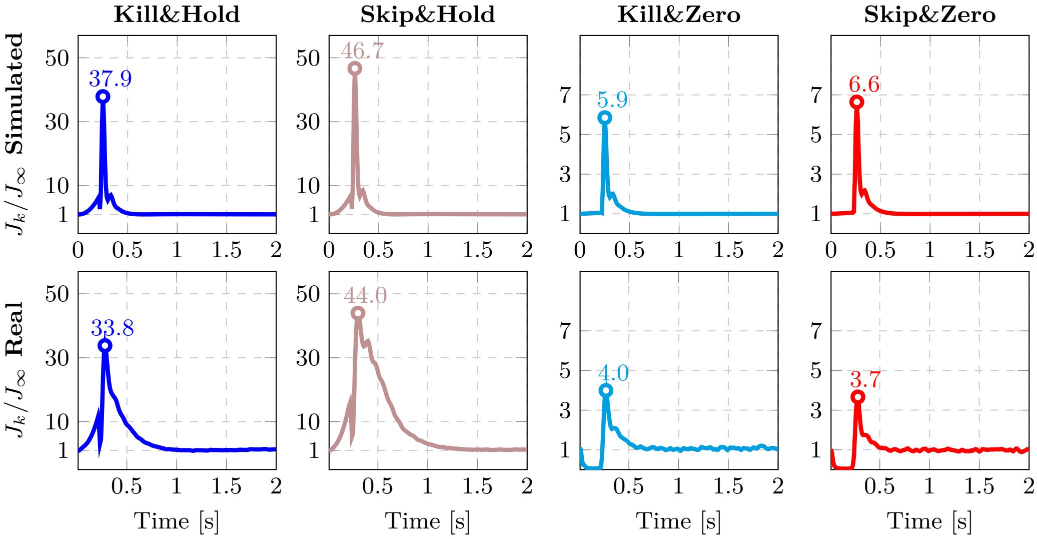
For the performance analysis, we considered a one-shot burst fault of a specific length $m$, followed by a long period of normal execution. Assuming that the pendulum starts close to the upright equilibrium, with stationary cost $J_\infty$, we calculate how the covariance $P_k$ and performance cost $J_k$ evolve during and after the burst interval using Equations (20)–(21).[4] These calculations assume an ideal, linear model of the pendulum. The simulation results for different strategies and bursts of length $m = 20$ are shown in the upper half of Figure 5. For Hold, it is seen that the cost grows exponentially during the initial fault interval (the first $20\,t_s = 0.2\,\mathrm{s}$). This is true also for Zero, although the growth rate is too small to be visible. The reason for the poor performance of Hold is that any non-zero held control signal will actively push the pendulum away from its unstable upright equilibrium even further than either disturbances or noise would already do without a proper control action.

The large spike in cost comes when the controller is reactivated at time 0.2 s. Here, the Hold strategy again shows much worse performance than Zero, with the peak cost being almost an order of magnitude worse. The difference between Kill and Skip-Next is relatively small, with the latter strategy consistently performing slightly worse than the former. This is due to the small extra delay caused by using old data in the Skip-Next strategy.

We conducted experiments on a Furuta pendulum, using the same controller for the real plant rather than its model.[5] Initially, we performed 500 experiments with 500 jobs each and no deadline misses, to determine the nominal variance of the system – i.e., the stationary variance used to find the static cost $J_\infty$. For each strategy $\mathcal{H}$ we then ran 500 identically set up experiments. In each experiment, the control task operated according to the task model

[4] The analysis is implemented using `JitterTime` [19], `https://www.control.lth.se/jittertime`.

[5] A video, showing experiments with the real system and bursts of deadline misses can be viewed at `https://youtu.be/0P0K_7lvKVU`. The video shows a comparison of all the strategies for bursts of $(m = 20, n = 480)$. Furthermore, we have included additional experiments with $(m = 50, n = 450)$ and $(m = 75, n = 425)$ for the Skip&Hold strategy. The results of the additional experiments with higher values of $m$ are not described in the paper, as stability could not be guaranteed (and in fact the pendulum is not at all times kept in the upright position).



■ **Figure 5** Normalised performance cost $J_k/J_\infty$ obtained with the Furuta pendulum. The upper part of the figure shows simulated data, while the lower part of the figure shows the corresponding values obtained averaging the results of 500 experiments with the real process and hardware. Each experiment corresponds to a 500 jobs of the controller (20 misses and 480 hits).

from Definition 2, experiencing a burst of length $m = 20$ misses, followed by by a recovery interval with $n = 480$ deadline hits.

Due to system model uncertainties (e.g., friction) being significant, the rotation angle around the arm axis displayed a considerable variance. We removed the state from the covariance calculations, since the arm angle majorly impacted the variance despite its inconsequential significance on the system dynamics (the pendulum can be stabilised with the arm being around any position, provided that the pendulum itself is kept in the upright position). Including the rotation angle would not change the shape of the performance degradation seen in Figure 5. However, it would make the results obtained with different strategies $\mathcal{H}$ not comparable (in some of them, the rotation angle could have varied less across the 500 experiments). The covariance matrix $P_k$ was derived by calculating the variance of the closed-loop state vector $\tilde{x}_k$ according to Equation (9), in each time step $k$.

The resulting performance cost can be seen in the lower half of Figure 5, where the cost $J_k$ was calculated according to Equation (11) and normalised using the stationary cost $J_\infty$. Comparing the simulated (upper) and real (lower) performance costs in Figure 5, we notice the similarities between the simulated analysis and the analysis performed on the physical plant. Particularly, the strategies involving Hold actuation show similar behaviours. For these strategies, the simulated and real values are very close for the transient burst interval, the secondary cost peak (seen around time 0.4 s), and the maximum normalised cost $J_{M,\mathcal{H}}$. However, the real cost is recovering slower than in the simulations – an effect that arises due to the nonlinear effects present in the real process, but unmodelled in the simulated environment. Instead, comparing the Zero actuation strategies, the performance cost of the physical experiments during the burst interval seem to improve compared to the simulations. This is again likely due to the unmodelled dynamics (e.g., friction) appearing in the physical experiment but not in the simulations. The stiction component of the friction reduces the variance of the states when the actuation signal becomes zero. With longer burst intervals, a similar behaviour as for the Hold actuation strategies would appear. Despite this difference, both the recovery interval, the secondary cost peak (around 0.4 s), and the maximum normalised costs $J_{M,\mathcal{H}}$ are comparable.

We conclude that the results of the experiments performed on the physical process support the validity of the performance analysis presented in Section 5.2.

## 6.2 Control Benchmark

In Section 6.1 we extensively discussed the results obtained with a single plant (the Furuta pendulum), with the aim of showing that simulating the performance cost yields interesting and relevant results. As the main novelty of this paper lays in the introduction of the performance analysis as an additional tool to evaluate the behaviour of control systems that can miss deadlines, we here focus on performance.

We use a set of representative process industrial plants [7], developed to benchmark PID design algorithms in the control literature. The set includes 9 different batches of stable plants, each presenting different features that can be encountered in process industrial plants, for a total of 133 plants.[6] For each batch, all systems have the same structure, but different parameters. For example, the fourth batch is a stable system with a set of repeated eigenvalues, and a single parameter specifying the system order, which can take six possible values (3, 4, 5, 6, 7, or 8). Almost all the plants have a single independent parameter. The only exception is Batch 7, for which we can specify two different configuration parameters, the first one having 4 possible values and the second one having 9 potential alternatives, with a total of 36 possible configurations.

The analysis methodology presented in this paper is valid for *all* linear control systems. In Section 6.1, we introduced an LQR controller to analyse the Furuta pendulum. To demonstrate the generality of the analysis, here, we focus on the most common controller class: proportional and integral (PI) controllers. These controllers constitute the vast majority of all the control loops in the process industry.[7] We also performed the analysis for proportional, integral, and derivative (PID) controllers obtaining similar results. Introducing our tuning for PID controllers requires additional clarifications and details, which we omit due to space limitations.

For each plant we derived a PI controller according to the methodology presented in [28]. In order to showcase the applicability of our analysis to different linear systems, controllers, and noise models, we analyse the resulting closed-loop systems for $m \in [1, 20]$, under the assumption that the systems are affected by brown noise (in comparison to the white noise applied to the Furuta Pendulum). The brown noise model integrates the white noise and is thus applicable to systems where the noise is more dominant at lower frequencies (e.g., oscillations from nearby machinery). Figure 6 shows the results for $m = 10$.

The first result that the figure shows is that the plant dynamics plays an important role in how the system reacts to misses. For example, the plants in Batch 4 and Batch 8 need around 20 hits to recover from a burst of 10 misses. On the contrary, the plants in Batch 6 and Batch 7 need a higher number of hits to recover from the same burst interval. The second result that is apparent from the figure is that the Hold actuation strategy recovers much better (performance-wise) than Zero. The reason why Hold outperforms Zero can be explained by the brown noise. The control signal will actively counteract the integrated noise dynamics,

[6] In our analysis, we present results with 134 plants. In fact, the test set was used in [28] to assess a control design method, and an additional plant was added to the set during this assessment. We included this additional plant in our analysis.

[7] A 2001 survey by Honeywell [24] states that 97% of the existing industrial controllers are PI controllers.

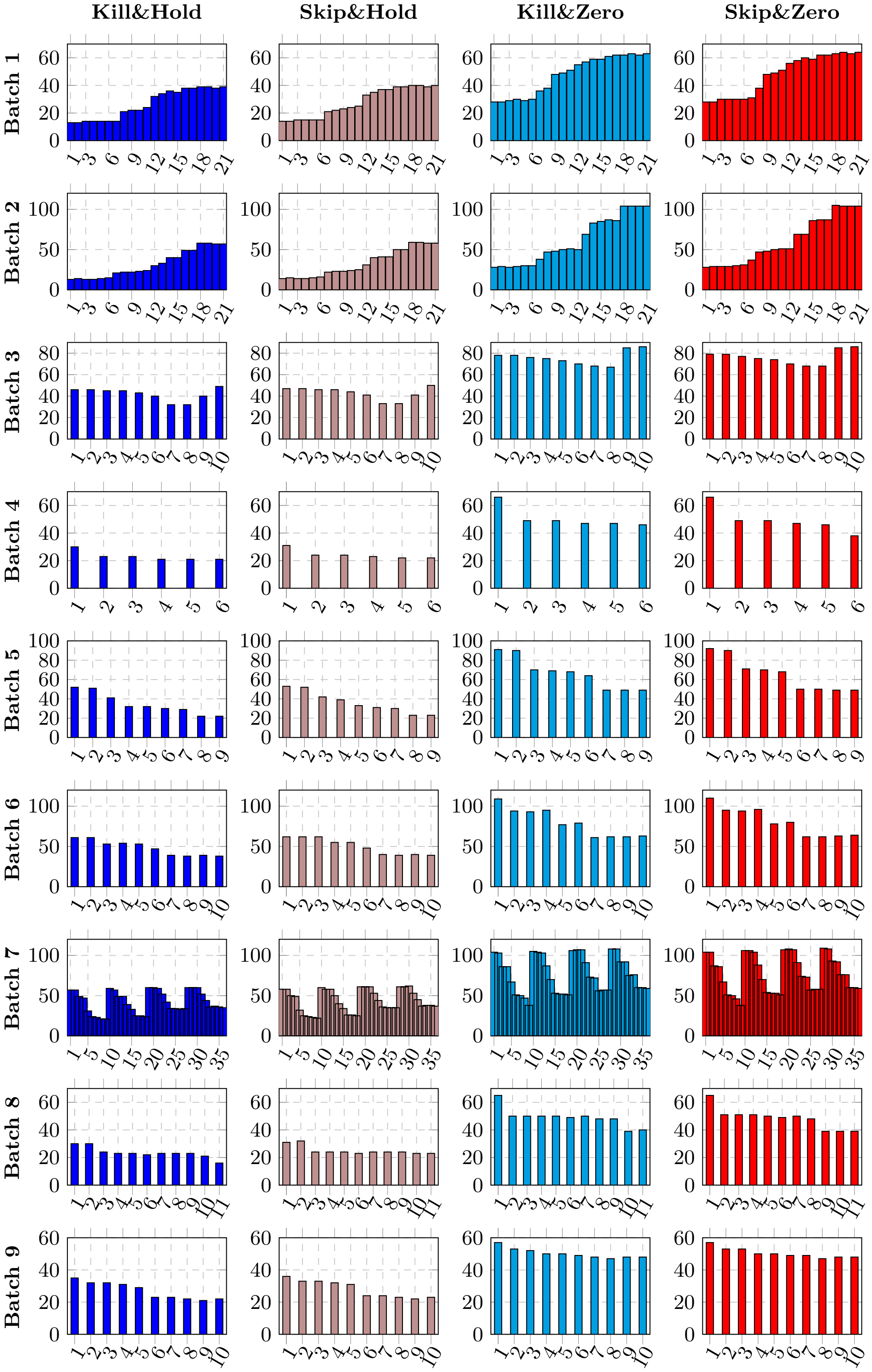


**Figure 6** Performance Recovery Interval $n^*_{\mathcal{H}}$ needed to recover from a burst of 10 deadline misses for different strategies and all the plants in the 9 batches for PI controllers designed according to [28].

meaning that zeroing the control signal removes the compensation against the integrated noise. Finally, comparing the deadline handling strategies, Kill performs marginally better than Skip-Next. Under Kill, the controller uses fresh data at the beginning of the recovery interval, while Skip-Next uses old data. However, we assumed ideal rollback (i.e., zero additional computation time for the rollback and clean state) for the Kill strategy. In real systems, rollback is difficult to realise and the advantage provided by Kill over Skip-Next may therefore become unimportant. These findings are consistent throughout all the plants in the experimental set, regardless of the burst interval length $m$.

The plant dynamics and noise affect the behaviour and performance of the strategies. Comparing the results of Section 6.1 with the aggregate results, it becomes apparent that the actuation strategy (Zero or Hold) affects control performance significantly more than the deadline handling strategy. For the Furuta pendulum (an unstable, nonlinear plant influenced by white noise) Zero performed the best, but for the process industrial systems (stable, linear plants influenced by brown noise) Hold outperformed Zero. These results were apparent even with no consideration taken to the deadline handling strategies. Thus, we conclude that the plant and noise model should be the ruling factor when choosing the actuation strategy, while the deadline handling strategy is mainly limited by the constraints imposed by the real-time implementation.

## 7 Conclusions

In this paper we analysed control systems and their behaviour in the presence of bursts of deadline misses. We provided a comprehensive set of tools to determine how robust a given control system is to faults that hinder the computation to complete in time, with different handling strategies. Our analysis tackles both stability and performance. In fact, we have shown that analysing the stability of the system is not enough to properly quantify the robustness to deadline misses, as the performance loss could be significant even for stable systems. We introduced two performance metrics, linked to the recovery of a system from a burst of deadline misses.

A limitation of the presented performance analysis is that it only applies to linear control systems. However, the approach can easily be extended to analyse *time-varying* linear systems and can also be used for local analysis of a nonlinear system that should follow a given reference trajectory. In fact, to illustrate the applicability to real (e.g., nonlinear) systems, we applied the analysis to a Furuta pendulum and compared the results of simulations obtained with a model of the process to the real execution data. The results support our claim that the proposed performance analysis is a valid approximation of the real-world system performance.

We performed additional tests on a large batch of industrial plants, using modern control design techniques. From our experimental campaign, we conclude that the choice of actuation strategy affects the control performance significantly more than the choice of deadline handling strategy.

## References

1 F. Abdi, C. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo. Preserving physical safety under cyber attacks. *IEEE Internet of Things Journal*, 6(4), 2019.

2 F. Abdi, R. Mancuso, R. Tabish, and M. Caccamo. Restart-based fault-tolerance: System design and schedulability analysis. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

3 F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo. Application and system-level software fault tolerance through full system restarts. In *8th International Conference on Cyber-Physical Systems (ICCPS)*, 2017.

4 L. Ahrendts, S. Quinton, T. Boroske, and R. Ernst. Verifying weakly-hard real-time properties of traffic streams in switched networks. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

5 B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. An empirical survey-based study into industry practice in real-time systems. In *41st IEEE Real-Time Systems Symposium (RTSS)*, 2020.

6 S. Altmeyer and R. I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2014.

7 K. J. Åström and T. Hägglund. Revisiting the Ziegler—Nichols step response method for PID control. *Journal of Process Control*, 14(6):635–650, 2004.

8 K. J. Åström and T. Hägglund. *Advanced PID Control.* The Instrumentation, Systems and Automation Society, 2006.

9 K. J. Åström and B. Wittenmark. *Computer-Controlled Systems: Theory and Design.* Prentice Hall, 3rd edition, 1997.

10 G. Bernat and A. Burns. Combining $\binom{n}{m}$-hard deadlines and dual priority scheduling. In *18th IEEE Real-Time Systems Symposium (RTSS)*, pages 46–57, 1997.

11 G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50:308–321, 2001.

12 T. Bund and F. Slomka. Controller/platform co-design of networked control systems based on density functions. In *4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 11–14. ACM, 2014.

13 T. Bund and F. Slomka. Worst-case performance validation of safety-critical control systems with dropped samples. In *23rd International Conference on Real Time and Networks Systems (RTNS)*, pages 319–326. ACM, 2015.

14 G. Buttazzo, M. Velasco, and P. Marti. Quality-of-control management in overloaded real-time systems. *IEEE Transactions on Computers*, 56(2):253–266, 2007.

15 M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *18th IEEE Real-Time Systems Symposium (RTSS)*, pages 330–339, 1997.

16 M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *21st IEEE Real-Time Systems Symposium (RTSS)*, pages 295–304, 2000.

17 B. S. Cazzolato and Z. Prime. On the dynamics of the Furuta pendulum. *Journal of Control Science and Engineering*, 2011.

18 A. Cervin. Analysis of overrun strategies in periodic control tasks. *IFAC Proceedings Volumes*, 38(1):219–224, 2005.

19 A. Cervin, P. Pazzaglia, M. Barzegaran, and R. Mahfouzi. Using JitterTime to analyze transient performance in adaptive and reconfigurable control systems. In *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1025–1032, 2019.

20 A. Chen, Ha. Xiao, A. Haeberlen, and L. T. X. Phan. Fault tolerance and the five-second rule. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

21 H. Choi, H. Kim, and Q. Zhu. Job-class-level fixed priority scheduling of weakly-hard real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 241–253, 2019.

22 R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 168–179, 2013.

23 D. de Niz, L. Wrage, A. Rowe, and R. Rajkumar. Utility-based resource overbooking for cyber-physical systems. In *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 217–226, 2013.

24 L. Desborough. Increasing customer value of industrial control performance monitoring-honeywell's experience. *Preprints of CPC*, pages 153–186, 2001.

25 R. Ernst, S. Kuntz, S. Quinton, and M. Simons. The logical execution time paradigm: New perspectives for multicore systems. *Dagstuhl Reports*, 8:122–149, 2018.

26 G. Frehse, A. Hamann, S. Quinton, and M. Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *35th IEEE Real-Time Systems Symposium (RTSS)*, pages 53–62, 2014.

27 K. Furuta, M. Yamakita, and S Kobayashi. Swing-up control of inverted pendulum using pseudo-state feedback. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 206(4):263–269, 1992.

28 O. Garpinger and T. Hägglund. Software-based optimal PID design with robustness and noise sensitivity constraints. *Journal of Process Control*, 33:90–101, 2015.

29 M. Gaukler, T. Rheinfels, P. Ulbrich, and G. Roppenecker. Convergence rate abstractions for weakly-hard real-time control. *arXiv preprint arXiv:1912.09871*, 2019.

30 S. Kumar Ghosh, S. Dey, D. Goswami, D. Mueller-Gritschneder, and S. Chakraborty. Design and validation of fault-tolerant embedded controllers. In *Design, Automation & Test in Europe Conference Exhibition (DATE)*. IEEE, 2018.

31 D. Goswami, D. Mueller-Gritschneder, T. Basten, U. Schlichtmann, and S. Chakraborty. Fault-tolerant embedded control systems for unreliable hardware. In *International Symposium on Integrated Circuits (ISIC)*. IEEE, 2014.

32 A. Gujarati, M. Nasri, and B. B. Brandenburg. Quantifying the resiliency of fail-operational real-time networked control systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

33 A. Gujarati, M. Nasri, R. Majumdar, and B. B. Brandenburg. From iteration to system failure: Characterizing the fitness of periodic weakly-hard systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

34 M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995.

35 Z. A. H. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 584–589, 2017.

36 Z. A. H. Hammadeh, S. Quinton, and R. Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *14th International Conference on Embedded Software (EMSOFT)*. ACM, 2014.

37 Z. A. H. Hammadeh, S. Quinton, and R. Ernst. Weakly-hard real-time guarantees for earliest deadline first scheduling of independent tasks. *ACM Transactions of Embedded Computing Systems*, 18(6), 2019.

38 Z. A. H. Hammadeh, S. Quinton, M. Panunzio, R. Henia, L. Rioux, and R. Ernst. Budgeting under-specified tasks for weakly-hard real-time systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

39 M. Hertneck, S. Linsenmayer, and F. Allgöwer. Nonlinear dynamic periodic event-triggered control with robustness to packet loss based on non-monotonic lyapunov functions. In *58th IEEE Conference on Decision and Control (CDC)*, pages 1680–1685, 2019.

40 R. Jungers. *The Joint Spectral Radius: Theory and Applications*. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 2009.

41 M. Kauer, D. Soudbakhsh, D. Goswami, S. Chakraborty, and A. M. Annaswamy. Fault-tolerant control synthesis and verification of distributed embedded systems. In *Design, Automation & Test in Europe Conference Exhibition (DATE)*, 2014.

42 F. Khosravi, M. Glaß, and J. Teich. Automatic reliability analysis in the presence of probabilistic common cause failures. *IEEE Transactions on Reliability*, 66(2), 2017.

43 F. Khosravi, M. Müller, M. Glaß, and J. Teich. Uncertainty-aware reliability analysis and optimization. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 97—102, 2015.

44 C. Kirsch and A. Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer Berlin Heidelberg, 2012.

45 G. Koren and D. Shasha. Skip-Over: algorithms and complexity for overloaded systems that allow skips. In *16th IEEE Real-Time Systems Symposium (RTSS)*, pages 110–117, 1995.

46 S. Linsenmayer and F. Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *56th IEEE Conference on Decision and Control (CDC)*, pages 4765–4770, 2017.

47 S. Linsenmayer, M. Hertneck, and F. Allgower. Linear weakly hard real-time control systems: Time- and event-triggered stabilization. *IEEE Transactions on Automatic Control*, 2020.

48 M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein. Control-system stability under consecutive deadline misses constraints. In *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

49 D.C. Montgomery. *Introduction to Statistical Quality Control*. Wiley, 2009.

50 S. Natarajan, M. Nasri, D. Broman, B. B. Brandenburg, and G. Nelissen. From code to weakly hard constraints: A pragmatic end-to-end toolchain for timed C. In *40th IEEE Real-Time Systems Symposium (RTSS)*, pages 167–180, 2019.

51 P. P. O'Connor and A. Kleyner. *Practical Reliability Engineering*. Wiley Publishing, 5th edition, 2012.

52 L. Palopoli, L. Abeni, G. Buttazzo, F. Conticelli, and M. Di Natale. Real-time control system analysis: an integrated approach. In *21st IEEE Real-Time Systems Symposium (RTSS)*, pages 131–140, 2000.

53 P. Pazzaglia, A. Hamann, D. Ziegenbein, and M. Maggio. Adaptive design of real-time control systems subject to sporadic overruns. In *Design, Automation & Test in Europe Conference Exhibition (DATE)*, 2021.

54 P. Pazzaglia, C. Mandrioli, M. Maggio, and A. Cervin. DMAC: Deadline-Miss-Aware Control. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

55 P. Pazzaglia, L. Pannocchi, A. Biondi, and M. Di Natale. Beyond the weakly hard model: Measuring the performance cost of deadline misses. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, 2018.

56 S. Quinton, T. T. Bone, J. Hennig, M. Neukirchner, M. Negrean, and R. Ernst. Typical worst case response-time analysis and its use in automotive network design. In *51st Annual Design Automation Conference (DAC)*, pages 1–6, New York, NY, USA, 2014. ACM.

57 P. Ramanathan. Graceful degradation in real-time control applications using (m,k)-firm guarantee. In *27th IEEE International Symposium on Fault Tolerant Computing*, pages 132–141, 1997.

58 L. Schenato. To zero or to hold control inputs with lossy links? *IEEE Transactions on Automatic Control*, 54(5):1093–1099, 2009.

59 D. Soudbakhsh, L. T. X. Phan, A. M. Annaswamy, and O. Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 5(1):128–141, 2018.

60 D. Soudbakhsh, L. T. X. Phan, O. Sokolsky, I. Lee, and A. Annaswamy. Co-design of control and platform with dropped signals. In *4th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 129–140. ACM, 2013.

61 Y. Sun and M. Di Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Transactions on Embedded Computing Systems*, 16(5s), 2017.

62 G. Vankeerberghen, J. Hendrickx, and R. M. Jungers. JSR: A toolbox to compute the joint spectral radius. In *17th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 151—156. ACM, 2014.

63 N. Vreman and C. Mandrioli. Evaluation of burst failure robustness of control systems in the fog. In A. Cervin and Y. Yang, editors, *2nd Workshop on Fog Computing and the IoT (Fog-IoT)*, volume 80 of *OpenAccess Series in Informatics*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

64 W. Xu, Z. A. H. Hammadeh, A. Kröller, R. Ernst, and S. Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 247–256, 2015.

# On the Convolution Efficiency for Probabilistic Analysis of Real-Time Systems

**Filip Marković**
Mälardalen University, Västerås, Sweden

**Alessandro Vittorio Papadopoulos**
Mälardalen University, Västerås, Sweden

**Thomas Nolte**
Mälardalen University, Västerås, Sweden

## Abstract

This paper addresses two major problems in probabilistic analysis of real-time systems: space and time complexity of convolution of discrete random variables. For years, these two problems have limited the applicability of many methods for the probabilistic analysis of real-time systems, that rely on convolution as the main operation. Convolution in probabilistic analysis leads to a substantial space explosion and therefore space reductions may be necessary to make the problem tractable. However, the reductions lead to pessimism in the obtained probabilistic distributions, affecting the accuracy of the timing analysis. In this paper, we propose an optimal algorithm for down-sampling, which minimises the probabilistic expectation (i.e., the pessimism) in polynomial time.
The second problem relates to the time complexity of the convolution between discrete random variables. It has been shown that quadratic time complexity of a single linear convolution, together with the space explosion of probabilistic analysis, limits its applicability for systems with a large number of tasks, jobs, and other analysed entities. In this paper, we show that the problem can be solved with a complexity of $\mathcal{O}(n\log(n))$, by proposing an algorithm that utilises circular convolution and vector space reductions. Evaluation results show several important improvements with respect to other state-of-the-art techniques.

**2012 ACM Subject Classification** Mathematics of computing → Probabilistic algorithms; Computer systems organization → Real-time system specification

**Keywords and phrases** Probabilistic analysis, Random variables, Algorithm Complexity

**Digital Object Identifier** 10.4230/LIPIcs.ECRTS.2021.16

**Supplementary Material** *Software (ECRTS 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.1.1`

**Funding** This work was supported by the Swedish Research Council (VR) via the project "Practical Probabilistic Timing Analysis of Real-Time Systems (PARIS)", and by the Knowledge Foundation (KKS) via the project "Federated Choreography of an Integrated Embedded Systems Software Architecture (FIESTA)".
*Alessandro Vittorio Papadopoulos*: Swedish Research Council (VR) via the project "Pervasive Self-Optimizing Computing Infrastructures (PSI)".

**Acknowledgements** We want to thank Davor Čirkinagić who borrowed his computing system for performing the evaluation. Also, we are very grateful to the anonymous reviewers for their comments.

## 1 Introduction

In the last decades, probabilistic timing analysis has emerged as an important concept for assurance of real-time guarantees in various fields [10]. Compared to the deterministic-based worst-case execution time model, a probabilistic model of time parameters offers more expressiveness and a higher degree of representation of the actual system behaviour. Besides

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 16; pp. 16:1–16:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that, it can be more applicable for soft real-time systems, and systems with below-worst-case provisioning, that in general are more present than the strictly hard-real time systems.

One of the major concepts that has been used in the analysis of real-time systems is the linear convolution, which is also known as *addition*, when used on discrete random variables. This operation has been studied and adjusted for the analysis of real-time systems in many papers throughout the years [12,19,20,22], while its mere use is present in almost all research areas of probabilistic real-time systems [10].

However, as shown by Davis and Cucu-Grosjean [10], one of the main obstacles for probabilistic timing analysis is the fact that the linear convolution in the worst-case exhibits $\mathcal{O}(n^2)$ time and space complexity. This problem was first addressed with distribution down-sizing techniques [19,20,22,25], and later with the analytical methods [6–8,27,28] for the determination of deadline-miss probability. There are many benefits of using analytical methods, the major one being the reduction of the time and space complexity. However, analytical methods come with some limitations: (i) they often introduce over-approximation in the resulting deadline-miss probabilities [28], and (ii) the most accurate methods rely on convolution, e.g., *Pruning* and *Unify*, proposed by von der Brüggen et al. [28]. Moreover, analytical methods do not provide comprehensive information on the resulting probabilistic distributions. This can be very important for many present problems in the analysis of real-time systems since there are many more potential goals other than determining deadline miss probabilities, e.g., computing cache-miss probability, analysis of random replacement caches [11], analysis of the tasks with multiple probabilistic parameters [18], etc.

**This research.** There is a rich area of past and future research that is limited by the space and time complexity of linear convolution. In this paper, we focus on further reducing the space and time complexity of convolution-based analyses, as a fundamental operator for the probabilistic analysis of real-time systems. One of the main exploration lines seized in this paper is the concept from mathematics and signal processing known as the *circular convolution* [15,23]. The main contributions of this paper are:

- An algorithm for optimal down-sampling of random variables in terms of probabilistic expectation is proposed, which represents the quantitative degree of pessimism when analysing real-time systems. (Section 3)
- Methods which reduce the time complexity of convolution between two random variables from $\mathcal{O}(n^2)$ (linear convolution from state-of-the-art) to $\mathcal{O}(n\log(n))$. (Section 4)

The results of evaluation show that the proposed methods can be applicable for probabilistic analysis of real-time systems even with large numbers of analysed entities.

**Organisation.** The remainder of this paper is organised as follows. In Section 2, we describe the basic terminology and mathematical notation used in the paper. Section 3 describes the proposed algorithm for optimal down-sampling of random variables. Section 4 presents algorithms to reduce the time complexity of addition between two random variables. The evaluation is described in Section 5, the related work is presented in Section 6, and the paper is concluded with Section 7.

## 2 Terminology and mathematical notation

▶ **Definition 1** (Discrete Random Variable). *A discrete random variable $X$ on the probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is defined to be a measurable function $X : \Omega \to \mathbb{R}$ such that the image $X(\Omega)$ is a countable subset of $\mathbb{R}$, and $\{\omega \in \Omega : X(\omega) = x\} \in \mathcal{F}$ for $x \in \mathbb{R}$.*

**Table 1** List of important symbols used in the paper.

| Symbol | Brief explanation |
|---|---|
| $X, Y, Z$ | Discrete random variables in form of two-row matrices. |
| $X'$ | Down-sampled random variable. |
| $\mathbf{V}, \mathbf{W}, \mathbf{Q}$ | One-column vectors representing $X$, $Y$, and $Z$ in the given order. |
| $\lvert\mathbf{V}\rvert$ | Cardinality of vector $\mathbf{V}$. |
| $\mathbf{V} \odot \mathbf{W}$ | Element-wise product between two vectors. |
| $\mathcal{F}\{\mathbf{V}\}$ | Fourier Transform of vector $\mathbf{V}$. |
| $\mathcal{F}^{-1}\{\mathbf{V}\}$ | Inverse Fourier Transform of vector $\mathbf{V}$. |

In the above definition, $\Omega$ is a sample space, the set of all possible outcomes. $\mathfrak{F}$ is an event space, where an event is a set of outcomes in the sample space. $\mathbb{P}$ represents a probability function, that assigns each event in the event space a probability.

The image of $\Omega$ under $X$ is denoted with $\operatorname{Im} X$, and it is the set of values taken by $X$, with positive probability.

Given a random variable $X$, we define the cumulative distribution function of $X$ as $F_X(x) = \mathbb{P}(X \leq x)$, and its expected value (expectation) as $\mathbb{E}(X) = \sum_{x \in \operatorname{Im} X} x \cdot \mathbb{P}(X = x)$.

Throughout the paper, we will use a two-row matrix to represent the mapping between the obtainable values (sorted in increasing order), and their respective probabilities:

$$X \sim \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ \mathbb{P}(X = x_1) & \mathbb{P}(X = x_2) & \cdots & \mathbb{P}(X = x_n) \end{bmatrix}, \tag{1}$$

where the symbol $\sim$ denotes that the random variable $X$ has the probability distribution described by the two-row matrix, and $n$ is the cardinality of $\operatorname{Im} X$.

▶ **Definition 2** (Usual Stochastic order [26]). *Two random variables $X$ and $Y$, with cumulative distribution functions $F_X$ and $F_Y$, are said to be in the usual stochastic order, denoted as $X \succeq Y$, if and only if $\forall x$, $F_X(x) \leq F_Y(x)$.*

▶ **Definition 3** (Independence). *Two (discrete) random variables $X$ and $Y$ are independent if the pair of events $\{X = x\}$ and $\{Y = y\}$ are independent for all $x, y \in \mathbb{R}$. Formally,*

$$\mathbb{P}(X = x, Y = y) = \mathbb{P}(X = x)\mathbb{P}(Y = y) \qquad \textit{for all} \quad x, y \in \mathbb{R}.$$

▶ **Definition 4** (Convolution or sum of random variables). *If $X$ and $Y$ are independent discrete random variables on $(\Omega, \mathfrak{F}, \mathbb{P})$, then $Z = X + Y$ has the mass function*

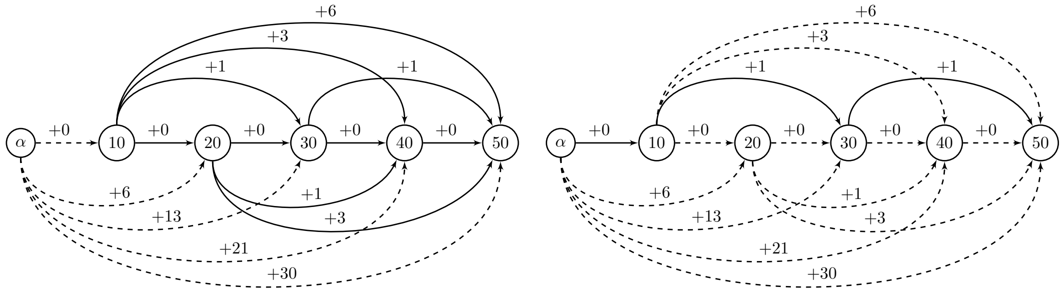
$$\mathbb{P}(Z = z) = \sum_{x=-\infty}^{\infty} \mathbb{P}(X = x)\mathbb{P}(Y = z - x) \qquad \textit{for } z \in \mathbb{R}.$$

▶ **Definition 5** (Element-wise product). *The element-wise product (also known as, Hadamard product, point-wise product, and Schur product) between two matrices $\mathbf{A}$ and $\mathbf{B}$ of the same dimension $m \times n$ is denoted as $\mathbf{A} \odot \mathbf{B}$.*

▶ **Definition 6** (Discrete convolution between two vectors). *Given two vectors $\mathbf{V}$ and $\mathbf{W}$ of equal length $n$, the discrete convolution between the two vectors is defined as*

$$(\mathbf{V} * \mathbf{W}) = \sum_{i=1}^{n} \mathbf{V}(i)\mathbf{W}(n-i) = \sum_{i=1}^{n} \mathbf{W}(i)\mathbf{V}(n-i),$$

*where $\mathbf{V}(i)$ (and $\mathbf{W}(i)$) is the $i$-th element of the vector $\mathbf{V}$ (of $\mathbf{W}$).*



**(a)** Potential paths for down-sampling. **(b)** Optimal path (solid).

**Figure 1** Graph representation of the additional expectations from the running example.

The majority of the above definitions are available in the book [13]. Definition 2 is cited from [26], while it is described by Diaz et al. [12] as the first-order stochastic dominance. In Table 1, we provide the list of the most important and frequently used symbols in the paper.

# 3 Down-sampling of Random Variables

We divide this section into two main parts. Section 3.1 describes the problem of optimal down-sampling with respect to minimising the probabilistic expectation. We solve this problem by proposing an algorithm that uses a dynamic programming approach. Section 3.2 proposes a linear down-sampling algorithm, which exhibits an improved time complexity compared to the optimal algorithm, at the cost of an increased over-approximation in some cases.

## 3.1 Optimal down-sampling of random variables

▶ **Problem 1** (Optimal down-sampling of random variables). *Given a discrete random variable $X$ where $n$ is the cardinality of* $\operatorname{Im} X$*, down-sample the random variable to cardinality $s$, $s < n$, such that the expectation of the derived variable $X'$ is minimised, while $X' \succeq X$.*

Upon down-sampling, the probabilistic expectation of the down-sampled random variable $X'$ is often larger than the expectation of the original random variable $X$ since the probability mass in $X'$ has to be shifted to the larger values in order to account that $X'$ upper bounds $X$ (Definition 2). For this reason, Maxim et al. [19] showed that expectation can be regarded as the metric of pessimism when the random variable is down-sampled from its original cardinality to a lower one. Many down-sampling algorithms were proposed in the state-of-the-art to reduce the probabilistic expectation upon down-sampling (e.g. Max-k re-sampling, Pessimism reduce re-sampling [19], etc.). In this section, we show how the random variable can be down-sampled to cardinality $s$ such that its probabilistic expectation is less than or equal to the expectation of any other potential down-sampling of the same cardinality. To better clarify the problem, we introduce the following running example.

▶ Example 1. Initially, the random variable $X$ contains the following values with the assigned probabilities (expressed with the notation of Equation (1)):

$$X \sim \begin{bmatrix} 10 & 20 & 30 & 40 & 50 \\ 0.6 & 0.1 & 0.1 & 0.1 & 0.1 \end{bmatrix}, \qquad \mathbb{E}(X) = 20. \tag{2}$$

The goal is to derive the down-sampled variable $X'$ of cardinality 3 such that $\forall x \in \operatorname{Im} X$, $F_{X'}(x) \leq F_X(x)$ and such that $\mathbb{E}[X']$ is minimised.

Problem 1 can be solved in two steps.

**Step 1.** For any two values $k, l \in \operatorname{Im} X$ with $k < l$, compute the additional probabilistic expectation that would be present in the resulting variable $X'$ if $k, l \in \operatorname{Im} X'$ and no other value between $k$ and $l$ is present in $\operatorname{Im} X'$.

Regarding the running example, the additional expectations can be represented with the weighted graph (see Figure 1a), such that vertices represent values, and solid edges represent additional expectation considering the case when two connected values are selected to be in the down-sampled variable.

**Step 2.** Use dynamic programming to select $s$ values prior the last value of $X$, such that the total additional expectation is minimised. Considering the graph representation (see Figure 1a) the problem is equivalent to the problem of finding the path of size $s$ from the artificial zero node (denoted $\alpha$), to the last node of the graph. The zero node is added such that path can start from any node, i.e. any value in $X$, without necessarily starting from the first value in $\operatorname{Im} X$.

In the remainder of the section, we formalise different terms that are used in the algorithm, which is followed by the pseudo code and the algorithm description.

▶ **Definition 7.** *Additional expectation $\mathbb{E}_{k,l}(X)$ that is introduced in $X'$ when only $l$ is present in $\operatorname{Im} X'$ within interval $(k, l]$, is defined with the following equation*

$$\mathbb{E}_{k,l}(X) = l \sum_{i=k+1}^{l} \mathbb{P}(X = i) - \sum_{i=k+1}^{l} i\, \mathbb{P}(X = i) = \sum_{i=k+1}^{l} (l - i)\mathbb{P}(X = i). \tag{3}$$

In the first equality term of Equation (3), we consider that $\mathbb{E}_{k,l}(X)$ is equal to the difference between the following two terms:

1. $l \sum_{i=k+1}^{l} \mathbb{P}(X = i)$, which is the expectation in the interval $(k, l]$ when all the values between $k$ and $l$ are removed. Meaning that for the removed values their probabilities are accumulated to the value $l$ in order to preserve safety (see Definition 2).
2. $\sum_{i=k+1}^{l} i\, \mathbb{P}(X = i)$, which is the original expectation of $X$ within the interval $(k, l]$.

The expression is further simplified in order to account for only one traversal from $k$ to $l$. From the example in Figure 1a, $\mathbb{E}_{20,40} = (40 - 30) \cdot 0.1 = +1$.

Considering the graph representation of the problem, where nodes represent values from $X$, and edges represent the potential selection, a weight on the edge between two nodes, $k$ and $l$, is equal to the additional expectation $\mathbb{E}_{k,l}(X)$ defined in Equation (3). Thus, Problem 1 is analogous to the problem of finding a path of size $s$ to the last value from $X$ such that cost (expectation) is minimised. We use plus symbols in the weights to express the concept of *additional* expectation that is introduced in the down-sampled random variable (RV) compared to the original one. To solve such problem, we propose Algorithm 1.

### 3.1.1 Algorithm description

Algorithm 1 takes the input variables: (i) $X$, which is the discrete random variable to be down-sampled, and (ii) $s$, which is the cardinality for down-sampling. Additionally to the down-sampled variable $X'$, the algorithm also outputs the difference $E_n$ between the expectation of the original random variable and the down-sampled one.

**Algorithm 1** Optimal down-sampling of a discrete random variable.

```
   Data: Discrete random variable $X$, cardinality $s$ for down-sampling
   Result: Down-sampled random variable $X'$
1  function optimalDS($X, s$):
2      $n \leftarrow \max(\operatorname{Im} X)$
3      $\alpha = \min(\operatorname{Im} X) - 1$
4      for $l \in \operatorname{Im} X$ do
5          $E_l \leftarrow \mathbb{E}_{\alpha,l}(X)$
6          $V_l \leftarrow \emptyset$
7      end
8      for $i \leftarrow 1$ to $s$ by $1$ do
9          for $l \in \operatorname{Im} X$ do
10             $E'_l \leftarrow +\infty$
11             $V'_l \leftarrow \emptyset$
12         end
13         for $k, l \in \operatorname{Im} X : k < l$ do
14             if $E'_l > E_k + \mathbb{E}_{k,l}$ then
15                 $E'_l \leftarrow E_k + \mathbb{E}_{k,l}$
16                 $V'_l \leftarrow V_k \cup k$
17             end
18         end
19         for $l \in \operatorname{Im} X$ do
20             $E_l \leftarrow E'_l$
21             $V_l \leftarrow V'_l$
22         end
23     end
24     $X' \leftarrow$ RV such that $\operatorname{Im} X' = V_n$ and $\mathbb{P}(X' = x') = \sum_{x=\operatorname{prec}(x')+\epsilon}^{x'} \mathbb{P}(X = x)$, where
        $\operatorname{prec}(x')$ is the value preceding $x'$ from $\operatorname{Im} X'$, and $\operatorname{prec}(\min(\operatorname{Im} X')) = \min(\operatorname{Im} X)$
25 return $X', E_n$
```

In line 3, the algorithm computes the artificial value $\alpha$ which represents the first value that precedes the minimum value from $\operatorname{Im} X$ (see Figure 1a). Then, in line 5, $\alpha$ is used to compute the minimum expectation $E_l$ that is imposed if we select some value $l$ from $X$ as the first selected for the down-sampled variable, without anymore considering the values that precede $l$. The additional expectation of performing such a step is equal to $\mathbb{E}_{\alpha,l}(X)$ (see Figure 1a). Since at the beginning, no value is still selected, for each value $l$ from $\operatorname{Im} X$, we initialise $V_l$ which is the set of values whose selection yields the minimum expectation until the value $l$. In lines 8–23, the algorithm computes and updates the minimum expectations that can be imposed until any value $l$ from $X$, upon selection of at most $i$ values that precede $l$. Upon accounting for each new selection, the minimum expectation until each value $l$ is updated (line 20), together with the set $V_l$ of selected values that yield the minimised expectation (line 21). After doing this step $s$ times, the $V_n$ represents the set that yields $E_n$, while $E_n$ represents the minimum additional expectation until the last value $n$ (line 2) upon selection of $s$ values prior the last one. This is equivalent to the dynamic programming problem: $E[i+1, l] = \min_{k<l}\{E[i,k] + \mathbb{E}_{k,l}\}$, where $E[a, x]$ is the minimum additional expectation until value $x$ after $a$ selections.

### 3.1.2 Algorithm correctness

We prove that the algorithm minimises the additional expectation by using induction.

**Proof.** By induction.
*Induction hypothesis:* For $l \in \operatorname{Im} X$, after the $i$-th iteration of the *for* loop at line 8, $E_l$ represents the minimum additional expectation that can be produced until value $l$, considering at most $i$ selected values prior $l$, while $V_l$ represents the set of selected $i$ values whose joint expectation yields $E_l$.
*Base case:* Before the loop at line 8 starts, the number of iterations is $i = 0$, $E_l$ is equal to the expectation $\mathbb{E}_{\alpha,l}(X)$ which is also the minimum additional expectation imposed without selecting any value that precedes $l$, as follows from Equation (3). Also, $V_l$ is an empty set, representing that no value preceding $l$ is selected. Thus, the base case holds.
*Inductive step:* We now prove that the induction hypothesis holds at the end of $i+1$ iteration of the for loop (lines 8–23). At the beginning of the $i+1$ iteration, for every value $l \in \operatorname{Im} X$, Algorithm 1 identifies the respective preceding value $k$ such that the expectation of selecting $k$ prior to $l$ yields the minimum expectation (lines 13–18). This is the case because the expectation is minimised over the all possible choices of $k : k < l \wedge k \in \operatorname{Im} X$ (line 13). Also, it follows from the induction hypothesis that until value $k$ the expectation is minimised with at most $i$ selected values before $k$. Then, the derived expectation $E_l'$ at the end of the loop (line 18) is the minimum expectation that can be produced with at most $i+1$ selected values until $l$, where $k$ is the additional $(+1)$ selected value. This further means that $E_l$ also represents the demanded minimum possible expectation since after line 20, $E_l = E_l'$. Analogically, the same holds for the set of selected values $V_l$ that yields $E_l$ since the set is computed as the union of 1) values which yield the minimum expectation $E_k$, as follows from the induction hypothesis, and 2) value $k$ which is the additional selected value that yields the minimum $E_l$ as shown previously. Finally, after $s$ iterations, it follows that the algorithm computes the minimum additional expectation $E_n$ which is the result of selecting $s$ values prior the last one, $n$. Analogously, it holds that $V_n$ stores the selected values whose selection yields the minimum added expectation. Thus, the random variable $X'$ (line 24) resulting from the selected values in $V_n$, yields the minimum additional expectation $E_n$ among the other possible random variables of the same cardinality, constructed from values in $X$, also upper-bounding $X$. This concludes the proof. ◀

### 3.1.3 Time complexity

The worst-case time complexity of Algorithm 1 is $\mathcal{O}(n^3)$ since there are at most $((n+1)n)/2$ expectations of form $\mathbb{E}_{k,l}$ to be computed, and for each such computation, Equation 3 has linear complexity, which finally results in $\mathcal{O}((n+1)n)\mathcal{O}(n) = \mathcal{O}(n^3)$.

## 3.2 Linear Down-sampling

In this subsection, we propose Algorithm 2 for down-sampling of random variables, that exhibits linear time complexity. The main idea behind the algorithm is uniform down-sampling of a probability distribution. The algorithm starts from the first value of the original random variable and it iterates until the last one (line 6). It assigns the subset of values from $\operatorname{Im} X$ such that the cumulative probability between consecutive selected values in $X'$ is as uniform as possible, under the condition of $X' \succeq X$. The considered terms are:

- $P^{un}$ – unassigned probability sum, i.e., part of the distribution that is not present in the current version of the down-sampled variable $X'$,
- $s$ – number of values to be assigned to the down-sampled variable $X'$,
- $p_\delta$ – probability sum threshold, which controls that the number of values in the resulting down-sampled variable does not exceed the predefined cardinality $s$,

**Algorithm 2** Linear down-sampling of a discrete random variable.

```
   Data: Discrete random variable $X$, cardinality $s$ for down-sampling
   Result: Down-sampled random variable $X'$
1  function linearDS($X, s$):
2      $P^{un} \leftarrow 1$ // sum of the unassigned probability values from $X$ to $X'$
3      $p_\delta \leftarrow P^{un}/s$
4      $\operatorname{Im} X' \leftarrow \emptyset$
5      $p_\Sigma \leftarrow 0$
6      for $l \in \operatorname{Im} X$ in an increasing order do
7          $p_\Sigma \leftarrow p_\Sigma + \mathbb{P}(X = l)$
8          $P^{un} \leftarrow P^{un} - \mathbb{P}(X = l)$
9          if $p_\Sigma \geq p_\delta$ then
10             $\operatorname{Im} X' \leftarrow \operatorname{Im} X' \cup l$ such that $\mathbb{P}(X = l) = p_\Sigma$
11             $s \leftarrow s - 1$
12             $p_\delta \leftarrow P^{un}/s$
13             $p_\Sigma \leftarrow 0$
14         end
15     end
16     $X' \leftarrow$ RV such that $\operatorname{Im} X' = V_n$ and $\mathbb{P}(X' = x') = \sum_{x=\operatorname{prec}(x')+\epsilon}^{x'} \mathbb{P}(X = x)$, where
         $\operatorname{prec}(x')$ is the value preceding $x'$ from $\operatorname{Im} X'$, and $\operatorname{prec}(\min(\operatorname{Im} X')) = \min(\operatorname{Im} X)$
17 return $X'$
```

- $p_\Sigma$ – probability sum from the last selected value until the currently observed one,
- $\operatorname{Im} X'$ – set of selected values for $X'$.

Given the above terms, in each iteration the algorithm assigns the currently observed value $l \in \operatorname{Im} X$ to the down-sampled variable $X'$, but only in case the probability sum $p_\Sigma$, from the last selected value in $X'$, exceeds the probability threshold $p_\delta$ (line 9). Probability threshold $p_\delta$ maintains uniformity of the probability distribution and does not allow that more than $s$ values are selected within $X'$. This is achieved by constant re-computation of the still unassigned probability $P^{un}$, whenever a new value is assigned to $X'$ (line 10). Also, whenever a new value is assigned to $X'$, value $p_\Sigma$ is set to zero to account for the fact that no new value and respective probability is assigned from the last assignment (line 13). In case when $l$ is not selected, $p_\Sigma$ is updated with the probability resulting from $l$ itself (line 8). At the end, the algorithm returns the down-sampled random variable $X'$ (line 17).

## 3.3 Description of the algorithms using the running example

Given the random variable $X$ from Example 1, and the process of down-sampling to the cardinality of 3, Algorithm 1 selects the values 10, 30, and 50 as depicted in Figure 1b with solid arrows above the values. This is the shortest path of size 2 until the last value 50, with non-zero probability. Note that in order to select three values, Algorithm 1 should be invoked with $s = 2$ since the largest value from $\operatorname{Im} X$ will always be selected (line 24).

Algorithm 2 selects the same two values using a different approach. For the desired cardinality $s = 3$, it first computes that the probability sum threshold $p_\delta$ is equal to $1/3 = 0.333$ (line 3). Then, by trying to uniformly distribute the cumulative probability sum, it immediately at value 10 (lines 6 – 8) identifies that the threshold is exceeded since $0.3 < \mathbb{P}(X = 10)$ (line 9), and adds 10 to the down-sampled variable (line 10). By recomputing the threshold (line 12), for the remainder of the distribution, it derives the new threshold $p_\delta = (1 - 0.6)/2 = 0.2$ since 0.4 is the unassigned probability sum, and there are two more

$$\text{(a)}\quad (X,Y) \xRightarrow[\text{multiplication}]{\text{addition}} Z'' \xRightarrow[\text{sorting}]{\text{sorting}} Z' \xRightarrow[\text{addition}]{\text{normalisation}} Z$$

$$\text{(b)}\quad (X,Y) \sim (\mathbf{V},\mathbf{W}) \xrightarrow{\mathcal{F}\{\cdot\}} (\hat{\mathbf{V}},\hat{\mathbf{W}}) \xrightarrow{\text{element-wise product}} \hat{\mathbf{Q}} \xrightarrow{\mathcal{F}^{-1}\{\cdot\}} \mathbf{Q} \sim Z$$

**Figure 2** Overview of (a) linear, and (b) circular convolutions for computing $X + Y = Z$.

values to be selected. At $l = 20$, its probability is $\mathbb{P}(X = 10) = 0.1$, thus the threshold is not reached, but in the next iteration $l = 30$, it is (line 9, i.e. $0.1+0.1 \geq 0.2$). At the end, the sum of the remaining unassigned probability mass is equal to $P^{un} = (1-0.6-0.2) = 0.2$, and there is only one value to be selected. This means that the algorithm selects the last value 50 and assigns to it the probability of 0.2. For both algorithms, $\mathbb{E}(X') = 22$ and $X' = \left[\begin{smallmatrix} 10 & 30 & 50 \\ 0.6 & 0.2 & 0.2 \end{smallmatrix}\right]$.

## 4 Efficient convolution

Probabilistic timing analysis techniques suffer from a large time complexity that is essentially attributable to the linear convolution operator [10].

▶ **Problem 2** (Efficient convolution of random variables). *Improve the efficiency of computing the exact result of the convolution between two random variables.*

In the following, we describe different ways of computing the sum of two random variables, and we present different improvements that can be used to reduce their time complexity.

Let $X$ and $Y$ be independent discrete random variables, such that

$$X \sim \begin{bmatrix} x_1 & \cdots & x_n \\ \mathbb{P}(X = x_1) & \cdots & \mathbb{P}(X = x_n) \end{bmatrix}, \quad Y \sim \begin{bmatrix} y_1 & \cdots & y_m \\ \mathbb{P}(Y = y_1) & \cdots & \mathbb{P}(Y = y_m) \end{bmatrix}.$$

The addition $X + Y$ of the two random variables is the random variable $Z$ whose probability distribution is characterised by the convolution between the probability distributions of $X$ and $Y$, and it is defined as:

$$\mathbb{P}(Z = z) = \sum_{k=0}^{+\infty} \mathbb{P}(X = k)\mathbb{P}(Y = z - k), \quad \operatorname{Im} Z = \{z | \forall x \in \operatorname{Im} X, \forall y \in \operatorname{Im} Y, z = x + y\}.$$

**Linear convolution.** The linear convolution (also known as *canonical convolution*) is performed in three steps, as described by Milutinović et al. [22]. We show this in Figure 2(a), where the symbol $\xRightarrow[\text{p}]{v}$ denotes that operation $v$ is performed on the values of the random variables (indicated above the arrow), and that operation $p$ is applied to the respective probability distributions (indicated below the arrow). Reading the figure from the left, we start with $X$ and $Y$, with their respective probability distributions. Then, the algebraic addition of each possible pair of values in $\operatorname{Im} X$ and in $\operatorname{Im} Y$ is computed, and for each computed value the multiplication of the respective probability distribution is computed, obtaining the variable $Z''$. Then, the values in $\operatorname{Im} Z''$ are sorted in increasing order, and the associated probabilities are sorted accordingly, thus deriving the variable $Z'$. At the end, a normalisation is performed on the values of $Z'$ such that the repeated values are combined, and their respective probabilities are summed. Such algorithm has a time complexity of $\mathcal{O}(n \cdot m)$, where $n$ and $m$ are the cardinalities of $\operatorname{Im} X$ and $\operatorname{Im} Y$, respectively. For more details, refer to paper by Milutinović et al. [22].

**Circular convolution.** To solve Problem 2, we build upon the idea of *circular convolution*. There are many mathematical sources that explain the benefit of circular over the linear convolution [15, 23]. The circular convolution idea is shown in Figure 2(b). Starting from $X$ and $Y$, we first compute vectors $\mathbf{V}$ and $\mathbf{W}$, respectively, such that vector indexes represent values, while the vector elements represent probabilities of the corresponding random variable. Note that in Figure 2(b), we use $\xrightarrow{e}$ to represent that operation $e$ is performed on the vector elements. On the vectors $\mathbf{V}$ and $\mathbf{W}$, we perform the Fourier Transformation operation, obtaining the new vectors $\hat{\mathbf{V}}$ and $\hat{\mathbf{W}}$. Then, we perform the element-wise product between the transforms, deriving the transform $\hat{\mathbf{Q}}$. Next, we compute the inverse Fourier transformation of $\hat{\mathbf{Q}}$, deriving the vector $\mathbf{Q}$, which characterises random variable $Z$, i.e., indexes of $\mathbf{Q}$ represent values of $Z$, while vector elements represent probabilities of $Z$. In Figure 2, this is denoted with $\mathbf{Q} \sim Z$.

The above-described process of computing the Fourier and inverse Fourier transformations, together with the element-wise product of two transforms, is known as *circular convolution*, and it has a time complexity of $\mathcal{O}(d \log(d))$, where $d = \max(\operatorname{Im} Z) - \min(\operatorname{Im} Z)$. We discuss the whole process formally, in more detail, in the remaining part of this section.

## 4.1 Formal description of the circular convolution of random variables

Let us start from the convolution theorem [1,3,5]. In the following formulation of the theorem we narrow it to vectors although it holds for more complex mathematical entities.

▶ **Theorem 1** (Fourier's convolution theorem [24]). *The Fourier transform of the convolution of two vectors $\mathbf{V}$ and $\mathbf{W}$ is equal to the element-wise product of the Fourier transforms of the two vectors, i.e.*

$$\mathcal{F}\{(\mathbf{V} * \mathbf{W})\} = \mathcal{F}\{\mathbf{V}\} \odot \mathcal{F}\{\mathbf{W}\}. \tag{4}$$

In the above equation, $\odot$ represents linearly complex element-wise multiplication between vectors $\mathcal{F}\{\mathbf{V}\}$ and $\mathcal{F}\{\mathbf{W}\}$ of Fourier coefficients. To compute the convolution of $\mathbf{V}$ and $\mathbf{W}$, we can compute the inverse Fourier transform of the right side of the equality, and this process is known as the circular convolution. However, the result of the circular convolution $\operatorname{cconv}(\mathbf{V}, \mathbf{W})$ of two vectors $\mathbf{V}$ and $\mathbf{W}$ is equal to the result of the corresponding linear convolution, (as presented in [15, 21]) when the following equations hold

$$\begin{aligned} \operatorname{cconv}(\mathbf{V}, \mathbf{W}) = \mathcal{F}^{-1}\left\{\hat{\mathbf{V}} \odot \hat{\mathbf{W}}\right\}, \text{ where } \hat{\mathbf{V}} &= \mathcal{F}\{\mathbf{V}^\frown 0_v\}, \\ \hat{\mathbf{W}} &= \mathcal{F}\{\mathbf{W}^\frown 0_w\}, \\ v &= \operatorname{nptwo}(|\mathbf{V}| + |\mathbf{W}| - 1) - |\mathbf{V}|, \\ w &= \operatorname{nptwo}(|\mathbf{V}| + |\mathbf{W}| - 1) - |\mathbf{W}|, \end{aligned} \tag{5}$$

where $\operatorname{nptwo}(a)$ is a function that returns the first power of two greater than or equal to some value $a \in \mathbb{N}$, and $\frown$ is a concatenation operator. In the above equation, vectors $\mathbf{V}$ and $\mathbf{W}$ first need to be zero-padded to the same size, and that size must be greater than or equal to the the sum of their respective sizes minus one, as shown by Langton and Levin [15]. This size, in the equation, is represented by the following term $|\mathbf{V}| + |\mathbf{W}| - 1$. The zero-padding of the vectors is performed with the vector concatenation operator ($\frown$). This operator is used between the desired vector (e.g. $\mathbf{V}$) and the zero-column vector (e.g. $0_v$). The concatenation of the zero-column vector to the desired vector leads to the desired zero-padding. Similar computations are defined for the vector $\mathbf{W}$ as well, considering the zero-column vector $0_w$.

Additionally, in the equation, the zero-padding is increased to reach the size that is equal to the first power of two that succeeds $|\mathbf{V}| + |\mathbf{W}| - 1$. This step, of computing the $\operatorname{nptwo}(\cdot)$ function, is performed in order to allow for the linearithmic time complexity that can be achieved by using the Discrete Fast Fourier Transformation, known also as Cooley-Tukey algorithm [9].

Without the loss of generality, for the sake of the simplified equations and the running example, suppose that $\{0, 1, 2, \dots, b\}$, where $b \in \mathbb{N}$, is the support of discrete random variables $X$ and $Y$.

Considering sum $Z$ of random variables $X$ and $Y$, it can be computed using Equation (5) such that for discrete random variables $X$ and $Y$, their probabilities are represented as the elements of vectors $\mathbf{V}$ and $\mathbf{W}$ respectively, as shown in the following equations.

$$\begin{aligned}
&\mathbf{V} = (v_j) = \mathbb{P}(X = j) \wedge j \in \{0, \dots, m_X\}\ , \text{ where } m_X = \max(\{x \mid x \in \operatorname{Im} X\}),\\
&\mathbf{W} = (w_j) = \mathbb{P}(Y = j) \wedge j \in \{0, \dots, m_Y\}\ , \text{ where } m_Y = \max(\{y \mid y \in \operatorname{Im} Y\}),\\
&\mathbf{Q} = \operatorname{cconv}(\mathbf{V}, \mathbf{W}) = (q_j) \text{ and } Z \sim \begin{bmatrix} 0 & \dots & j & \dots & m_X + m_Y \\ q_0 & \dots & q_j & \dots & q_{m_X+m_Y} \end{bmatrix}.
\end{aligned} \tag{6}$$

▶ Example 2.

$$X \sim \begin{bmatrix} 200 & 300 \\ 0.6 & 0.4 \end{bmatrix} \qquad Y \sim \begin{bmatrix} 150 & 200 \\ 0.6 & 0.4 \end{bmatrix} \qquad Z \sim \begin{bmatrix} 350 & 400 & 450 & 500 \\ 0.36 & 0.24 & 0.24 & 0.16 \end{bmatrix}$$

$$\mathbf{V} = \begin{array}{cl} \begin{bmatrix} 0 \\ \vdots \\ 0.6 \\ 0 \\ \vdots \\ 0.4 \end{bmatrix} & \begin{matrix} 0 \\ \\ 200 \\ 201 \\ \\ 300 \end{matrix} \end{array} \qquad \mathbf{W} = \begin{array}{cl} \begin{bmatrix} 0 \\ \vdots \\ 0.6 \\ 0 \\ \vdots \\ 0.4 \end{bmatrix} & \begin{matrix} 0 \\ \\ 150 \\ 151 \\ \\ 200 \end{matrix} \end{array} \qquad \mathbf{Q} = \begin{array}{cl} \begin{bmatrix} \vdots \\ 0.36 \\ \vdots \\ 0.24 \\ \vdots \\ 0.24 \\ \vdots \\ 0.16 \\ \vdots \end{bmatrix} & \begin{matrix} \\ 350 \\ \\ 400 \\ \\ 450 \\ \\ 500 \\ 511 \end{matrix} \end{array}$$

Starting from the random variable $X$, in Equation (6), we first define vector $V$ such that its index $j$ represents all the values in the domain of $X$, starting from 0 until the last value $m_X \in X$ that has non-zero probability of occurrence. Then, the $j$-th value $v_j$ in $V$ represents the probability $P(X = j)$. Similar is performed for vector $W$ but considering the random variable $Y$. Then, using Equation (5) we compute the circular convolution of $V$ and $W$ thus deriving vector $Q$, whose $j$-th value is equal to the probability $P(X + Y = j)$. Finally, as follows from Theorem 1 and Equation (6), the vector $Q$ contains the combined information on probability values from $\operatorname{Im} X + Y$.

In the above example, the omitted numbers represent probabilities equal to zero. The majority of computations between those zeros can be avoided, while still maintaining the exact computation of the final result. This is the problem that we solve in the following subsection by proposing two methods for vector reductions such that the computations derive the exact result.

## 4.2 Fast and efficient computation of the exact result

Compared to the solution from Equation (6), it is possible to derive the resulting random variable $Z$ with more efficient computations.

▶ **Improvement 1.** *Reducing the vector sizes using the greatest common divisor.*

In this solution, the improvement is made by using the greatest common divisor among the values from $\operatorname{Im} X$ and $\operatorname{Im} Y$. As follows from Definition 4, a value with zero probability of occurrence cannot lead to the non-zero probability value in $Z$. The same holds for the analogous Equation (6). Thus, we can improve the computation by considering the minimum quantum that considers all non-zero values from both equations, as follows:

$$\delta = \mathrm{GCD}(\operatorname{Im} X \cup \operatorname{Im} Y), \tag{7}$$

$$\mathbf{V} = (v_j) = \mathbb{P}(X = j) \wedge j \in \{0, \dots, m_X\} \text{ , where } m_X = 1/\delta \cdot \max(\{x \mid x \in \operatorname{Im} X\}), \tag{8}$$

$$\mathbf{W} = (w_j) = \mathbb{P}(Y = j) \wedge j \in \{0, \dots, m_Y\} \text{ , where } m_Y = 1/\delta \cdot \max(\{y \mid y \in \operatorname{Im} Y\}), \tag{9}$$

$$\mathbf{Q} = \operatorname{cconv}(\mathbf{V}, \mathbf{W}) = (q_j), \tag{10}$$

$$Z \sim \begin{bmatrix} 0 \cdot \delta & \dots & j \cdot \delta & \dots & (m_X + m_Y) \cdot \delta \\ q_0 & \dots & q_j & \dots & q_{m_X + m_Y} \end{bmatrix}. \tag{11}$$

With the above equation, the sum $Z$ of $X$ and $Y$ from Example 2 would look as follows:

$$\delta = 50,\ \mathbf{V} = \begin{array}{c} \begin{bmatrix} 0 \\ \vdots \\ 0.6 \\ 0 \\ 0.4 \end{bmatrix} \end{array} \begin{array}{l} \scriptstyle 0 \\ \\ \scriptstyle 4 \\ \scriptstyle 5 \\ \scriptstyle 6 \end{array},\ \mathbf{W} = \begin{bmatrix} 0 \\ \vdots \\ 0.6 \\ 0.4 \end{bmatrix} \begin{array}{l} \scriptstyle 0 \\ \\ \scriptstyle 3 \\ \scriptstyle 4 \end{array},\ \mathbf{Q} = \operatorname{cconv}(\mathbf{V}, \mathbf{W}) = \begin{bmatrix} \vdots \\ 0.36 \\ 0.24 \\ 0.24 \\ 0.16 \\ \vdots \end{bmatrix} \begin{array}{l} \\ \scriptstyle 7 \\ \scriptstyle 8 \\ \scriptstyle 9 \\ \scriptstyle 10 \\ \scriptstyle 15 \end{array} \tag{12}$$

$$Z \sim \begin{bmatrix} 350 = 7 \cdot 50 & 400 = 8 \cdot 50 & 450 = 9 \cdot 50 & 500 = (6 + 4) \cdot 50 \\ 0.36 & 0.24 & 0.24 & 0.16 \end{bmatrix}.$$

The essence of the above described transformation is to change the base metric unit such that unnecessary computations are avoided. Compared to Example 2, the vector sizes are roughly 30 times less in the above equation, which also propagates to computation time.

▶ **Improvement 2.** *Reducing the vector sizes by removing the starting zero intervals.*

In this improvement we focus on the starting values of the vectors $\mathbf{V}$ and $\mathbf{W}$. Consider Example 2 and the follow-up Equation (12), both vectors $\mathbf{V}$ and $\mathbf{W}$ start with probabilities equal to zero, followed by more zero probabilities that represent values that are not in $\operatorname{Im} X$ and $\operatorname{Im} Y$. We show that both vectors can be reduced by ignoring starting zero intervals, thus starting from the probabilities of the minimum values in $\operatorname{Im} X$ and $\operatorname{Im} Y$, without losing computation precision. Let us start from random variables $X$ and $Y$:

$$X \sim \begin{bmatrix} x_1 & \dots & x_n \\ \mathbb{P}(X = x_1) & \dots & \mathbb{P}(X = x_n) \end{bmatrix}, \quad Y \sim \begin{bmatrix} y_1 & \dots & y_m \\ \mathbb{P}(Y = y_1) & \dots & \mathbb{P}(Y = y_m) \end{bmatrix}. \tag{13}$$

▶ **Proposition 2.** *Given two random variables $X$ and $Y$, the convolution $X + Y$ is*

$$X{+}Y \sim \begin{bmatrix} x_1 + y_1 \\ 1 \end{bmatrix} + \begin{bmatrix} x_1 - x_1 & \dots & x_n - x_1 \\ \mathbb{P}(X = x_1) & \dots & \mathbb{P}(X = x_n) \end{bmatrix} + \begin{bmatrix} y_1 - y_1 & \dots & y_m - y_1 \\ \mathbb{P}(Y = y_1) & \dots & \mathbb{P}(Y = y_m) \end{bmatrix}. \tag{14}$$

**Proof.**

$$\begin{aligned} &\begin{bmatrix} x_1 + y_1 \\ 1 \end{bmatrix} + \begin{bmatrix} x_1 - x_1 & \dots & x_n - x_1 \\ \mathbb{P}(X = x_1) & \dots & \mathbb{P}(X = x_n) \end{bmatrix} + \begin{bmatrix} y_1 - y_1 & \dots & y_m - y_1 \\ \mathbb{P}(Y = y_1) & \dots & \mathbb{P}(Y = y_m) \end{bmatrix} \\ &= \begin{bmatrix} x_1 \\ 1 \end{bmatrix} + \begin{bmatrix} x_1 - x_1 & \dots & x_n - x_1 \\ \mathbb{P}(X = x_1) & \dots & \mathbb{P}(X = x_n) \end{bmatrix} + \begin{bmatrix} y_1 \\ 1 \end{bmatrix} + \begin{bmatrix} y_1 - y_1 & \dots & y_m - y_1 \\ \mathbb{P}(Y = y_1) & \dots & \mathbb{P}(Y = y_m) \end{bmatrix} \\ &= \begin{bmatrix} x_1 & \dots & x_n \\ \mathbb{P}(X = x_1) & \dots & \mathbb{P}(X = x_n) \end{bmatrix} + \begin{bmatrix} y_1 & \dots & y_m \\ \mathbb{P}(Y = y_1) & \dots & \mathbb{P}(Y = y_m) \end{bmatrix} \sim X + Y \end{aligned}$$

◀

The benefit of using Proposition 2 is observable when we want to generate vectors $\mathbf{V}$ and $\mathbf{W}$. Consider creating vectors from random variable $X \sim \left[\begin{smallmatrix}1000 & 1001\\ 0.4 & 0.6\end{smallmatrix}\right]$ and $Y \sim \left[\begin{smallmatrix}1005 & 1006\\ 0.4 & 0.6\end{smallmatrix}\right]$. Instead of generating vectors of sizes measured in thousand units, we can simply apply the proposition and derive the following term:

$$\left[\begin{smallmatrix}1000 & 1001\\ 0.4 & 0.6\end{smallmatrix}\right] + \left[\begin{smallmatrix}1005 & 1006\\ 0.4 & 0.6\end{smallmatrix}\right] = 1000 + 1005 + \left[\begin{smallmatrix}0 & 1\\ 0.4 & 0.6\end{smallmatrix}\right] + \left[\begin{smallmatrix}0 & 1\\ 0.4 & 0.6\end{smallmatrix}\right], \tag{15}$$

and therefore we can represent random variables with two vectors, each having size of two.

Based on Proposition 2 and Improvement 1, we introduce the following computations.

$$X' = X - x_1,\ Y' = Y - y_1,\ \delta = \mathrm{GCD}(\mathrm{Im}\,X' \cup \mathrm{Im}\,Y'), \tag{16}$$

$$\mathbf{V} = (v_j) = \mathbb{P}(X' = j) \wedge j \in [0, \dots, m_X\}\ , \text{ where } m_X = 1/\delta \cdot \max(\{x \mid x \in \mathrm{Im}\,X'\}), \tag{17}$$

$$\mathbf{W} = (w_j) = \mathbb{P}(Y' = j) \wedge j \in [0, \dots, m_Y\}\ , \text{ where } m_Y = 1/\delta \cdot \max(\{y \mid y \in \mathrm{Im}\,Y'\}), \tag{18}$$

$$\mathbf{Q} = \mathrm{cconv}(\mathbf{V}, \mathbf{W}) = (q_j), \tag{19}$$

$$Z \sim \begin{bmatrix} x_1 + y_1 + 0 \cdot \delta & \dots & x_1 + y_1 + j \cdot \delta & \dots & x_1 + y_1 + (m_X + m_Y) \cdot \delta \\ q_0 & \dots & q_j & \dots & q_{m_X + m_Y} \end{bmatrix}. \tag{20}$$

The above set of computations is almost the same as the one for the first improvement, but the major difference is that we use Proposition 2 and therefore compute the circular convolution only for variables $X'$ and $Y'$ which results in the furthermore reduced vectors $\mathbf{V}$ and $\mathbf{W}$. In Equation (20), the result of $X + Y$ is restored by applying the proposition $(x_1 + y_1 + j \times \delta)$. We show the vector size reduction in the running example:

$$x_1 + y_1 = 200 + 150 = 350,\ X' \sim \begin{bmatrix} 0 & 100 \\ 0.6 & 0.4 \end{bmatrix},\ Y' \sim \begin{bmatrix} 0 & 50 \\ 0.6 & 0.4 \end{bmatrix},\ \delta = 50,$$

$$\mathbf{V} = \begin{array}{c|c} \begin{bmatrix} 0.6 \\ 0 \\ 0.4 \end{bmatrix} & \begin{smallmatrix} 0 \\ 1 \\ 2 \end{smallmatrix} \end{array},\ \mathbf{W} = \begin{array}{cc} \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix} & \begin{smallmatrix} 0 \\ 1 \end{smallmatrix} \end{array},\ \mathbf{Q} = \mathrm{cconv}(\mathbf{V}, \mathbf{W}) = \begin{array}{cc} \begin{bmatrix} 0.36 \\ 0.24 \\ 0.24 \\ 0.16 \end{bmatrix} & \begin{smallmatrix} 0 \\ 1 \\ 2 \\ 3 \end{smallmatrix} \end{array},$$

$$Z \sim \begin{bmatrix} 350 = 350 + 0 \cdot 50 & 400 = 350 + 1 \cdot 50 & 450 = 350 + 2 \cdot 50 & 500 = 350 + 3 \cdot 50 \\ 0.36 & 0.24 & 0.24 & 0.16 \end{bmatrix}.$$

In this simple example we reduced the number of elements in vectors from 512 values (as in Example 2) to 4 values at most, still deriving the exact random variable $Z$.

**Impact of the improvements on the probabilistic timing analysis.** Improvements 1 and 2 may simplify and improve the efficiency of many analysis types that are characterised with the iterative computation of the increasing probabilistic distribution.

▶ Example 3 (Computation of the probability distribution). Consider the computation of the probabilistic distribution from two tasks ($\tau_1$ and $\tau_2$) whose probabilistic execution times are defined with the following random variables $C_1 \sim \left[\begin{smallmatrix}1000 & 1001\\ 0.4 & 0.6\end{smallmatrix}\right]$ and $C_2 \sim \left[\begin{smallmatrix}1005 & 1006\\ 0.4 & 0.6\end{smallmatrix}\right]$. If we want to compute the probabilistic distribution that involves 100 jobs of $C_1$ and 200 jobs of $C_2$, we can simply use Proposition 2 and derive the following result: $100 \cdot C_1 + 200 \cdot C_2 = 100 \cdot 1000 + 200 \cdot 1005 + S$, where random variable $S$ is characterised with $\mathcal{F}^{-1}\left\{\bigodot_1^{100} \hat{\mathbf{V}}_1 \odot \bigodot_1^{200} \hat{\mathbf{V}}_2\right\}$.

Vectors $\hat{\mathbf{V}}_1$ and $\hat{\mathbf{V}}_2$ are the Fourier transforms of vectors that represent $C_1$ and $C_2$ (each having size of two). The important benefit is that the random variable $S$, which follows the summation of the deterministic values, can be derived from the inverse Fourier transformation of the vector of 600 elements $(2 \cdot 100 + 2 \cdot 200)$, compared to the naive method from Equation (6), where the final vector size would be 301600 elements.

This is a huge improvement in the computation efficiency, while the magnitude of space and time reduction becomes even greater with each new job that may be considered in the analysis.

▶ Example 4 (Computation of the deadline miss probability). Let us consider the very same two tasks ($\tau_1$ and $\tau_2$) from the previous example, and assume that we want to compute the deadline miss probability at time instant $D_2 = 2000$. Furthermore, $\tau_1$ can be released at most two times until $D_2$, while $\tau_2$ can be released at most once. Instead of performing the entire computation, we can first apply Proposition 2 and check where the image of the resulting distribution starts, i.e. what is the first value $s$ with non-zero probability of occurrence in the final distribution. Thus we derive that $s = 2 \cdot 1000 + 1 \cdot 1005 = 3005$. Since the image of the summed distribution starts from 3005, and since $s > D_2$, this implies that $\mathbb{P}(C_1 + C_1 + C_2 > 2000) = 1$ and there is no need to perform convolutions. Such an improvement reduces the number of probabilistic computations for such types of problems.

## 4.3 Efficient repetitive convolutions

In timing analysis, there are often cases when one variable is summed multiple times, e.g. request bound function (RBF) of form $RBF_i(\Delta) = C_i \cdot \alpha(\Delta)$ where $C_i$ is the worst-case execution time of a task, $\Delta$ is the time interval under consideration, and $\alpha(\Delta)$ represents a function that upper-bounds the number of arrivals of jobs of $\tau_i$ within some time interval of length $\Delta$. It has been shown by Bozhko and Brandenburg [4] that RBF can be used to concretize many existing types of response-time analysis, under various taskset model assumptions, and thus it represents the fundamental computation in the analysis of real-time systems.

Analogously, as shown in many probabilistic timing analysis papers, e.g., [12, 19], probabilistic request bound can be computed by consecutive addition of the random variables that represent upper-bounded probabilistic execution time of a task. Similar computation is necessary in many other areas of real-time system analysis, e.g., probability of cache miss and hit, etc. Therefore, in this section we describe the algorithm that efficiently computes consecutive additions of the same random variable, based on the consecutive Hadamard product of the Fourier coefficients, using the powers of two[1].

Algorithm 3 computes the result of $n$ convolutions of random variable $X$. This computation can be achieved by creating the sum vector $\hat{\mathbf{V}}_{sum} = \mathbf{1}$, of all values equal to one, (line 5) which is iteratively multiplied $n$ times with the Fourier Transform $\mathcal{F}\{\mathbf{V}\}$ of vector $\mathbf{V}$, where $\mathbf{V}$ characterises $X$ (line 4). However, instead of that, we can perform fewer multiplications by constantly computing the power vector $\hat{\mathbf{V}}'$, which is initially set to $\mathcal{F}\{\mathbf{V}\}$ (line 6). We show it by the following example.

Let us assume that we need 9 multiplications of $\mathcal{F}\{\mathbf{V}\}$. In its binary form, 9 is equal to $\mathcal{B}(9) = 1001$, and let us traverse to each bit of $\mathcal{B}(9)$ one by one (lines 7 – 16). On each bit shift (line 9), we first multiply $\hat{\mathbf{V}}_{sum}$ with $\hat{\mathbf{V}}'$ (line 11) if the bit at the $i$-th index of $\mathcal{B}(9)$ is equal to 1 (line 10). Then, regardless of the previous computation, after each shift we compute $\hat{\mathbf{V}}'$ as the power of itself (line 13). Following this rule, the computations are:

[1] It has been shown by Milutinovic et al. [22] that the power of two technique reduces computation time also for linear convolution.

**Algorithm 3** Fast computation of consecutive summations of a random variable.

```
   Data: $X$ – random variable, $n$ – number of necessary convolutions
   Result: Sum $X_\Sigma$ equal to $n$ additions of $X$
1  function fastSum($X, n$):
2      $s \leftarrow n \cdot \min(\{x \mid x \in \mathrm{Im}\, X\}$ // direct application of Proposition 2
3      $m_X \leftarrow \max(\{x \mid x \in \mathrm{Im}\, X\}) - \min(\{x \mid x \in \mathrm{Im}\, X\}$
4      $\mathbf{V} \leftarrow$ compute using Equation (6) and zero-pad until $n \cdot m_X$
5      $\hat{\mathbf{V}}_{sum} \leftarrow$ column vector of elements equal to 1, of size $|\mathbf{V}|$
6      $\hat{\mathbf{V}}' \leftarrow \mathcal{F}\{\mathbf{V}\}$
7      $k, i \leftarrow 0$
8      while $k \neq n$ do
9          $b \leftarrow \mathcal{B}(n, i)$ // $\mathcal{B}(n, i)$ is a function that checks the binary representation of $n$ and
             returns the value at index $i$
10         if $b = 1$ then
11             $\hat{\mathbf{V}}_{sum} \leftarrow \hat{\mathbf{V}}_{sum} \odot \hat{\mathbf{V}}'$
12         end
13         $\hat{\mathbf{V}}' \leftarrow \hat{\mathbf{V}}' \odot \hat{\mathbf{V}}'$
14         $k \leftarrow k + b \times 2^i$
15         $i \leftarrow i + 1$
16     end
17     $\mathbf{V} \leftarrow \mathcal{F}^{-1}\left\{\hat{\mathbf{V}}_{sum}\right\}$ // where $v_j$ is the $j$-th element in $\mathbf{V}$
18     $X_\Sigma \leftarrow \begin{bmatrix} s+0 & \dots & s+j & \dots & s+n \cdot m_X \\ v_0 & \dots & v_j & \dots & v_{n \cdot m_X} \end{bmatrix}$
19 return $X_\Sigma$
```

**Initial values.** $\hat{\mathbf{V}}_{sum} \leftarrow \mathbf{1}$ and $\hat{\mathbf{V}}' \leftarrow \mathcal{F}\{\mathbf{V}\}$

$(i = 0 \text{ and } \mathcal{B}(9, i) = 1) \;:\; \hat{\mathbf{V}}_{sum} \leftarrow (\hat{\mathbf{V}}_{sum} \odot \hat{\mathbf{V}}') = \mathcal{F}\{\mathbf{V}\}$ and $\hat{\mathbf{V}}' \leftarrow (\hat{\mathbf{V}}')^2 = (\mathcal{F}\{\mathbf{V}\})^2$

$(i = 1 \text{ and } \mathcal{B}(9, i) = 0) \;:\; \hat{\mathbf{V}}' \leftarrow (\hat{\mathbf{V}}')^2 = (\mathcal{F}\{\mathbf{V}\})^4$

$(i = 2 \text{ and } \mathcal{B}(9, i) = 0) \;:\; \hat{\mathbf{V}}' \leftarrow (\hat{\mathbf{V}}')^2 = (\mathcal{F}\{\mathbf{V}\})^8$

$(i = 3 \text{ and } \mathcal{B}(9, i) = 1) \;:\; \hat{\mathbf{V}}_{sum} \leftarrow (\hat{\mathbf{V}}_{sum} \odot \hat{\mathbf{V}}') = (\mathcal{F}\{\mathbf{V}\} \odot (\mathcal{F}\{\mathbf{V}\})^8) = (\mathcal{F}\{\mathbf{V}\})^9$

At the end of this process, $\hat{\mathbf{V}}_{sum}$ is equal to $(\mathcal{F}\{\mathbf{V}\})^9$. The very same principle is used in the algorithm (lines 8 – 16). At the end of the algorithm (lines 17 and 18), it just computes the inverse Fourier Transform of $\hat{\mathbf{V}}_{sum}$, which then characterises $X_\Sigma$.

**Space-complexity reduction of repetitive computations.** Although the time complexity is improved with the above-mentioned methods, there may be the cases where space complexity can still be an issue despite the vector reductions proposed by Improvements 1 and 2. E.g. after $k$ additions of some random variable $X$, the resulting random variable in the worst-case may have the image that is $k$ times greater than the one of $X$.

This problem can be further addressed by using the principles implied by the central limit theorem in probability theory. In more details, considering the case when one random variable $X$ is added to itself multiple times, with each new addition the resulting sum tends more towards the normal distributions, even though the $X$ is not normally distributed. Consider for the moment Figure 3, we show the original random variable $X$, the random variable after three additions of $X$, and the random variable after 20 additions of $X$. You can see that by each new summation, the resulting sum resembles more to the normal distribution, although initially it has quite opposite properties compared to it.

Therefore, Algorithm 3 may be improved by applying the down-sampling methods on the inverse Fourier transforms of $\hat{\mathbf{V}}_{sum}$ and $\hat{\mathbf{V}}'$. The proposed down-sampling algorithms are tailored to decrease the vector sizes by safely removing the starting zero tail that contains

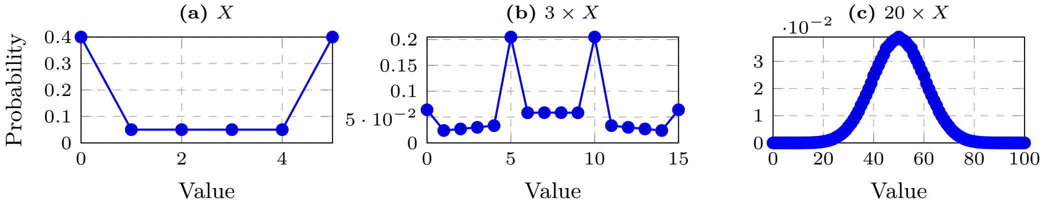


**Figure 3** Consecutive additions of the same random variable. The points are connected in order to show the resemblance to the continuous normal distribution.

close-to-zero probabilities. The removed zero interval can then be accounted by Proposition 2. Additionally, the down-sampling method known as domain-quantisation [19] can be an efficient way to further reduce the vector sizes and enable Improvement 1 from Section 4.2.

# 5 Evaluation

The evaluation is organised in three parts: (i) Evaluation of the down-sampling algorithms, (ii) Evaluation of the convolution computations, and (iii) Computation of the deadline miss probability. The code of the implementation is available (see [17]).

**Hardware and software configuration.** In all of the experiments, we used a PC with i7 4770k CPU with a frequency of 2.6 GHz, and 32 GB of RAM memory. All the algorithms are implemented in MATLAB, using Advanpix Multiprecision Computing Toolbox [16].

## 5.1 Evaluation of the down-sampling algorithms

**Goal of the evaluation and the evaluated entities.** In this evaluation, we compared the two proposed down-sampling algorithms, Optimal (OP), from Section 3, and Linear (LN), from Section 3, with Domain Quantisation (DQ), and Pessimism Reduce (PR), both proposed by Maxim et al. [19]). The later two algorithms are selected since PR introduces the least pessimism in the down-sampled variables among the state-of-the-art algorithms, while DQ is known as the fastest down-sampling algorithm [19]. The comparison between the algorithms is made according to three criteria: (a) Computation time of the algorithms as a function of the size of the initial random variable, (b) Probabilistic expectation, i.e., pessimism introduced upon down-sampling, (c) Probability of exceeding the median. The later is evaluated since the probability of exceeding some value is often used in probabilistic analysis, e.g., deadline miss probability, that is, the probability of exceeding the deadline.

**Experiment Setup.** In this experiment, we considered $|\mathbf{V}| \in \{200, 400, 600, 800, 1000\}$, and for every cardinality $|\mathbf{V}|$ we sampled 1000 realisations of the random variable $X$ associated with $\mathbf{V}$, using UUniFast algorithm [2]. The experiments analyse the performance of the down-sampling operation to the maximum size of 20 values, for all the 5000 realisations.

**Experiment Results.** Figure 4(a) shows the average computation time of down-sampling as a function of the cardinality of the initial realisation of the vector $\mathbf{V}$. The results show that the average computation time of OP is impacted the highest (4.7s for the initial size of 1000 values), compared to the other down-sampling approaches. This is due to its cubic time complexity. For the other algorithms, execution time did not exceed 0.01 seconds. According

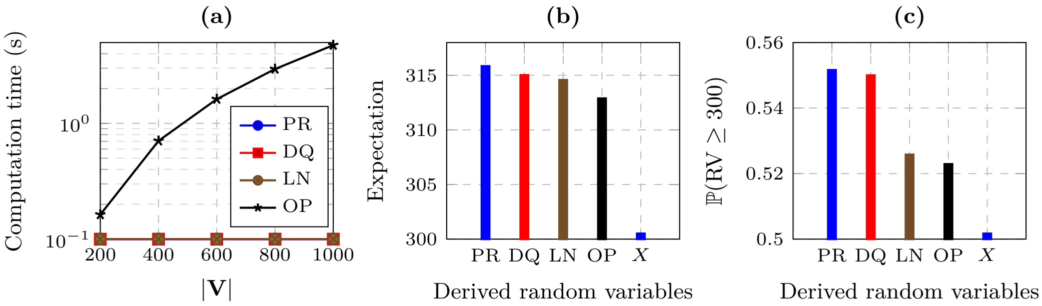


**Figure 4** Results of the evaluation on the comparison of down-sampling algorithms.

to MATLAB documentation for *tic toc* functions, execution times which do not exceed the 0.1 seconds are not representative with enough confidence and for this reason are not reported. The average execution time of PR ($1\mu$s for the cardinality 1000) seems to be greater than the average execution time of LN and DQ (below $100\mu$s for the cardinality 1000), but this comparison demands for the further, more precise investigation.

Figure 4(b) shows the average probabilistic expectation, for the evaluated algorithms and the expectation of the initial random variable ($X$ in the figure). This figure presents the data only for $|\mathbf{V}| = 600$. OP succeeds to minimise the additional expectation the most, compared to the other algorithms, followed by LN.

Figure 4(c) shows the average probability of exceeding the median, for $|\mathbf{V}| = 600$. The results show that the probability of exceeding 300, is least when using OP, followed by LN.

**Discussion.** In the above experiments, we showed that OP and LN can introduce less pessimism compared to the state-of-the-art algorithms. This is relevant when the down-sampled variable needs to be used in the probabilistic analysis since the pessimism linearly grows with each new addition to the other variables or itself. The improvements of OP come with the computation cost. Since OP is a parallelisable algorithm, based on dynamic programming, the computation improvements can be further addressed. During the evaluation, we also discovered that each of the evaluated algorithms may derive the best down-sampling with respect to exceedence probability. Therefore, the potential improvement may arise by combining DQ, LN, PR, and OP, which remains for the future work.

## 5.2 Evaluation of the convolution algorithms

**Goal of the evaluation and the evaluated entities.** In this evaluation, we compared Algorithm 3, referred in the following as CC (for circular convolution) with the analogous method that uses the power-based addition, for linear convolution, proposed by Milutinović et al. [22], referred as LC (linear convolution). LC is the fastest method in the state-of-the-art for computing the linear convolution of random variables and we implemented it with the linear-convolution function (*conv*) in MATLAB. The comparison criterion in the experiment was the average computation time as a function of the number of convolutions, analogous to the number of summed jobs in the probabilistic schedulability analysis.

**Experiment Setup.** In this experiment, we generated three random variables, using UUniFast algorithm [2], with size $|\mathbf{V}| \in \{100, 1000, 10000\}$. The results are shown in Figure 5. For each realisation we report the computation time as a function of the number of convolutions

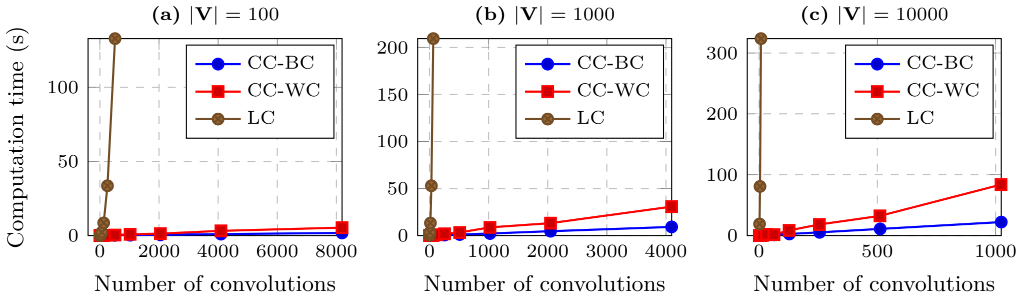


**Figure 5** Computation time as a function of number of additions, e.g., number of analysed jobs.

performed. The above-described generation of the probability distribution did not affect the computation times, and therefore we used only one variable per setup. Since Algorithm 3 performs the least number of vector multiplications when the number of convolutions is a power of two, and the largest number of multiplications when the number of convolutions is equal to a power of two minus one, we reported those two lines separately. CC-WC represents the worst-case computation time (worst-case number of vector multiplications) with ticks that are by one less than the power of two, while BC-WC represents the best-case computation time, ticks being powers of two. The two modes of computation do not significantly affect LC, and therefore we reported only the results for ticks equal to powers of two.

**Experiment Results.** Figure 5(a) shows the case of $|\mathbf{V}| = 100$. We observe that LC takes 132s to compute 512 convolutions, while CC-WC takes only 5s for $8191 = 2^{13} - 1$ convolutions. Due to the smaller number of multiplications, CC-BC takes only 1.7s for performing $8192 = 2^{13}$ convolutions. A similar trend can be observed for $|\mathbf{V}| \in \{1000, 10000\}$ in Figure 5(b)–(c). Finally, it can observed that the size of the $\mathbf{V}$ has a significant impact on the computation time. In fact, looking at CC-WC for $|\mathbf{V}| = 10000$, when the number of convolutions is $1023 = 2^{10} - 1$, the average computation time is 83 seconds, while for CC-BC when the number of convolutions is $1024 = 2^{10}$, the computation time is only 22 seconds.

**Discussion.** The difference of time complexities of linear and circular convolution can be observed by the derived results. The proposed algorithm, based on the circular convolution and vector reductions defined in the paper, outperforms the linear-convolution-based algorithm in this experiment setup. However, it should be stated that there are cases when LC can provide better computation times compared to CC since the computation time of LC depends on the cardinality of the images of random variables, while the computation time of the circular convolution depends on the difference between the largest and the lowest values within the respective images. Regardless of this, the scaling of the circular convolution is better due to a lower time-complexity class.

## 5.3 Computation of the deadline miss probability

**Goal of the evaluation and the evaluated entities.** In order to show the applicability of the proposed computations in the state-of-the-art probabilistic analysis, we implemented the deadline miss probability analysis (DMPA), according to Equation (3) described by von der Brüggen et al. [28], using however the circular convolution for variable additions, instead of the linear convolution (as shown in [18]). In the implementation, we also used Improvement 2 from Section 4.2. Regarding the evaluation, we compared the circular-convolution-based

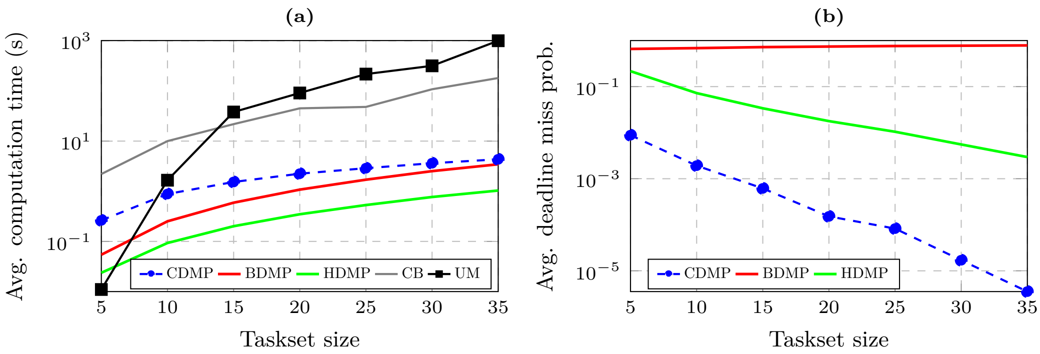


**Figure 6** Computation time as a function of a taskset size, measuring deadline miss probability.

approach (CDMP), with the two fastest DMPA approximations, i.e., the Hoeffding bound (HDMP), and the Bernstein bound (BDMP), as defined by von der Brüggen et al. [28]. To the best of the authors' knowledge, these two approximation approaches are the fastest in the state-of-the-art for the given problem. Their drawback is that they tend to over-approximate the deadline-miss probabilities. Compared to the timing values reported by von der Brüggen et al. [28], our implementations for HDMP and BDMP achieve an average speedup factor of 6; for the sake of completeness, we therefore report the average computation time also of the other approaches presented in [28], scaling the timing values according to the speedup factor. The additional methods are: Unify method (UM), proposed by von der Brüggen et al. [28] and the method based on the Chernoff bound (CB), which was proposed by Chen and Chen [6]. UM derives the exact deadline-miss probability, while CB is more accurate and efficient compared to UM, but less efficient approximation compared to HDMP and BDMP.

**Experiment Setup.** In this experiment, we replicated the setup proposed by von der Brüggen et al. [28]. Thus, 1000 tasksets were generated per each presented point, considering the sizes of $5, 10, \dots, 35$. For each taskset we generated the utilisations using UUniFast [2], with the taskset utilisation set to 0.7. Task periods were generated according to a log-uniform distribution ranging from 10ms to 1000ms. Normal execution modes were computed by multiplying the utilisation and the period for each generated pair of values. The periods and normal execution mode times were ceiled to be multiples of $50\mu$s. This allowed for the discrete random-variable support while not severely affecting the taskset utilisation given by UUniFast. The implementation of the circular convolution did not include Improvement 1 from the paper, in order to not take the advantage of the experiment setup that considers only two execution time modes. Implicit-deadlines were assumed. Probability of occurrence for the normal execution was set to 0.975, while the probability of occurrence for the abnormal execution mode was set to 0.025. Abnormal execution was set to be two times greater than the normal execution. For more details on the setup, refer to their paper.

**Experiment Results.** Figure 6(a) shows that CDMP is applicable for computation of the deadline-miss probability. With taskset size of 35, CDMP achieves to compute the deadline miss probability of the lowest priority task, taking 4.3s, while BDMP takes 3.4s, and HDMP around 1s. The projected computation times for CB and UM are still several times larger despite the speedup scaling, although these results may be improved or worsened based on the implementation choices in MATLAB and the underlined hardware. In Figure 6(b), we show the average deadline miss result computed by CDMP, BDMP, and HDMP. Since the result of CDMP is the exact deadline miss probability, we observe that the BDMP estimation becomes more pessimistic with the increase of the taskset size, while the HDMP bound is still significantly worse than the result derived with CDMP (note the log scale).

**Discussion.** We conclude that the methods proposed in this paper are promising for the deadline miss analysis since the computation time is low, while the results are exact. However, the further empirical investigations should address the impact of the more complex execution time distributions and taskset parameters. For example, using $1\mu s$ as the basic unit of time for random variables may increase the computation time of the method due to necessary vector increase. Also, further investigations should take into account more complex assumptions, e.g. probabilistic executions and inter-arrival times, as proposed by Maxim et al. [18].

## 6 Related work

As described by Davis and Cucu-Grosjean [10,11], the issues of computational intractability of the linear convolution in the probabilistic schedulability and timing analysis are investigated for years in the research area of real-time systems. There are two main research directions that tried to solve this problem: (i) Down-sampling methods, and (ii) Analytical methods. The goal of the down-sampling methods is to approximate the random variables such that its image size is reduced. Then, the linear convolution can be applied, however introducing over-approximation. There are several works in this domain, e.g., Refaat et al. [25], Diaz et al. [12], Kim et al. [14]. More recent improvement was made by Maxim et al. [19, 20] where several down-sampling algorithms were proposed, among which are *domain quantisation* and *reduced pessimism*. Both were used in the evaluation of this paper. Finally, the most recent improvement was made by Milutinović et al. [22] addressing the cache-miss probability analysis. In their work, several improvements for speeding up the linear convolution were proposed, using parallel computations and power operations, while they also pointed that the use of circular convolution may be relevant for addressing the tractability issues. Contrary to the above methods, Chen and Chen [6] proposed an analytical approach for computing the deadline-miss probability, which is based on the *Chernoff bounds*. Following this line of research, von der Brüggen et al. [27, 28] proposed several algorithms for approximating the deadline miss probability (based on the Hoeffding and the Bernstein inequalities). Next to those, they also proposed two exact methods: *Pruning*, which is a multinomial-based approach combined with the pruning techniques, and *Unify*, which is a combination of *Pruning* with the approach based on the union of equivalence classes. More recently Chen et al. [7] proposed the improvement for the Chernoff bounds approximation, which is solved by considering an equivalent convex optimisation problem. This improvement reduces the computation time of deriving the approximation, while it also improves its accuracy.

## 7 Conclusions

In this paper, we addressed two problems that consider random variables and their use in the analysis of probabilistic real-time systems. The first problem considered the space reduction, i.e., the down-sampling of a random variable. This is the problem of approximation, such that the distribution of the random variable is preserved while its set of values is reduced. Such process in the probabilistic analysis often introduces pessimism, that then propagates towards the final results of the probabilistic response-time and similar types of analysis. We proposed an optimal algorithm for down-sampling, which minimises upon probabilistic expectation, i.e., pessimism introduced upon down-sampling.

The second addressed problem considers the efficiency of computing the convolution between random variables. This problem for years limited the applicability of many existing and possibly future work in the domain of probabilistic analysis of real-time systems. In this paper, we showed how the circular convolution can be used to address this problem, reducing the time complexity of a single discrete convolution from $O(n^2)$ to $O(n \log(n))$. Using the circular convolution with the vector reductions proposed in the paper, we showed in the evaluation that the proposed approach shows promising results with respect to its applicability in the existing problems of probabilistic analysis.

## References

1 George B. Arfken and Hans J. Weber. Mathematical methods for physicists, 1999.

2 Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, 2005.

3 Jonathan M. Blackledge. *Digital image processing: mathematical and computational methods.* Elsevier, 2005.

4 Sergey Bozhko and Björn B Brandenburg. Abstract response-time analysis: A formal foundation for the busy-window principle. In *Euromicro Conf. Real-Time Syst. (ECRTS 2020)*, 2020.

5 Ronald Newbold Bracewell. *The Fourier transform and its applications.* McGraw-Hill, 1999.

6 Kuan-Hsun Chen and Jian-Jia Chen. Probabilistic schedulability tests for uniprocessor fixed-priority scheduling under soft errors. In *IEEE Int. Symp. Industrial Emb. Syst. (SIES)*, pages 1–8, 2017.

7 Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. Efficient computation of deadline-miss probability and potential pitfalls. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 896–901, 2019.

8 Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Analysis of deadline miss rates for uniprocessor fixed-priority scheduling. In *IEEE Int. Conf. Emb. and Real-Time Computing Syst. and Applications (RTCSA)*, pages 168–178, 2018.

9 James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

10 Robert Ian Davis and Liliana Cucu-Grosjean. A survey of probabilistic schedulability analysis techniques for real-time systems. *LITES: Leibniz Trans. Emb. Syst.*, pages 1–53, 2019.

11 Robert Ian Davis and Liliana Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES: Leibniz Transactions on Embedded Systems*, pages 1–60, 2019.

12 Jose Luis Diaz, Jose Maria Lopez, Manuel Garcia, Antonio M Campos, Kanghee Kim, and Lucia Lo Bello. Pessimism in the stochastic analysis of real-time systems: Concept and applications. In *IEEE Int. Real-Time Syst. Symp. (RTSS)*, pages 197–207, 2004.

13 Geoffrey Grimmett and Dominic Welsh. *Probability: an introduction.* Oxford U. Press, 2014.

14 Kanghee Kim, Jose Luis Diaz, Lucia Lo Bello, José María López, Chang-Gun Lee, and Sang Lyul Min. An exact stochastic analysis of priority-driven periodic real-time systems and its approximations. *IEEE Transactions on Computers*, 54(11):1460–1466, 2005.

15 Charan Langton and Victor Levin. *The Intuitive Guide to Fourier Analysis and Spectral Estimation.* Mountcastle Company, 2017.

16 Advanpix LLC. Multiprecision computing toolbox for MATLAB. URL: `http://www.advanpix.com/`.

17 Filip Marković, Alessandro Vittorio Papadopoulos, and Thomas Nolte. Artifact-evaluation—on-the-convolution-efficiency, 2021. URL: `https://github.com/Aeoliphile/Artifact-Evaluation---On-the-convolution-efficiency`.

18 Dorin Maxim and Liliana Cucu-Grosjean. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 224–235. IEEE, 2013.

19 Dorin Maxim, Mike Houston, Luca Santinelli, Guillem Bernat, Robert I Davis, and Liliana Cucu-Grosjean. Re-sampling for statistical timing analysis of real-time systems. In *Int. Conf. Real-Time and Network Syst. (RTNS)*, pages 111–120, 2012.

20 Dorin Maxim, Luca Santinelli, and Liliana Cucu-Grosjean. Improved sampling for statistical timing analysis of real-time systems. *Int. Conf. Real-Time and Network Syst. (RTNS)*, pages 17–20, 2010.

21 Stephen McGovern. MATLAB Central File Exchange, Fast Convolution. `https://www.mathworks.com/matlabcentral/fileexchange/5110-fast-convolution`. Accessed: 2021-01.

22 Suzana Milutinović, Jaume Abella, Damien Hardy, Eduardo Quiñones, Isabelle Puaut, and Francisco J Cazorla. Speeding up static probabilistic timing analysis. In *Int. Conf. Architecture of Computing Syst. (ARCS)*, pages 236–247, 2015.

23 Alan V Oppenheim, John R Buck, and Ronald W Schafer. *Discrete-time signal processing. Vol. 2.* Upper Saddle River, NJ: Prentice Hall, 2001.

24 Athanasios Papoulis. The fourier integral and its applications. *McCraw-Hill*, 1962.

25 Khaled S Refaat and Pierre-Emmanuel Hladik. Efficient stochastic analysis of real-time systems via random sampling. In *Euromicro Conf. Real-Time Syst. (ECRTS)*, pages 175–183, 2010.

26 Moshe Shaked and J. George Shanthikumar, editors. *Univariate Stochastic Orders*, pages 3–79. Springer New York, New York, NY, 2007.

27 Georg von der Brüggen. *Realistic Scheduling Models and Analyses for Advanced Real-Time Embedded Systems.* PhD thesis, TU Dortmund (Germany), 2019.

28 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. In *Euromicro Conf. Real-Time Syst. (ECRTS)*, 2018.