Workshop on Next Generation Real-Time Embedded Systems

NG-RES 2020, January 21, 2020, Bologna, Italy

Edited by

Marko Bertogna
Federico Terraneo

Editors

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

Federico Terraneo
Politecnico di Milano, Italy
federico.terraneo@polimi.it

ACM Classification 2012
Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems

ISBN 978-3-95977-136-8

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/978-3-95977-136-8.

Publication date
January, 2020

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.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0):
https://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.NG-RES.2020.0

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface
Marko Bertogna and Federico Terraneo 0:vii

Invited Talk

SDN for Dynamic Reservations on Real-Time Networks
Luis Almeida 1:1–1:1

Regular Paper

Energy Minimization in DAG Scheduling on MPSoCs at Run-Time: Theory and Practice
Bertrand Simon, Joachim Falk, Nicole Megow, and Jürgen Teich 2:1–2:13

Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems
José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto 3:1–3:14

A Low Energy FPGA Platform for Real-Time Event-Based Control
Silvano Seva, Claudia Esther Lukaschewsky Mauriziano, William Fornaciari, and Alberto Leva 4:1–4:11

Real-Time Task Migration for Dynamic Resource Management in Many-Core Systems
Behnaz Pourmohseni, Fedor Smirnov, Stefan Wildermann, and Jürgen Teich 5:1–5:14

Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020).
Editors: Marko Bertogna and Federico Terraneo
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume collects the papers presented at the first edition of the Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020). The workshop is co-located with the 2020 edition of the HiPEAC conference and was held at Bologna, Italy on January 21th, 2020.

The traditional concept of embedded systems is constantly evolving to address the requirements of the modern world. Cyber-physical systems, networked control systems and Industry 4.0 are introducing an increasing need for interconnectivity. A steadily increasing algorithmic complexity of embedded software is fueling the adoption of multicore and heterogeneous architectures. As a consequence, meeting real-time requirements is now more challenging than ever. The NG-RES workshop focuses on real-time embedded systems, with particular emphasis on the distributed and parallel aspects. The workshop is a venue for both the networking and multicore real-time communities aiming at cross-fertilization and multi-disciplinary approaches to the design of embedded systems.

The scope of the NG-RES workshop include the following topics:

- Programming models, paradigms and frameworks for real-time computation on parallel and heterogeneous architectures
- Networking protocols and services (e.g., clock synchronization) for distributed real-time embedded systems
- Scheduling and schedulability analysis for distributed and/or parallel real-time systems
- Application of formal methods to distributed and/or parallel real-time systems
- Compiler-assisted solutions for distributed and/or parallel real-time systems
- Middlewares for distributed and/or parallel real-time systems

In this first edition of the workshop four regular papers were accepted, each of which receiving between two and three peer reviews. In addition, we are glad to have an invited talk by Luis Almeida titled "SDN for dynamic reservations on real-time networks". We would like to thank the authors of the NG-RES 2020 papers, the members of our program committee, our invited speaker, our publisher Schloss Dagstuhl as well as the HiPEAC organizers for contributing to the success of this workshop.

Marko Bertogna and Federico Terraneo

Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020).
Editors: Marko Bertogna and Federico Terraneo

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program committee

General Chair

- Marko Bertogna, Università di Modena e Reggio Emilia, Italy

Program Chair

- Federico Terraneo, Politecnico di Milano, Italy

Web and Submission Chair

- Federico Reghenzani, Politecnico di Milano, Italy

Program committee

- Jaume Abella Ferrer, Barcelona Supercomputing Center, Spain
- Benny K. Akesson, TNO, Netherlands
- Roberto Cavicchioli, Università di Modena e Reggio Emilia, Italy
- Francisco J. Cazorla, Barcelona Supercomputing Center, Spain
- Leandro Soares Indrusiak, University of York, United Kingdom
- Alberto Leva, Politecnico di Milano, Italy
- Martina Maggio, Lund University, Sweden
- Christine Rochange, Institut de Recherche en Informatique de Toulouse, France
- Alessandro Vittorio Papadopoulos, Mälardalen University, Sweden
- Marco Solieri, Università di Modena e Reggio Emilia, Italy
- Juergen Teich, Friedrich Alexander Universität, Germany

Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020).
Editors: Marko Bertogna and Federico Terraneo

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

SDN for Dynamic Reservations on Real-Time Networks

Luis Almeida
CISTER – Research Center on Real-Time and Embedded Computing Systems and
IT – Instituto de Telecomunicações,
University of Porto - Faculty of Engineering, Porto, Portugal
https://web.fe.up.pt/~lda/

Abstract

Recent growing frameworks such as the IoT, IIot, Cloud/Fog/Edge computing, CPS, etc, bring the networking platforms on which they rely to the spotlight, as first class citizens of an increasingly software-dependent landscape. As a result, networks play an increasingly central role in supporting the needed system-wide properties. In particular, we have been working to provide openness and adaptivity together with timeliness guarantees. This combination seems fundamental to support inherently dynamic applications in a resource efficient way, covering not only the cases of systems of systems, systems with variable number of users, components or resources but also systems that undergo frequent live maintenance and even reconfiguration during their lifetime. Examples range from autonomous vehicles to collaborative robotics, remote interactions, fog/edge computing, flexible manufacturing, etc.

We postulate that combining openess and adaptivity with guaranteed timeliness can only be achieved with an adequate communication abstraction supported on adequate protocols. To this end, we have been proposing channel reservation-based communication as a means to provide scalable and open latency-constrained communication and thus enable a more efficient system design.

In this talk we will show our recent work in using Software-Defined Networking (SDN) to provide standard interfaces for traffic flexibility. We proposed extending the SDN OpenFlow protocol with adequate services to take advantage of flexible real-time communication protocols and thus provide standard interfaces for flexible real-time reservations, too. We call it the Real-Time OpenFlow framework (RTOF). We end describing and assessing a prototype implementation based on the HaRTES Ethernet switches.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Networks → Programmable networks

Keywords and phrases Latency-constrained networks, Real-time communication, Software-defined networking

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.1

Category Invited Talk

Funding This work is funded by FCT/MCTES through national funds and when applicable co-funded EU funds under the project UIDB/EEA/50008/2020.

Short bio. Luis Almeida graduated in Electronics and Telecommunications Eng. in 1988 and received a Ph.D. in Electrical Eng. in 1999, both from the University of Aveiro in Portugal. He is currently an associate professor in the Electrical and Computer Engineering Department of the University of Porto (UP), Portugal, and Vice-Director of the CISTER research unit at UP where he coordinates the Distributed and Real-Time Embedded Systems (DaRTES) lab. Among several appointments, he is Vice-Chair of the IEEE Technical Committee on Real-Time Systems (chair after 2020), Program and General Chair of the IEEE Real-Time Systems Symposium (2011-2012 respectively) and Trustee of the RoboCup Federation (2008-2016) including Vice-President (2011-2013). His research interests revolve around real-time networks for distributed industrial/embedded systems including for teams of mobile robots.

© Luis Almeida;
licensed under Creative Commons License CC-BY
Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020).
Editors: Marko Bertogna and Federico Terraneo; Article No. 1; pp. 1:1–1:1

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Energy Minimization in DAG Scheduling on MPSoCs at Run-Time: Theory and Practice

Bertrand Simon
Universität Bremen, Germany
bsimon@uni-bremen.de

Joachim Falk
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
joachim.falk@fau.de

Nicole Megow
Universität Bremen, Germany
nicole.megow@uni-bremen.de

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
juergen.teich@fau.de

Abstract

Static (offline) techniques for mapping applications given by task graphs to MPSoC systems often deliver overly pessimistic and thus suboptimal results w.r.t. exploiting time slack in order to minimize the energy consumption. This holds true in particular in case computation times of tasks may be workload-dependent and becoming known only at runtime or in case of conditionally executed tasks or scenarios. This paper studies and quantitatively evaluates different classes of algorithms for scheduling periodic applications given by task graphs (i.e., DAGs) with precedence constraints and a global deadline on homogeneous MPSoCs purely at runtime on a per-instance base. We present and analyze algorithms providing provably optimal results as well as approximation algorithms with proven guarantees on the achieved energy savings. For problem instances taken from realistic embedded system benchmarks as well as synthetic scalable problems, we provide results on the computation time and quality of each algorithm to perform a) scheduling and b) voltage/speed assignments for each task at runtime. In our portfolio, we distinguish as well continuous and discrete speed (e.g., DVFS-related) assignment problems. In summary, the presented ties between theory (algorithmic complexity and optimality) and execution time analysis deliver important insights on the practical usability of the presented algorithms for runtime optimization of task scheduling and speed assignment on MPSoCs.

2012 ACM Subject Classification Software and its engineering → Scheduling; Theory of computation → Scheduling algorithms

Keywords and phrases energy minimization, speed scaling, precedence graphs, scheduling, critical path, MPSoC

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.2

1 Introduction

Dynamic voltage and frequency scaling (DVFS) on modern processors is a mean to actively control the power and energy consumption of an MPSoC (multi-processor system-on-chip). It is used for thermal chip management in combination with dynamic power management (DPM) [5]. But it can also be used in the context of dynamic energy minimization of programs executed on the MPSoC, e.g., for real-time applications. Here, a plethora of methods has been proposed to optimize the mapping (including task assignment and scheduling) of tasks of one or multiple applications to processor cores including the selection of processor speed(s) such that, given worst-case task execution times, a global deadline is met. While first investigations only considered uni-processor systems, a great number of approaches has emerged to apply DVFS optimization algorithms offline when targeting MPSoCs [7, 15, 18].

© Bertrand Simon, Joachim Falk, Nicole Megow, and Jürgen Teich;
licensed under Creative Commons License CC-BY
Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020).
Editors: Marko Bertogna and Federico Terraneo; Article No. 2; pp. 2:1–2:13

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These approaches, however, generally suffer from assuming fixed execution times of tasks given (e.g., WCETs). However, for most applications, the execution times of tasks may depend on the workload to be processed. Or, tasks may only be conditionally executed according to control flow information [22]. Hence, a static assignment of schedule and speeds for executing tasks might not be optimal. Choudhury et al. [6] proposed a combination of offline techniques to compute worst-case and average case execution times of tasks. At run-time, a computationally inexpensive method calculates observed slack, and adapts processor speeds for energy reduction, while still guaranteeing a global deadline not to be violated. Other approaches such as [23] exploit the knowledge of special models of computation such as synchronous dataflow (SDF) to apply a mixed offline and online DVFS optimization for MPSoCs. Still, the structure of the task graph and thus periodicity of executions is assumed static. In most general applications, however, both the execution times and the task graph structure may vary over time. Here, approaches using control/dataflow graphs (CDFGs) have been proposed such as in the work of Tariq et al. [25]. However, the presented computationally complex analysis and optimization is again purely static as task execution probabilities are used and thus only the *expected* energy consumption is targeted.

On the theoretical side, Yao et al. [27] initiated the algorithmic study of speed scaling in 1995. This area received a lot of attention since then; see the surveys [2, 12]. Most of these studies focus on scheduling *independent tasks* (without precedences) and a *single processor*. Regarding the speed choice model, only few theoretical works address the *discrete speed* model which is computationally much more complex but more realistic; see, e.g. [13, 16, 17]. Most related to our investigations is the work by Aupy et al. [3] that studies the problem of minimizing the energy consumption under a given mapping of tasks to cores, and where the power consumed by a core running at speed s is equal to s^α. They consider both the continuous and the discrete speed model. Pruhs et al. [21] focus on the problem of minimizing the makespan under an energy budget in the continuous speed model with the same power law. In this framework, they designed an approximation algorithm with a polylogarithmic ratio. Bampis et al. [4] later proposed a 2-approximation for the same problem, which matches the best known algorithm for makespan minimization without energy considerations [10]. Our contributions include to rephrase these results for our framework (energy minimization with a fixed deadline) and analyze the algorithms performance experimentally. We also add new results building on this earlier work.

A major goal of this paper is to analyze whether algorithms providing provably optimal results or at least approximation bounds on the quality of the results can be implemented and practically applied in a real MPSoC system to be executed at runtime. In this regard, at least to our knowledge, the following questions have so far not satisfactorily been answered for the problem of scheduling task graphs purely at runtime based on dynamically emerging task dependence structure and worst case execution times.

- **Are there in theory sound algorithms that also can be applied in practice on an MPSoC?** E.g., depending on the absolute time scale, many real-world applications do require solutions to be computed within a time scale of 1 to 10 ms in order to be of practical use.
- **How do these execution times scale with the problem size?** Problem instances ranging in size between 10 to 500 tasks should be handled in practice within such time scales. If not possible:
- **Are there fast and scalable algorithms with provable approximation bounds on the optimality of energy consumption?**

Our main contribution is to bring together theoretical and computational results for continuous and discrete speed scaling for precedence-constrained task systems with the goal to minimize the energy consumption. We distinguish two classes of problems: The first assuming

a processor assignment and schedule of tasks on cores given, and the second computing the full schedule including the task to processor mapping as well while minimizing energy. We present algorithms building on mathematical optimization techniques such as convex and integer linear programs as well as rounding solutions of relaxations. Previously known methods are adapted to our setting and we also provide new results. For the full portfolio of considered settings (continuous/discrete speed choice, unlimited/bounded number of processors) we give both, an exact algorithm and a computationally efficient (i.e., polynomial-time) algorithm. In cases where optimal polynomial-time algorithms are ruled out under standard complexity assumptions, we give a polynomial-time algorithm with an approximation guarantee, i.e., we guarantee for any input instance that the total energy needed by our algorithm to finish all tasks by the deadline is at most a certain factor away from a minimum energy needed by any algorithm. Such theoretical approximation results give very rigorous worst-case guarantees on the solution quality under any possible input. They are of high importance particularly for safety-critical real-time applications. In our experiments on real-world instances, it is shown that the solution quality is substantially better than the ones guaranteed in our theorems for the worst case.

Moreover, we rigorously analyze the applicability of all of our algorithms on problem instances taken from realistic embedded system benchmarks as well as synthetic scalable problems. As one result, it turns out that the mathematical optimization methods are applicable for MPSoC system applications despite their complexity. Running times between 1 to 10 ms for instances up to 100 tasks are in the acceptable range for many applications. If not, also a linear-time algorithm (previously used in similar settings [3, 11, 20]) that combines optimality with scalable performance for a majority of task graph instances exhibiting a series/parallel dependence structure is presented. Overall, our results include new and old algorithms with optimality/approximation guarantees while revealing their practicability for use in MPSoCs.

2 Formal problem definition and notation

We are given a set of tasks to be executed without preemption on m cores. Precedence relations between the tasks are given as a directed acyclic graph $G = (V, E)$, where each node in the graph is associated with a task. If there is an arc in E from task j to task k then task k cannot start before task j is completed. A task $j \in V$ has a nominal execution time, or weight, $w_j \geq 0$.

For comparability of the analyzed algorithms, we assume a homogeneous multi-processor architecture in the following with uniform cores. At any time, the speed s of a core can be set to any *eligible* value between $s_{min} > 0$ and $s_{max} \geq s_{min}$, and it is part of a scheduling algorithms decision to which speed to set the processor. It depends on the particular model, which values in $[s_{min}, s_{max}]$ are eligible; we consider the *continuous model*, in which any rational value is eligible, and the *discrete model*, which allows speeds only from a given finite set of speeds. A core that is set to speed s consumes power at the rate s^α, where $\alpha \geq 1$ is a small constant. The total energy consumed is the power consumption integrated over time.

In the continuous model, we may assume that a task is executed at a uniform speed. This follows directly from the convexity of the power function [27]. For discrete speeds, we add the restriction that a task has to run at a uniform speed. This is a reasonable assumption as in many processing environments it is not possible to change the processor speed during the execution of a task. If a task j of weight w_j is executed at speed $s_j \in [s_{min}, s_{max}]$, then the time to complete is $x_j = w_j/s_j$ and the energy consumed during the computation of j is

$$E_j = x_j \cdot s_j^\alpha = w_j \cdot s_j^{\alpha-1} = w_j^\alpha / x_j^{\alpha-1}. \quad (1)$$

We consider the following problem: given a deadline $D > 0$ and a node-weighted graph $G = (V, E, w)$, schedule all tasks in graph G and decide upon the processor speeds such that all tasks finish before the deadline D and the total energy consumption is minimized. If minimizing the energy consumption is intractable, we design approximation algorithms. An algorithm is called an r-approximation if it always computes a solution finishing before the deadline, with an energy consumption being at most r times the minimal energy consumption.

In our investigations we distinguish two problem classes of different complexity:

- SpeedScaling: we are given the mapping of each task to its core and the order in which each core executes the tasks mapped to it (encoded in G). The problem is then equivalent to minimizing the *critical path* of the graph G. That is, find speeds such that the total execution time of the longest path (w.r.t. execution times x_j) is minimized.
- Speeds&Scheduling: in addition to selecting the speeds at which each task should be executed, we provide a schedule for the tasks, i.e., we determine the core and the starting time for each task.

3 Continuous speeds

We consider the setting in which each core can be set to any rational speed value in the given interval $[s_{min}, s_{max}]$.

3.1 SpeedScaling Problem

As mentioned earlier, this problem is equivalent to determining the speeds such as to minimize the *critical path* of the graph G. This problem has been studied to some extent before. We summarize relevant known algorithms and provide new ones. We present two algorithms:

1. an optimal polynomial time algorithm CVX-speed which relies on a convex programming formulation inspired by the idea of Bampis et al. [4];
2. a linear-time algorithm SPG-speed for a special graph class, namely Series-Parallel Graphs, which are very common in practice. Our algorithm is a small modification of an algorithm in [3, 11, 20] and it computes an exact optimal solution when there is no limitation in the speeds. Our experiments show that this limitation is not prohibitive in our context.

Details on the algorithms follow below. The experimental evaluation is presented in Section 5.

3.1.1 CVX-speed

We provide a convex programming formulation with linear constraints that computes the exact solution for the energy minimization problem in the SpeedScaling setting. Such programs can be solved in polynomial time up to an arbitrary precision [19] with the Ellipsoid method. The formulation is inspired by a convex program for makespan minimization by Bampis et al. [4].

Each task j is associated to a constant speed s_j. The variable x_j represents the processing time of Task j in the solution, which is equal to w_j/s_j. The variable d_j represents the completion time of task j.

$$\min \sum_{j \in V} \frac{w_j^\alpha}{x_j^{\alpha-1}} \tag{2}$$

$$\text{s.t.} \quad d_j \leq D, \qquad \forall j \in V \tag{3}$$

$$x_j \leq d_j, \qquad \forall j \in V \tag{4}$$

$$d_j + x_k \leq d_k, \qquad \forall (j,k) \in E \tag{5}$$

$$w_j/s_{max} \leq x_j \leq w_j/s_{min}, \qquad \forall j \in V. \tag{6}$$

The first three constraints ensure that tasks are executed one after the other, without preemption, respecting the precedence constraints and meeting the deadline D. Constraint 6 ensures that the speed limits are respected. Finally, the objective function computes the energy consumption for the schedule that is to be minimized. For a computed solution of the convex program, the speed s_j for task j is implied by w_j/x_j. We therefore have the following result.

▶ **Theorem 1.** CVX-SPEED *computes an optimal solution in polynomial time.*

3.1.2 SPG-speed

In the most general definition by Lawler [14], series-parallel graphs (or SP-graphs) are defined recursively as being either a single task, the series composition of two graphs (noted $(G_1; G_2)$), or the parallel composition of two graphs (noted $(G_1||G_2)$). In $(G_1; G_2)$, the tasks of G_2 cannot start before all tasks of G_1 have terminated. In $(G_1||G_2)$, there exist no precedence constraints between the tasks of G_1 and G_2.

In the context of minimizing the makespan of malleable jobs, an algorithm has been proposed and studied in [11, 20], and a similar algorithm has been used in our context in [3]. The principle of the algorithm is to define an *equivalent* task of a series and a parallel composition of two graphs. Specifically, if $\mathcal{L}_G$ represents the equivalent weight of G, we have:

- $\mathcal{L}_{T_i} = w_i$
- $\mathcal{L}_{(G_1;G_2)} = \mathcal{L}_{G_1} + \mathcal{L}_{G_2}$
- $\mathcal{L}^\alpha_{(G_1||G_2)} = \mathcal{L}^\alpha_{G_1} + \mathcal{L}^\alpha_{G_2}$

The problem of selecting the speeds for a graph G in order to minimize the energy consumption is then equivalent to the problem of selecting the speed for a unique task of weight $\mathcal{L}_G$. The minimum energy necessary to schedule a graph G under a deadline D is therefore equal to $\mathcal{L}^\alpha_G/D^{\alpha-1}$, using the speed $\mathcal{L}_G/D$, see Equation (1). In order to compute the speed at which each task has to be scheduled in such a solution, the algorithm SPG-SPEED associates a speed s to each subgraph:

- $s(G) = \mathcal{L}_G/D$
- In $(G_1; G_2)$, $s(G_1) = s(G_2) = s(G_1; G_2)$.
- In $(G_1||G_2)$, $s(G_1) = s(G_1||G_2)\mathcal{L}_{G_1}/\mathcal{L}_{(G_1||G_2)}$.

This result however requires to use speeds arbitrarily large, so the solution found may not respect the speed bounds, as specified in the following theorem.

▶ **Theorem 2** ([3, 11, 20])**.** *Given an SP-graph and ignoring the constraints s_{min} and s_{max},* SPG-SPEED *computes an optimal solution in linear time.*

3.2 Speeds&Scheduling Problem

Consider the setting in which an algorithm determines both, the speed allocation and the actual schedule including the mapping of tasks to cores. If the optimal solution requires to use the speed s_{max} for each task, then computing a schedule meeting a given deadline is already an NP-hard problem, as it is reducible to the classic $P|prec|C_{max}$ problem in the Graham three-field notation. The SPEEDS&SCHEDULING problem can therefore not have an approximation algorithm unless $P = NP$, as this includes computing a schedule meeting the given deadline. The best known scheduling algorithm for $P|prec|C_{max}$ is a 2-approximation [10], and cannot be improved under some complexity assumptions [24]. We therefore assume that the optimal solution uses speeds at most $s_{max}/2$, in order to focus on the problem of minimizing the energy and not on meeting the deadline, which is not the core of this paper. We show the following result.

▶ **Theorem 3.** APX-SCHED *is a $2^{\alpha-1}$-approximation if the optimal solution uses speeds at most $s_{max}/2$.*

The main idea of the algorithm builds on work in [4] for the related problem of minimizing the makespan under a fixed energy budget. The algorithm consists of two steps: firstly, a convex program is solved for computing the optimal speeds in a particular relaxation. Secondly, we fix these speeds and run a greedy heuristic for assigning the tasks to cores. The convex programming relaxation is as follows (recall that m is the number of cores).

$$\min \sum_{j \in V} \frac{w_j^\alpha}{x_j^{\alpha-1}} \tag{7}$$

$$\text{s.t.} \sum_{j \in V} x_j/m \leq D/2 \tag{8}$$

$$d_j \leq D/2, \quad \forall j \in V \tag{9}$$

$$x_j \leq d_j, \quad \forall j \in V \tag{10}$$

$$d_j + x_k \leq d_k, \quad \forall (j,k) \in E \tag{11}$$

$$w_j/s_{max} \leq x_j \leq w_j/s_{min}, \quad \forall j \in V. \tag{12}$$

Given an optimal solution for this program, we fix the speeds for the tasks. In the second step of the algorithm, we schedule the tasks using a list scheduling algorithm proposed by Graham [10]. That is, we consider tasks in any topological ordering (i.e., respecting the given precedence order) and assign a task to the core with currently smallest last completion time. If the makespan C obtained is smaller than D, the speeds are then lowered by a factor C/D.

Proof of Theorem 3. For a fixed speed assignment let $V := \sum_{i \in V} x_i/m$ denote the *volume* and let L denote the length of the critical path in G. Both, volume and critical path, are well known lower bounds on the makespan. Graham's list scheduling [10] yields a makespan of at most $V + L$. The convex program computes a speed assignment that minimizes the energy among all speed assignments for which both the volume and the critical path are not larger than $D/2$. Hence, Graham's list scheduling achieves a schedule where all tasks complete before $V + L \leq D$ and, thus, all tasks meet the deadline.

On the other hand, one can show that the energy consumed by this schedule is at most a factor $2^{\alpha-1}$ larger than the optimal. Indeed, consider an optimal schedule of makespan D using speeds at most $s_{max}/2$, and multiply every speed by 2. We obtain a speed assignment which is a solution to the convex program above, and whose energy cost is a factor $2^{\alpha-1}$ away from the optimal. As the speed assignment computed by the algorithm minimizes the objective function, its energy cost is not larger. ◀

In Section 5, we will show that on real-world instances, the solution quality is substantially better than the one guaranteed in Theorem 3 above. Finally, we remark that the problem is computationally highly intractable. Even for a given speed assignment, it is NP-complete to compute an optimal schedule even if all tasks have unit execution time [26] or if there are no precedence relations [9].

4 Discrete speeds

Consider the setting in which each core can run at $k \in \mathbb{N}$ possible speeds $v_1, v_2, \dots, v_k$ with $v_i < v_{i+1}$. Let the maximum ratio of speeds be $r = \max_i v_{i+1}/v_i$. Note that the mapping problem in this setting is already NP-hard even with $k = 2$ [3]. However, the more general model in which speed modifications are allowed during the execution admits a polynomial exact algorithm [3]. We also underline that the approximation ratios given in this section still hold if the optimal solution is allowed to use any rational speed in the interval $[v_1; v_k]$.

4.1 SpeedScaling problem

Assume the task-to-core assignment is given and we need to determine the speeds such as to minimize the *critical path* of the graph G. We present two algorithms: (1) an optimal exponential time algorithm ILP-D-SPEED based on an integer linear programming (ILP) formulation, (2) a polynomial time algorithm APX-D-SPEED that solves a convex program within an approximation factor $r^{\alpha-1}$.

4.1.1 ILP-D-speed

We define nk boolean variables $y_{i,\ell}$ which are equal to 1 if task i runs at speed v_ℓ and to 0 otherwise, and consider the following program similar to the convex program (2)-(6). The main difference is that the execution time of a task i is now equal to $\sum_{\ell \le k} \frac{w_i}{v_\ell} y_{i,\ell}$ and its energy consumption is equal to $\sum_{\ell \le k} w_i v_\ell^{\alpha-1} y_{i,\ell}$.

$$\text{minimize} \sum_{i \in V} w_i \sum_{\ell \le k} v_\ell^{\alpha-1} y_{i,\ell} \tag{13}$$

$$d_i \le D \qquad \forall i \in V \tag{14}$$

$$\sum_{\ell \le k} \frac{w_i}{v_\ell} y_{i,\ell} \le d_i \qquad \forall i \in V \tag{15}$$

$$d_i + \sum_{\ell \le k} \frac{w_j}{v_\ell} y_{j,\ell} \le d_j \qquad \forall (i,j) \in E \tag{16}$$

$$\sum_{\ell \le k} y_{i,\ell} = 1 \qquad \forall i \in V \tag{17}$$

$$\forall \ell \le k \;\; y_{i,\ell} \in \{0,1\} \qquad \forall i \in V. \tag{18}$$

The correctness of this ILP formulation therefore follows from the correctness of CVX-SPEED.

▶ **Theorem 4.** ILP-D-SPEED *computes an optimal solution in exponential time.*

In general, integer linear programs cannot be solved in polynomial time. However, our experiments show that on the datasets considered (up to 1000 tasks), this algorithm is at most 5 times slower than the polynomial-time algorithm CVX-SPEED.

4.1.2 APX-D-speed

The following algorithm is inspired by [3]. In a first step, we compute optimal *continuous* speeds $\bar{s}_j$ for each task j. This is done by running the fast algorithm SPG-SPEED, and, in case this algorithm does not succeed (e.g., the SP-graph restriction is not met), we solve the convex program (2)-(6) (algorithm CVX-SPEED) with $s_{min} = v_1$ and $s_{max} = v_k$. Then, the we run each task j at the speed s_j that is equal to the smallest speed v_i such that $v_i \geq \bar{s}_j$.

▶ **Theorem 5.** APX-D-SPEED *computes an $r^{\alpha-1}$-approximate solution in polynomial time.*

Proof. Consider a speed setting computed by the algorithm. Observe that the tasks respect the deadlines as the speeds s_j are not smaller than the speeds $\bar{s}_j$ that gave a valid solution. Let OPT be the energy consumed in an optimal solution. First, note that the energy consumed by executing each task at speed $\bar{s}_j$ is not larger than OPT. The algorithm runs each task j at speed s_j, consuming an energy $w_j s_j^{\alpha-1}$. The total energy consumed is then:

$$E = \sum_j w_j s_j^{\alpha-1} \leq \left(\frac{s_j}{\bar{s}_j}\right)^{\alpha-1} \sum_j w_j \bar{s}_j^{\alpha-1} \leq r^{\alpha-1} OPT.$$ ◀

4.2 Speeds&Scheduling problem

In this setting, an algorithm determines both, the speed allocation and the actual schedule including the mapping of tasks to cores. We present two algorithms: (1) an optimal exponential time algorithm ILP-D-SCHED based on solving an ILP, (2) a polynomial time algorithm APX-D-SCHED that solves a convex program within approximation factor $(2r)^{\alpha-1}$.

4.2.1 ILP-D-sched

We extend the ILP (13)-(18) by adding nm boolean variables $z_{i,c}$ equal to 1 if task i is executed on core c and to 0 otherwise, as well as n^2 variables $e_{i,j}$ indicating if task i has to be scheduled before task j. In particular, if two tasks are executed on the same core, then either $e_{i,j}$ or $e_{j,i}$ equals 1.

$$\text{minimize} \sum_{i \in V} w_i \sum_{\ell \leq k} v_\ell^{\alpha-1} y_{i,\ell} \tag{19}$$

$$d_i \leq D \qquad \forall i \in V \tag{20}$$

$$\sum_{\ell \leq k} \frac{w_i}{v_\ell} y_{i,\ell} \leq d_i \qquad \forall i \in V \tag{21}$$

$$d_i + \sum_{\ell \leq k} \frac{w_j}{v_\ell} y_{j,\ell} \leq d_j + D(1 - e_{i,j}) \qquad \forall i, j \in V \tag{22}$$

$$\sum_{\ell \leq k} v_{i,\ell} = 1 \qquad \forall i \in V \tag{23}$$

$$v_{i,\ell} \in \{0, 1\} \qquad \forall \ell \leq k,\ \forall i \in V \tag{24}$$

$$\sum_{c \leq m} z_{i,c} = 1 \qquad \forall i \in V \tag{25}$$

$$z_{i,c} \in \{0, 1\} \qquad \forall i \in V,\ \forall c \leq m \tag{26}$$

$$e_{i,j} \in \{0, 1\} \qquad \forall i, j \in V \tag{27}$$

$$e_{i,j} = 1 \qquad \forall (i, j) \in E \tag{28}$$

$$z_{i,c} + z_{j,c} - e_{i,j} - e_{j,i} \leq 1 \qquad \forall i, j \in V,\ \forall c \leq m \tag{29}$$

Equation (22) ensures that task j is executed after task i if $e_{i,j} = 1$, and does not have any impact if $e_{i,j} = 0$, so the program returns the same result as ILP-D-SPEED on a graph where the edges are represented by the variables $e_{i,j}$. The second important constraint is Equation (29), which ensures that if two tasks belong to the same core, either $e_{i,j}$ or $e_{j,i}$ equals 1. Therefore, a valid valuation of the variables $e_{i,j}$ corresponds to a directed graph which contains the edges of E, and which contains an edge between any two tasks that are placed on the same core (by the variables $z_{i,c}$). This corresponds to a valid input to the ILP-D-SPEED programming, so we have the following result.

▶ **Theorem 6.** ILP-D-SCHED *computes an optimal solution in exponential time.*

4.2.2 APX-D-sched

This algorithm combines the ideas of APX-SCHED and APX-D-SPEED: assuming again that the optimal solution uses speeds at most $v_k/2$, solve the convex program of APX-SCHED in order to associate each task to a speed $\bar{s}_j \in [v_1; v_k]$. Then, the algorithm runs each task j to the speed s_j equal to the smallest speed v_i such that $v_i \geq \bar{s}_j$, and schedules the tasks using a list scheduling algorithm.

▶ **Theorem 7.** APX-D-SCHED *computes a* $(2r)^{\alpha-1}$*-approximate solution in polynomial time if the optimal solution uses speeds at most* $v_k/2$*.*

Proof. We first note that, similarly to the APX-D-SPEED case, the energy used by the schedule obtained by APX-D-SCHED is at most a factor $r^{\alpha-1}$ away from the energy used by the APX-SCHED solution. Then, assuming that the optimal solution uses speeds at most $v_k/2$, we know that the energy used by the APX-SCHED solution is within a factor $2^{\alpha-1}$ of the optimal energy consumption. Combining these two results completes the proof. ◀

5 Experimental Results

In order to evaluate the quality of the presented approaches, we use a total of 5×5 benchmark graphs, i.e., five groups of five graphs of similar size. Our 5 smallest graphs are comprised of around 10 tasks and are derived from the Embedded System Synthesis Benchmarks Suite (E3S) [8]. These instances target processors of maximum frequency 250MHz, with a minimum frequency equal to 0.1MHz. 20 eligible speeds can be selected equally distributed between these limits. The deadlines associated to these graphs equal a few milliseconds, and are rather tight: several tasks need to be run at the maximum frequency. For larger graphs with 50, 100, 500, and 1000 tasks each, we selected graphs from the GENOME dataset of the Pegasus library [1]. The homogeneous processors used here were specified at a maximum frequency equal to 1.0GHz and again 20 equidistant speed setting, but assumed looser deadlines. All benchmarks belong to the class of SP-graphs, thus allowing the application of SPG-SPEED.

The benchmarks are executed on an Intel(R) Core(TM) i7-4770 CPU running at 3.40 GHz with 32 GiB of RAM using Ubuntu 18.04 LTS as underlying OS. To solve the ILPs for the ILP-D-SCHED and ILP-D-SPEED approaches, we use CPLEX 12.6 with a running time deadline of 5s. For the convex programs used by the CVX-SPEED and APX-D-SPEED approaches, we used MOSEK 8.1.

Figure 1 presents the results for the SPEEDSCALING problem, both in the continuous (CVX-SPEED and SPG-SPEED) and discrete speed (ILP-D-SPEED and APX-D-SPEED) settings.

Our first observation from the experiments is that the algorithm SPG-SPEED can be applied to all problem instances computing an optimal solution except for one single E3S graph instance where the prescribed speed limits were not respected. Moreover, it is really

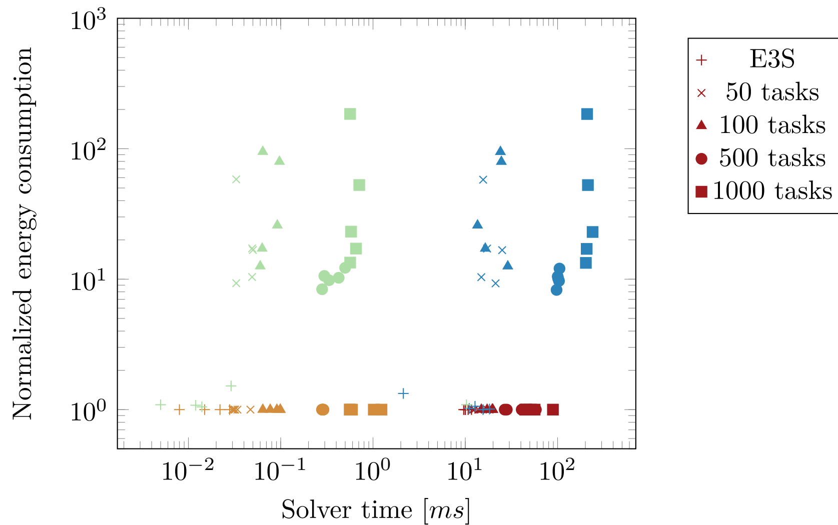


Figure 1 Depicted above are the trade-offs between solver run-time and energy used by the solution for the four approaches – CVX-SPEED, SPG-SPEED, ILP-D-SPEED, and APX-D-SPEED – that assume that mapping and scheduling is already given (only speed assignment). These trade-offs have been determined for five classes of five benchmark graphs each from the E3S benchmark suite and the Pegasus library. The energy displayed is normalized by the minimal energy consumed with continuous speeds.

fast, needing at most 0.02ms for each of the five E3S graphs and 1ms only for the largest graphs with 1000 tasks. It can therefore be applied at runtime even for problems with very small and tight deadlines. As a consequence, the algorithm APX-D-SPEED runs at a comparable speed, except for the one instance which is not solved by SPG-SPEED. Even solving optimally the convex program (CVX-SPEED) is possible in less than 10 ms for the E3S benchmarks, 15ms for 100-tasks graphs, but may be unaffordable for very large graphs (in average 60ms for 1000 tasks). When solving the ILP for discrete speeds, the solver time can even increase to 200ms for the largest graphs, but we do not observe an exponential growth for this dataset, contrarily to the worst-case theoretical complexity. Surprisingly, the quality of the solution of APX-D-SPEED is only a few percents away from the optimal discrete solution (ILP-D-SPEED). Therefore, APX-D-SPEED can obtain near-optimal results two orders of magnitude faster than by solving the ILP, on SP-graph instances. The restriction to the discrete speed model implies a higher increase in energy consumption for the GENOME dataset. This can be explained by the fact that the deadlines are looser, so the optimal continuous speeds are lower, and being forced to select a discrete speed incurs higher losses.

Figure 2 presents the results of the APX-D-SCHED algorithm that performs also task-to-core assignment and scheduling apart from speed selection. From the color code, it can be seen that the solver times are roughly equal the ones of the APX-SCHED algorithm. For each of the 25 graphs, the number of cores has been varied between 1 and 128. In each design point, the energy of the found solution has been normalized to the optimal energy for the discrete speed case with no core constraints as determined by the ILP-D-SPEED approach.

It can be seen that APX-D-SCHED is able to solve many instances of graphs with 50 to 100 tasks in less than 25ms. However, it does not find a solution for 4 out of 5 E3S graphs because of the tightness of deadlines assumed in these benchmarks and the assumptions made in Theorem 7. Finally, we omit to present and compare the solver times of ILP-D-SCHED as these start in the range of minutes even for the smallest and easiest problem instances. Hence, we conclude this approach to be of no use to be applied on an MPSoC at run-time.

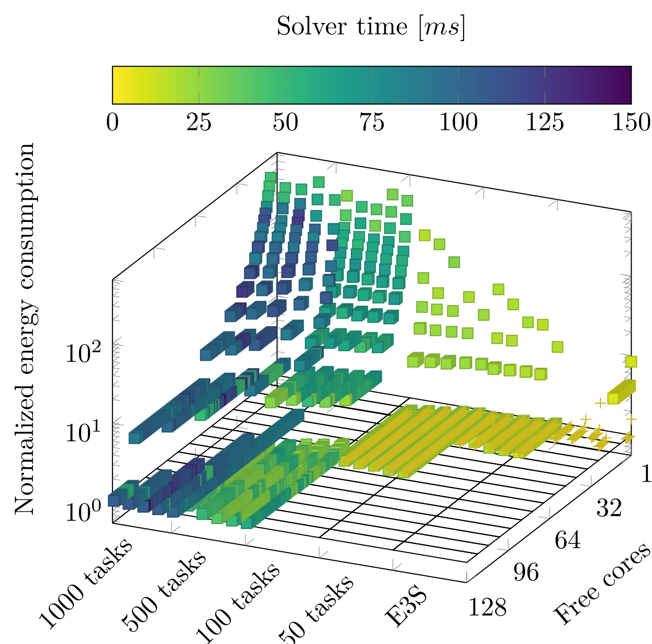


Figure 2 Consumed energy of the 5 × 5 benchmark graphs for solutions found by the APX-D-SCHED approach (squares) subject to a fixed number of available (free) cores ranging from 1 to 128. The results are normalized: a value of 1 corresponds to the case with optimum discrete speeds and infinitely many cores. The required solver time to find these solutions ranges from 7 ms (■) to 150 ms (■) according to the given color key. The crosses denote optimal-energy solutions as determined by the ILP-D-SCHED approach.

6 Conclusions

We have shown that for many task graphs of real-world applications, the graph structure allows to determine energy-optimal speed assignments in the range of a *ms* given real-time constraints by applying an algorithm called SPG-SPEED in case tasks have been mapped already to cores. For the more complex problem of additionally determining the task-to-core assignment and schedule of tasks on these cores, even problem instances with few tasks cannot be practically solved optimally at runtime. Yet here, approximation algorithms have been analyzed and shown to offer affordable solving times to determine at least solutions with provable guarantees on the solution quality.

In the future, we would like to extend our analysis of the ties between theory and practice from homogeneous MPSoCs to systems with more diverse and complex communication architectures. Moreover, the presented set of algorithms shall be integrated into a framework for run-time resource management on many-core systems that are required to stay within given bounds on execution time, energy and also other user-specific requirement corridors.

References

1 Epigenomics dataset from the Pegasus library. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`. [Online; accessed 02-September-2019].

2 Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

3 Guillaume Aupy, Anne Benoit, Fanny Dufossé, and Yves Robert. Reclaiming the energy of a schedule: models and algorithms. *Concurrency and Computation: Practice and Experience*, 25(11):1505–1523, 2013.

4. Evripidis Bampis, Dimitrios Letsios, and Giorgio Lucarelli. A note on multiprocessor speed scaling with precedence constraints. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 138–142, 2014.
5. Gang Chen, Kai Huang, and Alois Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. *ACM Trans. Embedded Comput. Syst.*, 13:111:1–111:21, 2014. doi:10.1145/2567935.
6. Pravanjan Choudhury, P. P. Chakrabarti, and Rajeev Kumar. Online Dynamic Voltage Scaling using Task Graph Mapping Analysis for Multiprocessors. In *20th International Conference on VLSI Design*, pages 89–94, 2007.
7. Pepijn J. de Langen and Ben H. H. Juurlink. Leakage-Aware Multiprocessor Scheduling. *Signal Processing Systems*, 57(1):73–88, 2009. doi:10.1007/s11265-008-0176-8.
8. R. Dick. Embedded System Synthesis Benchmarks Suite (E3S). http://ziyang.eecs.umich.edu/~dickrp/e3s/. [Online; accessed 02-September-2019].
9. M.R. Garey and D.S. Johnson. Strong NP-completeness results: motivation, examples, and implications. *J. Assoc. Comput. Mach.*, 25(3):499–508, 1978.
10. R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, November 1966. doi:10.1002/j.1538-7305.1966.tb01709.x.
11. Abdou Guermouche, Loris Marchal, Bertrand Simon, and Frédéric Vivien. Scheduling trees of malleable tasks for sparse linear algebra. In *European Conference on Parallel Processing*, pages 479–490. Springer, 2015.
12. Sandy Irani and Kirk Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, 2005.
13. Woo-Cheol Kwon and Taewhan Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Trans. Embedded Comput. Syst.*, 4(1):211–230, 2005.
14. Eugene L Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In *Annals of Discrete Mathematics*, volume 2, pages 75–90. Elsevier, 1978.
15. Keqin Li. Performance Analysis of Power-Aware Task Scheduling Algorithms on Multiprocessor Computers with Dynamic Voltage and Speed. *IEEE Trans. Parallel Distrib. Syst.*, 19(11):1484–1497, 2008. doi:10.1109/TPDS.2008.122.
16. Minming Li and F. Frances Yao. An Efficient Algorithm for Computing Optimal Discrete Voltage Schedules. *SIAM J. Comput.*, 35(3):658–671, 2005.
17. Nicole Megow and José Verschae. Dual Techniques for Scheduling on a Machine with Varying Speed. *SIAM J. Discrete Math.*, 32(3):1541–1571, 2018.
18. Andrew Nelson, Orlando Moreira, Anca Mariana Molnos, Sander Stuijk, Ba Thang Nguyen, and Kees Goossens. Power Minimisation for Real-Time Dataflow Applications. In *14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD 2011)*, pages 117–124, 2011.
19. Yurii Nesterov and Arkadii Nemirovskii. *Interior Point Polynomial Algorithms in Convex Programming.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994.
20. G. N. Srinivasa Prasanna and Bruce R. Musicus. Generalized Multiprocessor Scheduling and Applications to Matrix Computations. *IEEE TPDS*, 7(6):650–664, 1996. doi:10.1109/71.506703.
21. Kirk Pruhs, Rob van Stee, and Patchrawat Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43(1):67–80, 2008.
22. Dongkun Shin and Jihong Kim. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design, 2003, Seoul, Korea, August 25-27, 2003*, pages 408–413, 2003.
23. Amit Kumar Singh, Anup Das, and Akash Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In *The 50th Annual Design Automation Conference 2013 (DAC 2013)*, pages 115:1–115:7, 2013.
24. Ola Svensson. Hardness of Precedence Constrained Scheduling on Identical Machines. *SIAM J. Comput.*, 40(5):1258–1274, 2011.

25 Umair Ullah Tariq and Hui Wu. Energy-Aware Scheduling of Periodic Conditional Task Graphs on MPSoCs. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 13, 2017.

26 J.D. Ullman. NP-complete scheduling problems. *J. Comput. System Sci.*, 10:384–393, 1975.

27 F. Frances Yao, Alan J. Demers, and Scott Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. of the 36th Annual Symposium on Foundations of Computer Science (FOCS 1995)*, pages 374–382, 1995.

Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems

José Martins
Centro Algoritmi, Universidade do Minho, Portugal
jose.martins@dei.uminho.pt

Adriano Tavares
Centro Algoritmi, Universidade do Minho, Portugal
atavares@dei.uminho.pt

Marco Solieri
Università di Modena e Reggio Emilia, Italy
marco.solieri@unimore.it

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

Sandro Pinto
Centro Algoritmi, Universidade do Minho, Portugal
sandro.pinto@dei.uminho.pt

Abstract

Given the increasingly complex and mixed-criticality nature of modern embedded systems, virtualization emerges as a natural solution to achieve strong spatial and temporal isolation. Widely used hypervisors such as KVM and Xen were not designed having embedded constraints and requirements in mind. The static partitioning architecture pioneered by Jailhouse seems to address embedded concerns. However, Jailhouse still depends on Linux to boot and manage its VMs. In this paper, we present the Bao hypervisor, a minimal, standalone and clean-slate implementation of the static partitioning architecture for Armv8 and RISC-V platforms. Preliminary results regarding size, boot, performance, and interrupt latency, show this approach incurs only minimal virtualization overhead. Bao will soon be publicly available, in hopes of engaging both industry and academia on improving Bao's safety, security, and real-time guarantees.

2012 ACM Subject Classification Security and privacy → Virtualization and security; Software and its engineering → Real-time systems software

Keywords and phrases Virtualization, hypervisor, static partitioning, safety, security, real-time, embedded systems, Arm, RISC-V

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.3

Funding This work is supported by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 037902; Funding Reference: POCI-01-0247-FEDER-037902].
José Martins: Supported by FCT grant SFRH/BD/138660/2018.

1 Introduction

In domains such as automotive and industrial control, the number of functional requirements has been steadily increasing for the past few years [8, 42]. As the number of the resulting increasingly complex and computing power-hungry applications grows, the demand for high-performance embedded systems has followed the same trend. This has led to a paradigm shift from the use of small single-core microcontrollers running simple bare-metal applications or real-time operating systems (RTOSs), to powerful multi-core platforms, endowed with complex memory hierarchies, and capable of hosting rich, general-purpose operating systems (GPOSs).

© José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto;
licensed under Creative Commons License CC-BY
Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020).
Editors: Marko Bertogna and Federico Terraneo; Article No. 3; pp. 3:1–3:14

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

At the same time, the market pressure to minimize size, weight, power, and cost, has pushed for the consolidation of several subsystems onto the same hardware platform. Furthermore, these typically take the form of mixed-criticality systems (MCSs) by integrating components with distinct criticality levels. For example, in automotive systems, network-connected infotainment is often deployed alongside safety-critical control systems [8]. As such, great care must be taken when consolidating mixed-criticality systems to balance the conflicting requirements of isolation for security and safety, and efficient resource sharing.

Virtualization, an already well-established technology in desktop and servers, emerges as a natural solution to achieve consolidation and integration. It requires minimal engineering efforts to support legacy software while guaranteeing separation and fault containment between virtual machines (VMs). Several efforts were made to adapt server-oriented hypervisors, such as Xen [19, 47] or KVM [26, 12], to embedded architectures (mainly Arm) with considerable success. However, given the mixed-criticality nature of the target systems, the straightforward logical isolation has proven to be insufficient for the tight embedded constraints and real-time requirements [1]. Moreover, these embedded hypervisors often depend on a large GPOS (typically Linux) either to boot, manage virtual machines, or provide a myriad of services, such as device emulation or virtual networks [4, 41]. From a security and safety perspective, this dependence bloats the system trusted computing base (TCB) and intercepts the chain of trust in secure boot mechanisms, overall widening the system's attack surface [32]. More, due to the size and monolithic architecture of such OSs, this tight coupling also hampers the safety certification process of systems deploying such a hypervisor.

The static partitioning hypervisor architecture, pioneered by Siemens' Jailhouse [41], has been recently experiencing increasing adoption in MCSs from both academia and industry. This architecture leverages hardware-assisted virtualization technology to employ a minimal software layer that statically partitions all platforms resources and assigns each one exclusively to a single VM instance. It assumes no hardware resources need to be shared across guests. As each virtual core is statically pinned to a single physical CPU, there is no need for a scheduler, and no complex semantic services are provided, further decreasing size and complexity. Although possibly hampering the efficient resource usage requirement, static partitioning allows for stronger guarantees concerning isolation and real-time. Despite its design philosophy, Jailhouse falls short by still depending on Linux to boot the system and manage its "cells", suffering from the same aforementioned security ills of other hypervisors.

Despite the strong CPU and memory isolation provided by the static partitioning approach, this is still not enough as many micro-architectural resources such as last-level caches, interconnects, and memory controllers remained shared among partitions. The resulting contention leads to a lack of temporal isolation, hurting performance and determinism [3, 2]. Furthermore, this can be exploited by a malicious VM to implement DoS attacks by increasing their consumption of a shared resource [6], or to indirectly access other VM's data through the implicit timing side-channels [13]. To tackle this issue, techniques such as cache partitioning (either via locking or coloring) or memory bandwidth reservations were already proposed and implemented at both the operating system and hypervisor level [48, 27, 30, 22].

In this paper, we present Bao, a minimal, from-scratch implementation of the partitioning hypervisor architecture. Despite following the same architecture as Jailhouse, Bao does not rely on any external dependence (except the firmware to perform low-level platform management). Also, given the simplicity of the mechanism, it provides baked in support for cache coloring. Bao originally targets the Armv8 architecture, and experimental support for the RISC-V architecture is also available. As we strongly believe that security through obscurity, the approach followed by a majority of industry players, has been proven time and time again to be ineffective, Bao will be available open-source by the end of 2019.

2 Bao Hypervisor

Bao (from Mandarin Chinese "bǎohù", meaning "to protect") is a security and safety-oriented, lightweight bare-metal hypervisor. Designed for MCSs, it strongly focuses on isolation for fault-containment and real-time behavior. Its implementation comprises only a minimal, thin-layer of privileged software leveraging ISA virtualization support to implement the static partitioning hypervisor architecture (Figure 1): resources are statically partitioned and assigned at VM instantiation time; memory is statically assigned using 2-stage translation; IO is pass-through only; virtual interrupts are directly mapped to physical ones; and it implements a 1-1 mapping of virtual to physical CPUs, with no need for a scheduler. The hypervisor also provides a basic mechanism for inter-VM communication based on a static shared memory mechanism and asynchronous notifications in the form of inter-VM interrupts triggered through a hypercall. Besides standard platform management firmware, Bao has no external dependencies, such as on privileged VMs running untrustable, large monolithic GPOSs, and, as such, encompasses a much smaller TCB.

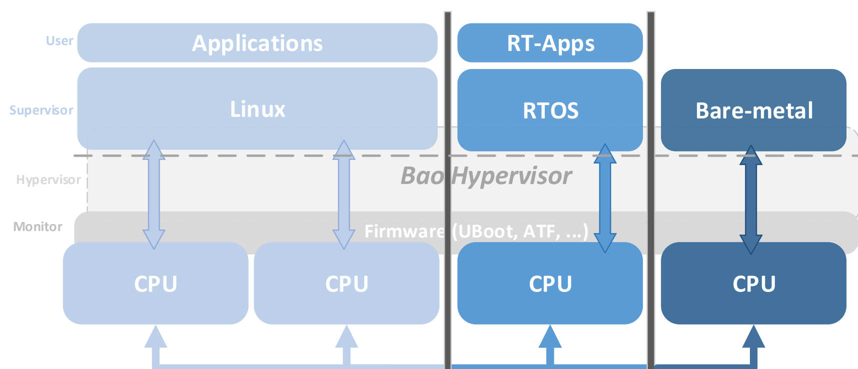


Figure 1 Bao's static partitioning architecture.

2.1 Platform Support

Bao currently supports the Armv8 architecture. RISC-V experimental support is also available but, since it depends on the hypervisor extensions, which are not yet ratified, no silicon is available that can run the hypervisor. Consequently, the RISC-V port was only deployed on the QEMU emulator, which implements the latest version of the draft specification (at the time of this writing, version 0.4). For this reason, for the remaining of the paper, we will only focus on the Arm implementation. As of the time of this writing, Bao was ported to two Armv8 platforms: Xilinx's Zynq-US+ on the ZCU102/4 development board and HiSilicon's Kirin 960 on the Hikey 960. So far, Bao was able to host several bare-metal applications, the FreeRTOS and Erikav3 RTOSs, and vanilla Linux and Android.

Except for simple serial drivers to perform basic console output, Bao has no reliance on platform-specific device drivers and requires only a minimal platform description (e.g., number of CPUs, available memory, and its location) to be ported to a new platform. For this reason, Bao relies on vendor-provided firmware and/or a generic bootloader to perform baseline hardware initialization, low-level management, and to load the hypervisor and guest images to main memory. This significantly reduces porting efforts.

On the supported Arm-based platforms, Bao relies on an implementation of the standard Power State Coordination Interface (PSCI) to perform low-level power control operations, further avoiding the need for platform-dependent drivers. On Arm-based devices, this has

been provided by Arm Trusted Firmware (ATF). On such platforms, Linux itself depends on PSCI for CPU hot-plugging. When such guests invoke PSCI services, Bao merely acts as a shim and sanitizer for the call arguments, to guarantee the VM abstraction and isolation, deferring the actual operation to ATF. Although we've been able to boot directly from ATF, we've been also using the well-known U-boot bootloader to load hypervisor and guest images.

2.2 Spatial and Temporal Isolation

Following the main requirement of isolation, Bao starts by setting up private mappings for each core. Using the recursive page table mapping technique, it avoids the need for a complete contiguous mapping of physical memory, which would otherwise be essential to perform software page table walks. This approach is usually not suitable when managing multiple address spaces and typically incurs a higher TLB footprint for page table look-ups. However, given that only a single address space is managed per CPU, and page tables are completely set-up at initialization, this is not necessarily true for our static architecture and design philosophy. Nevertheless, all cores share mappings for a per-CPU region for inter-core communication, and the hypervisor's image itself. Furthermore, only cores hosting the same VM will map its global control structure. These design decisions follow the principle of least privilege, where each core, and privilege level within it, only has (at least, direct) access to what it absolutely must. This hardens data integrity and confidentially by minimizing the available data possibly accessed by exploiting read/write gadgets available in the hypervisor. Furthermore, hypervisor code pages are marked as read-only and a X⊕W policy is enforced on hypervisor data pages by configuring them as non-executable.

Guest isolation itself starts, of course, with the logical address space isolation provided by 2-stage translation hardware virtualization support. To minimize translation overhead, page table, and TLB pressure, Bao uses superpages (in Arm terminology, blocks) whenever possible, which also possibly improves guest performance by facilitating speculative fetches. Regarding time, given exclusive CPU assignment, no scheduler is needed, which coupled with the availability of per-CPU architectural timers directly managed by the guests, allows for complete logical temporal isolation.

Despite the strong partitioning inherent to this architecture and the efforts taken to minimize the existent virtualization overheads, this is not enough to guarantee deterministic execution and meet the deadlines of critical guests' tasks. Micro-architectural contention at shared last-level caches (LLCs) and other structures still allows for interference between guest partitions. As such, given its simplicity, Bao implements a page coloring mechanism from the get-go, enabling LLC cache partitioning. Coloring, however, has several drawbacks. Firstly, it forces the use of the finest-grained page size available, precluding the benefits of using superpages. Secondly, as it also partitions the actual physical address space, leading to memory waste and fragmentation. Another problem regarding coloring is that, as Bao relies on a bootloader to load guest images, which are continuously laid out in memory, it needs to recolor them, i.e., copy the non-color compliant pages from the original loaded image to pages agreeing with the colors assigned to that specific VM, which will increase the VM's boot time. Coloring can be enabled and each color selected, independently for each VM.

2.3 IO and Interrupts

Bao directly assigns peripherals to guests in a pass-through only IO configuration. As in the supported architectures, specifically Arm, all IO is memory-mapped, this is implemented for free by using the existing memory mapping mechanisms and 2-stage translation provided by virtualization support. The hypervisor does not verify the exclusive assignment of a given peripheral, which allows for several guests to share it, albeit in a non-supervised manner.

The Generic Interrupt Controller (GIC) is the interrupt router and arbiter in the Arm architecture. Although it provides some interrupt virtualization facilities, the majority of the available hardware platforms feature either GICv2 or GICv3, which do not support direct interrupt delivery to guest partitions. All interrupts are forward to the hypervisor, which must re-inject the interrupt in the VM using a limited set of pending registers. Besides the privileged mode crossing overheads leading to an unavoidable increase in interrupt latency, this significantly increases interrupt management code complexity, especially if features such as interrupt priority are to be emulated. Bao's implementation does follow this path, as many RTOSs make use of interrupt priorities, sometimes even as a task scheduling mechanism [33, 40]. This problem was solved in the newest version of the spec, GICv4, which bypasses the hypervisor for guest interrupt delivery [12]. Furthermore, the limited virtualization support dictates that guest access to the central distributor must be achieved using trap and emulation. Depending on the frequency and access patterns of a guest to the distributor, this might significantly decrease performance. As of now, Bao only supports GICv2.

3 Evaluation

In this section, we present Bao's initial evaluation. First, we will focus on code size and memory footprint. Then we evaluate the boot time, performance, and interrupt latency. We compare guest native execution (bare) with hosted execution (solo) and hosted execution under contention (interf) to evaluate the arising interference when running multiple guests. We repeat the hosted scenarios with cache partitioning enabled (solo-col and interf-col), to understand the degree to which this first level of micro-architectural partitioning impacts the target partitions and helps to mitigate interference.

Our test platform is the Xilinx ZCU104, featuring a Zynq-US+ SoC with a quad-core Cortex-A53 running at 1.2 GHz, per-core 32K L1 data and instruction caches, and a shared unified 1MB L2/LLC cache. We execute the target test VM in one core while, when adding interference, we execute two additional bare-metal applications, each in a separate VM, which continuously write and read a 512KiB array with a stride equal to the cache line size (64 bytes). When enabling coloring, we assign half the LLC (512 KiB) to the VM running the benchmarks and one fourth (256 KiB) to each of the interfering bare-metal apps. Both the hypervisor code and benchmark applications were compiled using the Arm GNU Toolchain version 8.2.1 with -O2 optimizations.

3.1 Code Size and Memory Footprint

Bao is a complete from-scratch implementation with no external dependencies. In this section, we evaluate (i) code complexity using source lines of code (SLoC), and (ii) memory footprint by looking at the size of the final binary and then analyzing run-time consumption.

The code is divided into four main sections: the arch and platform directories contain target-specific functionality while the core and lib directories feature the main hypervisor logic and utilities (e.g., string manipulation, formatted print code), respectively. The total SLoC and final binary sizes for each directory are presented in Table 1.

Table 1 shows that, for the target platform, the implementation comprises a total of 5.6 KSLoC. This small code base reflects the overall low degree of complexity of the system. Most of the code is written in C, although functionalities such as low-level initialization and context save/restore (exception entry and exit) must be implemented in assembly. We can also see that the architecture-specific code contributes the most of the total SLoC. The largest culprit is the GIC virtualization support that amounts to almost 1/3 of the total Armv8

Table 1 Source lines of code (SLoC) and binary size (bytes) by directory.

	SLoC			size (bytes)				
	C	asm	total	.text	.data	.bss	.rodata	total
arch/armv8	2659	447	3106	22376	888	16388	482	40134
platform/xilinx/zcu104	281	0	281	464	136	0	0	600
core	1697	0	1697	14492	168	656	835	16151
lib	517	0	517	2624	0	0	24	2648
total	5154	447	5601	39956	1192	17045	1341	59535

code with about 750 SLoC. In core functionality, the code memory subsystem which includes physical page allocation and page-table management encompasses the bulk of the complexity comprising 540 SLoC. The resulting binary size is detailed in the rightmost section of Table 1. The total size of statically allocated memory is about 59 KiB. Note that the large .bss section size is mainly due to the static allocation of the root page tables. Ignoring it, this brings the total size of the final binary to be loaded to about 43 KiB.

Next, we assess the memory allocated at run-time. At boot time, each CPU allocates a private structure of 28 KiB. This structure includes the private CPU stack and page tables as well as a public page used for inter-CPU communication. For this quad-core platform, it amounts to a total of 112 KiB allocated at boot time. During initialization, Bao further allocates 4 pages (16 KiB) to use for an internal minimal allocation mechanism based on object pools. Furthermore, for each VM, the hypervisor will allocate a fixed 40 KiB for the VM global control structure plus 8 KiB for each virtual CPU. The largest memory cost for each VM will be the number of page tables which will depend first on the size of the assigned memory and memory-mapped peripherals, and second on if cache coloring is enabled or not. Table 2 shows the number of page tables used for different sizes of assigned memory. It highlights the large overhead introduced by the cache coloring mechanism on page table size. After all VMs are initialized, with the small exception of inter-CPU message allocation using the aforementioned object pools, no more memory allocation takes place.

Table 2 Page table size by VM memory size.

	no coloring		coloring	
size (MiB)	num. pages	size (KiB)	num. pages	size (KiB)
32	4	16	20	80
128	5	20	68	272
512	5	20	260	1040
1024	5	20	516	2064

3.2 Boot Overhead

In this section, we evaluate Bao's overhead on boot time (not the system's overall boot time). As such, no optimizations were carried out in any of the system's or the VMs' boot stages. In this platform, the complete boot flow includes several platform-specific boot stages: (i) a BootRom performs low-level initializations and loads the First-Stage Bootloader (FSBL) to on-chip memory, which then (ii) loads the ATF, Bao, and guest images to main memory. Next, (iii) the FSBL jumps to the ATF which then (iv) handles control to the hypervisor.

For our measurements, we use Arm's free-running architectural timer which is enabled in the early stages of ATF. Therefore, these are only approximate values to the platform's total boot time, as they do not take into account previous boot stages. We consider two cases: a

small VM (117 KiB image size and 32 MiB of memory) running FreeRTOS, and a large one (39 MiB image size and 512 MiB of memory) running Linux. For each VM, we consider the native execution (bare) scenario, and hosted execution with coloring disabled and enabled (solo and solo-col, respectively). We measure (i) hypervisor initialization as the time taken from the first instruction executed by the hypervisor to the moment it handles control to the VM, and (ii) the total boot time to the beginning of the first application inside the guest. We stress the fact that Bao does not perform guest image loading, as is the case for other embedded hypervisors. For this, it depends on a bootloader. As such, the image loading overhead is only reflected in the total time.

Table 3 Hypervisor initialization time and total VM boot time (ms).

	hyp. init. time		total boot time	
	avg	std-dev	avg	std-dev
freertos bare	n/a	n/a	2707.13	0.124
freertos solo	6.48	0.003	2720.84	0.118
freertos solo-col	9.21	0.004	2723.49	0.150
linux bare	n/a	n/a	11069.48	0.545
linux solo	9.59	0.004	11152.87	0.305
linux solo-col	155.39	1.202	11337.71	2.236

Table 3 shows the average results of 100 samples for each case. In the small VM case, the hypervisor initialization overhead is minimal (6.5 and 9.2 ms for the solo and sol-col scenarios, respectively). The total boot time increases by approximately 13 (0.5%) and 16 (0.6 %) ms, respectively, when compared with the bare scenario. In the case of the large VM running a Linux guest, Bao takes about 9.6 and a 156.2 ms to initialize itself and the VMs in the solo and solo-col case, respectively. Comparing with the native execution, the total boot time increases by about 83 (0.7 %) ms and 184 (2.4 %) ms with coloring disabled and enabled, respectively. The first point to highlight is the large increase in hypervisor initialization time with coloring enabled. This is mainly because Bao needs to color the flat image laid out by the bootloader, copying several segments of the image to color-compliant pages in the process. This is aggravated in the case of large guest images. Second, the increase in total boot time is always larger than the hypervisor initialization time. We believe this is the result of the virtualization overhead during guest initialization (e.g. 2-stage translation and GIC distributor trap and emulation).

3.3 Performance Overhead and Interference

To assess virtualization performance overhead and inter-VM interference, we employ the widely-used MiBench Embedded Benchmark Suite [14]. MiBench is a set of 35 benchmarks split into six subsets, each targeting a specific area of the embedded market: automotive (and industrial control), consumer devices, office automation, networking, security, and telecommunications. For each benchmark, MiBench provides two input data sets (small and large). We focus our evaluation on the automotive subset as this is one of the main application domains targeted by Bao. It includes three of the more memory-intensive benchmarks and therefore more susceptible to interference due to cache and memory contention [7] (qsort, susan corners, and susan edges).

Figure 2 shows the results for 1000 runs of the automotive MiBench subset. For each benchmark, we present the results as performance normalized to the bare-metal execution case, so higher values reflect poorer performance. To further investigate and understand the

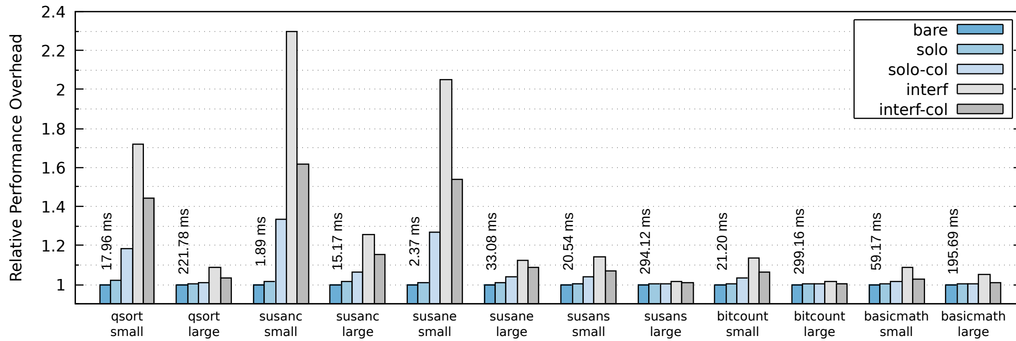


Figure 2 Performance overheads of Mibench automotive benchmark relative to bare-metal execution.

behavior of the benchmark, we collected information on L2 cache miss rate, data TLB miss rate, and stall cycle rate for memory access instructions for the qsort benchmarks. Table 4 shows the results for the small and large qsort benchmarks for each scenario.

Analyzing Figure 2, the same trend can be observed across all benchmarks to a higher or lower degree. First, observe that hosted execution causes a marginal decrease in performance. This is reflected in Table 4 by a small increase in both L2 cache and data TLB miss rates, which in turn explain the increase in memory access stall rate. As expected, this stems from the virtualization overheads of 2-stage address translation. Second, when coloring is enabled, the performance overhead is further increased. This is supported by the results in Table 4 that show an already noticeable increase across all metrics. Again, as expected, this can be explained by the fact that only half of L2 is available, and that coloring precludes the use of superpages, significantly increasing TLB pressure. In the interference scenario, there is significant performance degradation. The results in Table 4 confirm that this is due to the foreseen explosion of L2 caches misses. Finally, we can see that cache partitioning through coloring can significantly reduce interference. Table 4 shows that coloring can completely reduce L2 miss rate back to the levels of the solo colored scenario. However, looking back at Figure 2, we can see that this cutback is not mirrored in the observed performance degradation, which is still higher in the interf-col than the solo-col scenario. This can be explained by the still not address contention introduced downstream from LLC (e.g. write-back buffer, MSHRs, interconnect, memory controller) reflected in the difference in memory stall cycle rate. As expected, basicmath and bitcount were significantly less impacted by coloring and interference, given that these are much less memory-intensive.

Another visible trend in Figure 2 is that performance degradation is always more evident in the small data set variation of the benchmark. When comparing the small and large input data set variants, we see that, despite the increase in L2 cache miss rate in Table 4 being similar, the small variant experiences greater performance degradation. We believe this might be due to the fact that, given that the small input data set benchmarks has smaller total execution times, the cache miss penalty will more heavily impact them. This idea is supported by the observed memory access stall cycle rate in Table 4, which incurs in a much higher percentage increase for the small input data set case.

3.4 Interrupt Latency

To measure interrupt latency and minimize overheads unrelated to virtualization, we crafted a minimal bare-metal benchmark application. This application continuously sets up the architectural timer to trigger an interrupt each 10 ms. As the instant the interrupt is triggered

Table 4 Average L2 miss rate, data TLB miss rate and stall cycle on memory access rate for the small and large variants of MiBench's qsort benchmark.

		bare	solo	solo-col	interf	interf-col
small	L2 miss %	15.5	15.7	22.6	38.1	22.7
	DTLB miss %	0.021	0.023	0.058	0.023	0.059
	Mem. stall cyc. %	28.6	28.7	37.4	52.6	46.6
large	L2 miss %	10.1	10.1	13.4	31.7	13.4
	DTLB miss %	0.002	0.002	0.007	0.002	0.007
	Mem. stall cyc. %	4.9	5.0	5.6	8.5	7.2

is known, we calculate the latency as the difference between the expected wall-clock time and the actual instant it starts handling the interrupt. The timer has a 10 ns resolution. Results obtained from 1000 samples for each scenario are summarized in Table 5.

Table 5 Interrupt Latency (ns).

	avg	std-dev	min	max
native	140.4	11.1	140.0	490.0
solo	571.64	50.63	560.0	2170.0
solo-col	571.75	54.74	570.0	2300.0
interf	583.95	91.64	560.0	3460.0
interf-col	583.11	99.70	570.0	3620.0

When comparing native with the standalone hosted execution, we see a significant increase in both average latency and standard deviation, of approximately 430 ns and 40 ns, respectively, and of the worst-case latency by 1680 ns. This reflects the already anticipated GIC virtualization overhead due to the trap and mode crossing costs, as well as the interrupt management and re-injection. It is also visible that coloring, by itself, does not significantly impact average interrupt latency, but slightly increases the worst-case latency. The results in Table 5 also confirm the expected adverse effects of interference by cache and memory contention in interrupt latency, especially in the worst-case. Average latency grows ≈12 ns with an increase in the standard deviation of ≈41 ns and in worst-case of 1160 ns. Enabling coloring has no expressive benefits in average latency, and actually increases standard deviation and worst-case latency. We believe this was because, in this case, the relevant interference is not actually between VMs, but between the interfering guests and the hypervisor itself, which is not itself colored.

4 Related Work

Virtualization technology was introduced in the 1970's [38]. Nowadays, virtualization is a well-established technology, with a rich body of hypervisor solutions, mainly due to the large number of use cases ranging from servers, desktops, and mobiles [4, 29, 5, 44], to high- and low-end embedded systems [16, 46, 12, 28, 41, 21, 35].

Xen [4] and KVM [26] stand as the best representative open-source hypervisors for a large spectrum of applications. Xen [4] is a bare-metal (a.k.a. type-1) hypervisor that relies on a privileged VM, called Dom0, to manage non-privileged VMs (DomUs) and interface with peripherals. KVM [26] follows a different design philosophy; it was designed as a hosted hypervisor and integrated into Linux's mainline as of 2.6.20. Although initially developed

for desktop and server-oriented applications, both hypervisors have found their place into the embedded space. Xen on Arm [19] has presented the first implementation of Xen for Arm platforms and RT-Xen [47] has extended it with a real-time scheduling framework. KVM/ARM [12], in turn, has brought to light the concept of split-mode virtualization and pushed forward the hardware virtualization specification for Arm platforms.

From a different perspective, and to cope with the strict timing requirements of embedded real-time applications, a different class of systems proposes the extension of widely-used commercial RTOSes with virtualization capabilities. Green Hills INTEGRITY Multivisor, SysGo PikeOS [20], and OKL4 MicroVisor [17] are great examples of systems that take advantage of the already developed and certified RTOS infrastructure to provide the foundation to implement virtualization capabilities as services or modules atop. Also, there is another class of systems that makes use of security-oriented technologies, e.g. Arm TrustZone [37], for virtualization. TrustZone-assisted hypervisors such as SafeG [43] and LTZVisor [36] are typically dual-OS solutions which allow the consolidation of two different execution environments, i.e. an RTOS and a GPOS. In spite of both design philosophies striving for low-performance overhead and minimal interrupt latency, they typically present some limitations and fall short while supporting multiple VMs and scaling for multi-core configurations [36, 37].

Small-sized type-1 embedded hypervisors, such as Xtratum [11], XVisor [34], Hellfire/prplHypervisor [31], ACRN [23], and Minos [39] provide a good trade-off between fully-featured hypervisors and virtualization-enhanced RTOSes. Xtratum [11] was designed for safety-critical aerospace applications targeting LEON processors; nowadays, it is also available for the x86, PowerPC, and Armv7 instruction sets. Hellfire/prplHypervisor [31] was specially designed for real-time embedded systems targeting the MIPS architecture (with Virtualization Module support). XVisor [34] was designed as a tool for engaging both academia and hobbyist with embedded virtualization for Arm platforms. Intel researchers have developed ACRN [23], a lightweight hypervisor for the IoT segment and currently targeting the x86 platform. Minos [39] is an embryonic solution targeting mobile and embedded applications. Similarly to these hypervisors, Bao is also a type-1 hypervisor targeting Arm and RISC-V processors (and open to future support for MIPS or other embedded platforms); however, it distinguishes from the aforementioned solutions by following a static partition architecture which has an even reduced TCB and improved real-time guarantees.

Siemens's Jailhouse [41] pioneered the static partitioning architecture adopted by Bao. Jailhouse leverages the Linux kernel to start the system and uses a kernel module to install the hypervisor underneath the already running Linux. It then relies on this root cell to manage other VMs. Due to the proven advantages of static partitioning in embedded domains such as the automotive, other hypervisors are striving to support it. Xen has recently introduced Dom0-less execution [45], allowing DomUs to boot and execute without a Dom0, which also eliminates the Linux dependency. We strongly believe that Bao will still be able to distinguish itself from Xen Dom0-less by providing the same static partitioning benefits with a much smaller TCB and by implementing clean security features (see Section 5).

Recently, Google open-sourced Hafnium [15], a security-focused, type-1 hypevisor. It aims to provide memory isolation between a set of security domains, to better separate untrusted code from security-critical code, where each security domain is a VM.

5 On the Road

Bao's development is still at an embryonic stage. As of this writing, we are expanding support for the Arm architecture including SMMU (Arm's IOMMU) and the latest GIC versions (v3 and v4). We are also porting the system to a range of different platforms including NVIDIA's

Jetson TX2 and NXP's i.MX 8. Also, given the small size codebase, we are planning an overall refactoring to adhere to the MISRA C coding guidelines.

Bao implements cache coloring from the get-go, as a first-line of micro-architectural partitioning and isolation. We aim at implementing other state-of-the-art partitioning mechanisms (e.g. memory throttling), and color the hypervisor image itself, since we have verified that there are still contention issues between VMs or between VMs and the hypervisor. However, we believe that these issues should be supported by dedicated hardware mechanisms, to not increase code complexity and size as well as minimize overheads. Indeed, Arm has proposed the Memory System Resource Partitioning and Monitoring (MPAM) [25] extensions on Armv8.4. MPAM provides hardware support for shared cache, interconnect, and memory bandwidth partitioning. Unfortunately, no hardware featuring these extensions is available to date. We plan to implement support for MPAM using Arm Foundation Platform models, so we can test it on real hardware as soon as it is available.

Finally, since Bao is also a security-oriented hypervisor, Trusted Execution Environment (TEE) support is also on the roadmap. Typically, Arm TEEs are anchored in TrustZone technology, a set of secure hardware extensions that splits the platform into a secure and normal world [37]. TEE kernels and applications run on the secure side, while everything else (including the hypervisor) executes in the normal world. Currently, TrustZone does not support multiple isolated TEEs. Several secure world virtualization approaches have been proposed [18, 10, 24] and, recently, Arm has added secure world hypervisor support on Armv8.4. However, the dual-world approach of TrustZone-based TEEs has been shown to be fundamentally flawed [9]. Furthermore, we believe running an additional secure hypervisor would unnecessarily increase complexity, and that the secure world should only be used to encapsulate absolute security primitives (e.g. secure boot, attestation, authentication, key management). Bao's approach will take this into account, and using the already existing virtualization mechanisms, with no additional scheduling logic, will allow for multiple VMs inside a single hardware partition in the normal world. TEEs will be deployed on auxiliary VMs and only executed per request of the main guest. Another advantage of this approach is that it is portable and scalable across architectures and not specific to Arm.

6 Conclusion

In this paper, we presented the Bao hypervisor, a minimal, standalone and clean-slate implementation of the static partitioning architecture as a lightweight alternative to existing embedded hypervisors. Although development is still at an embryonic stage, preliminary evaluation shows it incurs only minimal virtualization overhead. We outline Bao's development roadmap which includes extended platform support and per-partition TEE support. Bao will be open-sourced by the end of 2019 in hopes of engaging both academia and industry in tackling the challenges of VM isolation and security.

References

1 L. Abeni and D. Faggioli. An Experimental Analysis of the Xen and KVM Latencies. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 18–26, May 2019. doi:10.1109/ISORC.2019.00014.

2 P. Axer, R. Ernst, He. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, Reinhard Von Hanxleden, R. Wilhelm, and W. Yi. Building Timing Predictable Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, March 2014. doi:10.1145/2560033.

3 A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, and M. Caccamo. Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, page 55, 2018.

4 P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. doi:10.1145/945445.945462.

5 K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, December 2010. doi:10.1145/1899928.1899945.

6 M. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, April 2019. doi:10.1109/RTAS.2019.00037.

7 A. Blin, C. Courtaud, J. Sopena, J. Lawall, and G. Muller. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 109–119, July 2016. doi:10.1109/ECRTS.2016.18.

8 P. Burgio, M. Bertogna, I. S. Olmedo, P. Gai, A. Marongiu, and M. Sojka. A Software Stack for Next-Generation Automotive Systems on Many-Core Heterogeneous Platforms. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 55–59, August 2016. doi:10.1109/DSD.2016.84.

9 D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE Symposium on Security and Privacy (S&P)*, Los Alamitos, CA, USA, 2020.

10 G. Cicero, A. Biondi, G. Buttazzo, and A. Patel. Reconciling security with virtualization: A dual-hypervisor design for ARM TrustZone. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1628–1633, February 2018. doi:10.1109/ICIT.2018.8352425.

11 A. Crespo, I. Ripoll, and M. Masmano. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *2010 European Dependable Computing Conference*, pages 67–72, April 2010. doi:10.1109/EDCC.2010.18.

12 C. Dall. *The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization*. PhD thesis, Columbia University, 2018.

13 Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 8:1–27, April 2018. doi:10.1007/s13389-016-0141-6.

14 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, December 2001. doi:10.1109/WWC.2001.990739.

15 Hafnium. Hafnium, 2019. URL: https://hafnium.googlesource.com/hafnium/.

16 G. Heiser. The Role of Virtualization in Embedded Systems. In *Workshop on Isolation and Integration in Embedded Systems*, 2008. doi:10.1145/1435458.1435461.

17 G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM. doi:10.1145/1851276.1851282.

18 Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 541–556, Vancouver, BC, August 2017. USENIX Association.

19 J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *IEEE Consumer*

Communications and Networking Conference, pages 257–261, 2008. `doi:10.1109/ccnc08.2007.64`.

20 R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

21 N. Klingensmith and S. Banerjee. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications*, HotMobile '18, pages 101–106, New York, NY, USA, 2018. ACM. `doi:10.1145/3177102.3177103`.

22 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, April 2019. `doi:10.1109/RTAS.2019.00009`.

23 H. Li, X. Xu, J. Ren, and Y. Dong. ACRN: A Big Little Hypervisor for IoT Development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 31–44, New York, NY, USA, 2019. ACM. `doi:10.1145/3313808.3313816`.

24 W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 2–16, New York, NY, USA, 2019. ACM. `doi:10.1145/3313808.3313810`.

25 Arm Ltd. Arm Architecture Reference Manual Supplement - Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A, 2018. URL: `https://developer.arm.com/docs/ddi0598/latest`.

26 U. Lublin, Y. Kamay, D. Laor, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

27 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, April 2013. `doi:10.1109/RTAS.2013.6531078`.

28 J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto. uRTZVisor: A Secure and Safe Real-Time Hypervisor. *Electronics*, 6(4), 2017. `doi:10.3390/electronics6040093`.

29 Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for High-performance Computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, April 2006. `doi:10.1145/1131322.1131328`.

30 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, February 2018. `doi:10.1109/ICIT.2018.8352429`.

31 C. Moratelli, S. Zampiva, and F. Hessel. Full-Virtualization on MIPS-based MPSOCs Embedded Platforms with Real-time Support. In *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*, SBCCI '14, pages 44:1–44:7, New York, NY, USA, 2014. ACM. `doi:10.1145/2660540.2661012`.

32 D. G. Murray, G. Milos, and S. Hand. Improving Xen Security Through Disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM. `doi:10.1145/1346256.1346278`.

33 R. Müller, D. Danner, W. S. Preikschat, and D. Lohmann. Multi Sloth: An Efficient Multi-core RTOS Using Hardware-Based Scheduling. In *26th Euromicro Conference on Real-Time Systems*, pages 189–198, July 2014. `doi:10.1109/ECRTS.2014.30`.

34 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, March 2015. `doi:10.1109/PDP.2015.108`.

35 S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on TrustZone-Enabled Microcontrollers? Voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304, April 2019. doi:10.1109/RTAS.2019.00032.

36 S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor: TrustZone is the Key. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 4:1–4:22, 2017. doi:10.4230/LIPIcs.ECRTS.2017.4.

37 S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 51(6):130:1–130:36, January 2019. doi:10.1145/3291047.

38 Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974. doi:10.1145/361011.361073.

39 Minos Project. Minos - Type 1 Hypervisor for ARMv8-A, 2019. URL: https://github.com/minos-project/minos-hypervisor.

40 E. Qaralleh, D. Lima, T. Gomes, A. Tavares, and S. Pinto. HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM multicore. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, September 2015. doi:10.1109/ETFA.2015.7301570.

41 R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits!(Almost). In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

42 A. Sadeghi, C. Wachsmann, and M. Waidner. Security and privacy challenges in industrial Internet of Things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015. doi:10.1145/2744769.2747942.

43 D. Sangorrín, S. Honda, and H. Takada. Dual Operating System Architecture for Real-Time Embedded Systems. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, pages 6–15, 2010.

44 J. Shuja, A. Gani, K. Bilal, A. Khan, S. Madani, S. Khan, and A. Zomaya. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. *ACM Computing Surveys*, 49(1):1:1–1:36, April 2016. doi:10.1145/2897164.

45 S. Stabellini. Static Partitioning Made Simple. In *Embedded Linux Conference (Noth America)*, 2019. URL: https://www.youtube.com/watch?v=UfiP9eAV0WA.

46 P. Varanasi and G. Heiser. Hardware-supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM. doi:10.1145/2103799.2103813.

47 S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 39–48, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038651.

48 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, April 2013. doi:10.1109/RTAS.2013.6531079.

A Low Energy FPGA Platform for Real-Time Event-Based Control

Silvano Seva
DEIB, Politecnico di Milano, Italy
silvano.seva@polimi.it

Claudia Esther Lukaschewsky Mauriziano
Graduate student at Politecnico di Milano, Italy
claudia.lukaschewsky@mail.polimi.it

William Fornaciari
DEIB, Politecnico di Milano, Italy
william.fornaciari@polimi.it

Alberto Leva
DEIB, Politecnico di Milano, Italy
alberto.leva@polimi.it

Abstract

We present a wireless sensor node suitable for event-based real-time control networks. The node achieves low-power operation thanks to tight clock synchronisation with the network master (at present we refer to a star network but extensions are envisaged). Also, the node does not employ any programmable device but rather an FPGA, thus being inherently immune to attacks based on code tampering. Experimental results on a simple laboratory apparatus are presented.

2012 ACM Subject Classification Hardware → Wireless integrated network sensors; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases real-time, event-based control, FPGA, wireless control networks

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.4

1 Introduction

Nowadays many control systems operate through a (partially) wireless network, and this will most likely become more and more frequent in the future. It is therefore expected that hardware and software architectures both support and foster this tendency, by allowing non-networked and/or wired existing solutions to transition toward the wireless network world in as seamless a manner as possible, and by prying the maximum advantage out of going networked, and above all wireless.

Restricting now the focus to the wireless case consistently with the scope of the paper, the important matter just mentioned has several facets, from resilience to communication deficiencies through bandwidth and energy efficiency up to security. These facets stem from two main motivations: the adoption of an inherently shared, disturbance-prone and publicly accessible medium as the radio, and the widespread use of battery-operated devices to reduce wiring as much as possible. And needless to say, the issues just sketched are to be addressed while offering real-time capabilities sharp enough for the intended application.

In this paper, which is part of a long-term research activity on networked event-based real-time control, we present a sensor node designed to operate in the context just mentioned, and we provide two main contributions. The first one is implementing a sensing device suitable for the particular event-based control technique presented in [15]. The second one is realising the said sensor using only hardware elements, without the involvement of software parts. The complete absence of any microcontroller or soft-core in the favour of a device

© Silvano Seva, Claudia Esther Lukaschewsky Mauriziano, William Fornaciari, and Alberto Leva;
licensed under Creative Commons License CC-BY

completely based on hardwired logic – besides proving that the complexity of the designed solution is adequate for an FPGA, and prospectively for an ASIC – brings about two main advantages. The first one is related to power consumption, as it is well known that an ASIC absorbs far less current[1] with respect to a microcontroller. This claim grounds on two main considerations: first of all, an ASIC-based device contains only the logic circuits necessary to perform its functionalities, while a microcontroller has different peripherals which, even when not in use, consume power. Then, coming to the technological side, in the fabrication process of an ASIC suitable techniques can be exploited to significantly reduce the power consumption, e.g. minimising the transistors' leakage current. A microcontroller, on the other hand, – being a pre-manufactured device – cannot be made as low power as needed, if not by going through a customised fabrication process.

The other advantage of using an hardwired control logic, most important for a safe operation in an IIoT context or in an industrial control network at large, is that it allows to have a strong resilience in the face of possible malicious attacks. This is because there is no way of altering the device's behaviour by reprogramming it, for the trivial reason that there is nothing to reprogram. In fact one may still think of re-configuring the FPGA, but first this is enormously more difficult than tampering with the code of a microcontroller, and then the same operation would be impossible if the design was turned into an ASIC. This is a key feature of our proposal, since in an "interconnected manufacturing" world, a damage to the communication infrastructures can lead to incidents and loss of equipment, also of huge extent.

As for the structure of the control networks which the presented device will be part of, at present we only target the star topology. This limitation is for the moment deemed acceptable, being also shared by several alternatives in the literature – like, e.g., the schemes presented in [9] and [8]. Thanks to the tight clock synchronisation technique on which our solution is based, however, we are confident that the above limitation will be released in future extensions, making it possible to realise real-time mesh wireless control networks.

The paper is organised as follows. After a brief literature review, we give a detailed description of the devised sensor node. Finally, an closed-loop experiment aimed at validating a prototype of the device is presented, alongside with a brief analysis the obtained results.

2 Related work

Most typically, digital controls are realised with periodic sampling. In recent years, however, event-based control (EBC) has emerged as a valid alternative in which control signals are instead computed "only when needed" [1, 2]. On the methodological side EBC requires a specialised theory to handle non uniform sampling [3] and guarantee stability properties [4], but its impact is evident also from the technological standpoint.

Focusing on this second aspect, in some cases EBC is viewed as a means to respond to an event immediately and not at the first sampling time following that event [10][2]. In some others it is viewed as a means for a periodic controller to skip the control signal computation

[1] It is common practice to talk about "current" instead of "power" consumption because the time integral of current, irrespective of the voltage that instead enters in the computation of power, directly provides the charge extracted from the battery, that can be easily compared against its capacity (correspondingly expressed in current-by-time units) to estimate the feasible device operation time before the battery needs replacing.

[2] Rigorously speaking this is true only for continuous-time EBC, but it can be reasonably assumed to hold also for fast-clocked event generators.

at some steps, in this case setting the clock of the event generator equal to that of the sampling – a particularly interesting feature wherever computational workloads are not negligible with respect to the control period [12]. And besides the above, the interest of EBC for real-time applications is testified by works such as [17] and many others: a survey can be found in [16].

Even more important is nowadays the conjunction of EBC and *wireless* control applications, where strict requirements in terms of energy and bandwidth consumption have to be faced, especially with battery-operated devices. The (wider) problem of energy efficiency in wireless devices for control has been addressed in works like [7, 13, 14], while specific reference to EBC is made e.g. in [20, 6, 5], and architectural aspects are investigated in works like [24, 23].

We have at this point to notice that the quoted research, when talking about "savings", tends to overlap *bandwidth* and *energy* saving [18]. This is legitimate and sensible, as less transmissions apparently achieve both objectives, but as EBC entails transmissions at *a priori* unknown instants, care has to be taken to both preserve the required timing and synchronisation properties [19] and minimise information losses due to network collisions [11].

Based on the minimum review above, we can conclude that a wireless device capable of low-power event generation with guarantees on the channel occupation instants would certainly provide a contribution to the problems above.

3 Platform description

This section describes in detail the structure of the device presented in this paper. First of all we give an overview of the assumptions we made about the network in which the node operates. Then, a functional description is given, followed by an in-depth description of some of the subsystems.

3.1 Network assumptions

The device presented in this treatise has been designed to be part of a wireless network fulfilling these requirements:

- all the network elements must consume as less power as possible,
- the transmission jitter of each data packet must be as low as possible to minimise the impact on the stability degree of the closed loop system.

Both these requirements can be accomplished using a Time Division Multiple Access (TDMA) network scheme, where each node can transmit and receive data only in prescribed time slots, assigned and known network-wide. This scheme proves to be effective for several aspects: as regards the power consumption, a TDMA approach allows each node to turn off its radio transceiver – meaning that the node cannot transmit nor receive any data packet – whenever its operation is not necessary. Each device, then, keeps its radio transceiver off except in correspondence of its time slots and only if there is data to send. The transceiver is also turned on periodically, to synchronise the device with the master node. In this latter case, energy saving is maximised when suitable synchronisation schemes are used – like the one presented in [22] – where, to synchronise, each device has to activate its radio transceiver for a very short period (in the order of the tenths of milliseconds) once every 60 seconds or so. The use of a properly synchronised TDMA scheme allows also to sensibly reduce the packet transmission jitter, both reducing the variability of the transmission period and avoiding collisions.

3.2 Functional description

The device is composed of a printed circuit board carrying three main functional elements:
- the sensing unit,
- a radio transceiver,
- and the control logic that manages all their functionalities.

In turn, the control logic is partitioned into three main functional blocks:
- one to manage the sensing unit and the event generation,
- one in charge of controlling the radio transceiver,
- and a supervising one.

The overall structure is subdivided in functional subsystems, summarised in Figure 1. The internals and the behaviour of the contained blocks are described in detail later on.

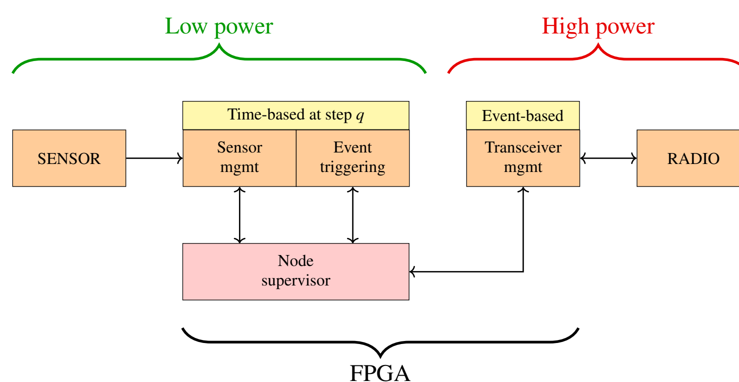


Figure 1 Organisation of the FPGA-based wireless sensor node into subsystems.

The figure evidences the presence of a “low-power” part that samples the sensor at a fixed time step and applies a digital low-pass filtering for noise mitigation, and of a “high-power’ part that is activated only on events, and takes care of transmitting data over the radio channel.

The sensing unit and the radio transceiver were sourced from the wide variety of components already available on the market off the shelf, while the control logic, due to its peculiarity, should come in the form of an ASIC, which in this work we emulate – as already said – with an FPGA. This is a component constituted by a given amount of blocks performing some basic logical operations. These blocks can be easily configured to perform more complex operations by means of tools that require an affordable effort on the part of the designer, which significantly lowers the burden of the development. An FPGA has however the disadvantage of a higher power consumption with respect to an ASIC, but this is of marginal interest when developing prototypes, as is the case here. Needless to say, therefore, we are not reporting consumption data.

The low-power part is constituted by the sensing element and the part of the ASIC designated to sampling, filtering and event generation: this part is always functioning and it has been designed in order to bring the power consumption to be as low as possible. To the high-power part belong the radio transceiver and its controlling module in the ASIC: here the main contribution to the absorbed power is given by the transceiver. Hence, in order to save power, the entire high-power section is generally completely turned off when there is no need for radio communication. As will be described more in detail later, in the case of an event, the low-power section wakes up the high-power one in order to send the measured values of the process’ output variable to the controller.

The above partitioning reflects also on the organisation of the clock signals used by the control logic to perform its various operations: all the components belonging to the low-power section are clocked with a low-frequency signal, around 32kHz, while the ones belonging to the high-power section are clocked with a signal having a frequency in the order of the MHz. This clock subdivision is strictly related as well to the need of reducing the power consumption, as with logic circuits, the absorbed power is directly proportional to the frequency of operation. Thus, all the components of the node that are going to operate continuously, are fed with a clock signal low enough to have a small power draw while preserving a good level of operational speed. On the other side, the section to which the radio transceiver belongs needs to be fed with a fast clock signal, to ensure proper operation of the transceiver itself. Nonetheless, as already mentioned, the power drawn by this part is minimised by turning off its clock signal when there is no need to use the transceiver.

All the node functionalities are coordinated at a high level by the node supervisor, also residing completely on the ASIC.

Sensor

In this treatise the sensor is, in the more general way, the component which allows to obtain a measurement of the process' output variables of interest in order to realise the control system. Without loss of generality, we assume the presence of an Analogue to Digital Converter (ADC) in the measurement chain, which allows to have a numerical representation of an analogue signal – usually in the form of a voltage – applied to its input. This approach makes our treatise applicable to a wide variety of sensors commonly used in the industrial world, since they usually give a representation of the measured quantity - temperature, pressure, flow, and so forth – in terms of a voltage measurable at the output terminals of the sensing element itself, or of a prescribed current (easily turned into voltage with a precision resistor).

Sensor manager

The sensor manager is constituted by a logic circuit contained in the ASIC and belonging to the low-power section. The purpose of this subsystem is to manage the exchange of commands and data to and from the ADC – which as said before, we consider to be our sensing unit. The sensor manager is in charge of acquiring, with a fixed and well-defined period, samples of the measured variable, in order to make them available to the other modules, namely the event trigger and the radio transceiver manager. Moreover, as already said, we assume that the sensor manager is also performing some signal conditioning after the samples are acquired in the form of a first-order low-pass digital filter.

Event trigger

The event trigger, like the sensor manager, is part of the ASIC and belongs to the low-power section. The role of this subsystem is to determine, at each step q, whether or not the measured variable has assumed a value such that there is the need to fire a wake-up event for the control system. This is done by following the Send on Delta rule: for each new filtered sample generated by the sensor manager, its difference with respect to the sample acquired in the preceding sampling step is computed. Then the absolute value said difference is compared against the event triggering threshold and, in case the this one is exceeded, an event signal is sent to the node supervisor in order to wake up the high-power part of the node.

Due to their strict interaction, the event trigger and the sensor manager are contained in the same hardware module, which is composed by an single state machine and datapath to execute both the sensor sampling and the comparison for the event triggering.

Node supervisor

The node supervisor has control over the high-level functionalities of the device, determining the proper sampling of the input signal according to the specified period and managing their sending to the master node in case an event is triggered. It also has the role of keeping the node's internal reference clock synchronised with the master node, in case a synchronised TDMA scheme is used. This clock can also be used to provide a unique timestamp for each measurement taken, to avoid ambiguities.

This subsystem has its own logic block, constituted by a state machine and a datapath, and is able to wake up the high-power section of the device when there is need to exchange date over the wireless network. Like the event trigger and sensor manager modules, this subsystem is always active during the device's operation.

Radio transceiver manager

This subsystem controls the radio transceiver, sending to it the commands required to send and receive data packets. Unlike the other modules, this part is normally turned off, meaning that no clock signal is applied to its circuitry, and is awakened only when there is the need to exchange data over the radio channel.

Radio transceiver

In our implementation the radio transceiver is an off the shelf component, containing both the RF front-end which processes the radio signal and the logic circuits necessary to perform data encoding and decoding. The chip used is a CC2520 manufactured by Texas Instruments, which provides radio communication using the IEEE 802.15.4 protocol. The management module inside the ASIC exchanges data and commands with the transceiver through an SPI interface, performing transceiver initialisation every time the transceiver is powered up, and transferring data packets to and from the transceiver's buffer.

3.3 Description of subsystems

After a general presentation of the node's internal structure was given, this section details the internal structure of both the event generator and transceiver manager subsystem. Each ot them is composed by one ore more finite-state machines with input and output signals to interact with other modules. Additionally, when the subsystem needs to manipulate some kind of data, performing logical and/or mathematical operations on it, the state machine is complemented with a data path block aimed at this objective.

Sampling and event generation module

The sampling and event generation module, as briefly described in the previous section, is in charge of acquiring samples from the ADC, filtering the obtained values, and determining if the conditions for the generation of an event are met. All these operations are performed by a single logic block composed of a data path, performing the data processing operations, and a state machine defining the execution sequence.

The ADC is interfaced with this module through an SPI interface, through which both the commands and data are exchanged and which also supplies the clock signal necessary to perform the analog-to-digital conversion.

The raw samples returned by the ADC are processed by a single-pole low-pass filter to avoid spurious fires of the event generator due to noise spikes. The filter having a discrete-time realization according to the equation

$$x(k) = (1 - \alpha) \cdot x(k - 1) + \alpha \cdot u(k), \tag{1}$$

where parameter α, $0 < \alpha < 1$, defines the time constant of the filter.

Moreover, if the value of α is restricted to be a fractional power of two, the filter can be realised using an iterative algorithm based only on two operations of the binary mathematics, namely the addition and the right shift by one position (which is equivalent to a division by two). This simplifies considerably the data path structure and the control flow.

Cascaded to the filter, the event generator block processes each filtered sample in order to determine if the conditions to fire an event are met, following the Send on Delta rule. The event generator keeps track of the value assumed by the filter's output when the last event has been generated - indicated with x_{le} hereinafter -, computes its difference Δ_x with respect to the current filtered sample x and compares the obtained value with the event triggering threshold. If the absolute value of Δ_x exceeds this threshold, an event signal is rose and the value of x_{le} is updated to x.

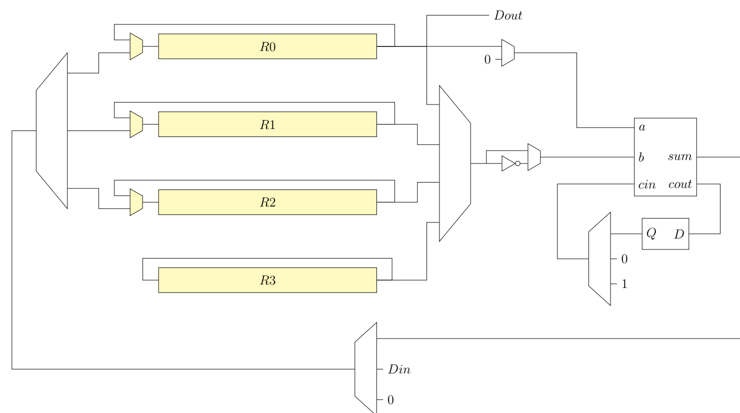


Figure 2 Datapath of the event generator.

The data path of the sampling and event generation module, shown in figure 2, consists of four 16-bit registers, an adder with carry-in and carry-out connections and a set of multiplexers and demultiplexers to manage the data flow. Each register has a particular function, as listed below.

- **R0**: general purpose register. Contains the raw data returned by the ADC immediately after the sample acquisition and the intermediate results during the filtering and event generation procedure.
- **R1**: contains the filtered value computed in the previous iteration, $x(k - 1)$.
- **R2**: contains last-event filtered value, x_{le}.
- **R3**: contains the threshold value for Δ_x.

All the values contained in the registers are binary numbers in two's complement form, where the most significant bit also carries information about the sign of the stored value

(positive if this bit is zero, negative otherwise). The data path used is 1-bit wide to keep the silicon area occupied by the module as low as possible and thus minimise the power consumption.

Transceiver manager

The transceiver management module controls all the functionalities of the node's radio transceiver. Its structure consists mainly of a single state machine exchanging both data and commands with the transceiver through a dedicated SPI bus.

The radio transceiver has two operating modes, packet transmit and packet receive, each of which is controlled by the management module using command sequences each constituted by multiple instructions needed to correctly initialise the transceiver's internal circuitry. These command sequences are permanently stored in a dedicated read-only memory: depending on the command received by the node supervisor, the management module fetches from the memory the correct one and sends it to the transceiver.

Not all the commands and status signals, however, are exchanged through the SPI bus: the *send packet* command and the *start of frame detected* signal are carried through dedicated connections between the radio transceiver's chip and the ASIC to minimise their latency. This aspect is fundamental when dealing with TDMA schemes and network synchronisation, as will be done in future developments.

4 Experimental results

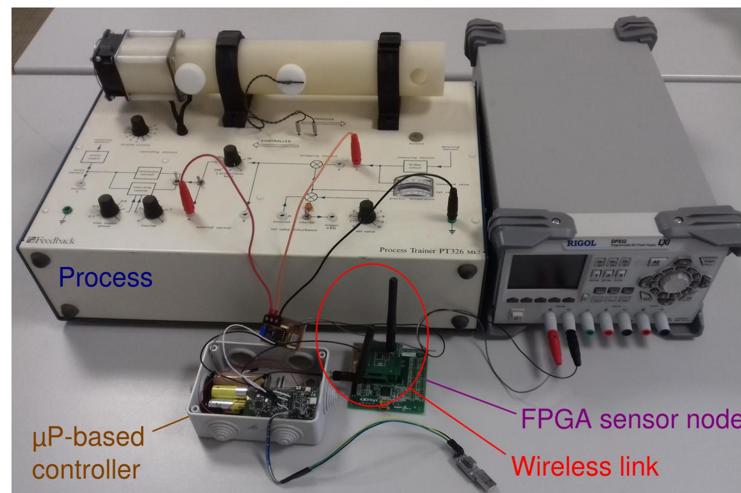


Figure 3 Experimental setup.

This section briefly presents an experiment aimed at evaluating the performance of our design. Here, the FPGA-based device has been used as a wireless sensing unit to measure the process output inside a closed-loop system. The controlled apparatus is a PT 326 thermal process trainer, manufactured by Feedback; the apparatus is composed of a plastic tube equipped with a blower and an electric heater on one of its ends and with a temperature sensor on the opposite one. The objective is to control the temperature of the airflow in the tube, as measured by the temperature sensor, acting on the heater's power. The speed of the blower can be changed too, to simulate different levels of external disturbances. Both the input and output signals of the process trainer come in the form of voltage levels, allowing for a straightforward connection to external sensing and control units.

In the setup, the FPGA-based sensor sends the measurements through a wireless connection to a *wandstem* board [21], which acts both as master node for the network and controller for the closed-loop system.

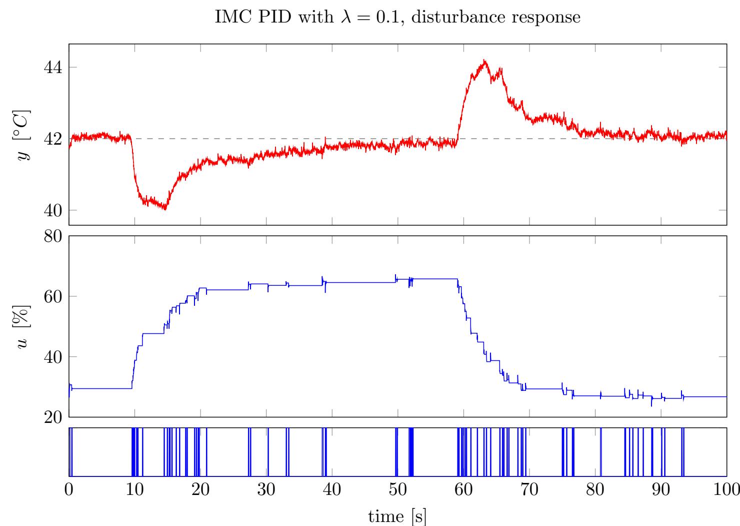


Figure 4 Experimental results.

The presented experiment is a load disturbance rejection one, where a step-shaped increase of the blower speed causes an non-measurable change in the airflow. The regulator used is an IMC-PID one whose program, written in C++ language, runs on the master node. During the experiments the process input and output and the number of samples sent have been logged. The result is shown shown in figure 4, where from top to bottom, the plots show the airflow temperature in °C, the value of the control effort and the time distribution of the packets sent over the wireless link, where a vertical bar is plotted whenever the regulator received a sample from the sensing node.

The results obtained in this and other experiments show that the realised device can be effectively used as the process' output sensor in a closed-loop system, allowing to obtain good performances while performing a quite low number of transmissions.

5 Conclusions and future work

We realised a low-power wireless sensor node suitable for real-time event-based control systems without making use of microcontrollers of soft-cores, thereby showing that an approach based completely on hardware components is feasible. This allows to significantly enhance both the energy efficiency – especially if the design is converted into an ASIC – and the resilience to malicious attacks.

As pointed out in the introduction, at present our design is effective only in a restricted context, namely where control networks have a single-hop star topology. Future research activity, then, will focus on overcoming these limitations.

Future developments will also be targeted towards enhancements in the aspects more related to event-based control. Plans are to implement a bidirectional communication between each node and the master one, for example to force the sending of measurements or to interrupt data sending triggered by events, and support for triggering rules different from the Send on Delta one will be added. We are also studying a timeout mechanism to

force the generation of an event after a given amount of time from the last one as a way to ensure proper operation of the closed-loop system. The timeout can also be exploited as a mechanism to periodically check the integrity of the communication channel.

References

1 K.E. Årzén. A simple event-based PID controller. In *Proc. 14th IFAC World Congress*, volume 18, pages 423–428, Beijing, China, 1999.

2 K.J. Åström. Event Based Control. In A. Astolfi and L. Marconi, editors, *Analysis and Design of Nonlinear Control Systems*, pages 127–147. Springer, Berlin, 2008.

3 K.J. Åström and B.M. Bernhardsson. Comparison of Riemann and Lebesgue sampling for first order stochastic systems. In *Proc. 41st IEEE Conference on Decision and Control, 2002*, volume 2, pages 2011–2016, Barcelona, Spain, 2002.

4 M. Beschi, A. Visioli, S. Dormido, and J. Sánchez. On the presence of equilibrium points in PI control systems with send-on-delta sampling. In *Proc. 50th IEEE Conference on Decision and Control and European Control Conference CDC-ECC 2011*, pages 7843–7848, Orlando, FL, USA, 2011.

5 T. Blevins, D. Chen, S. Han, M. Nixon, and W. Wojsznis. Process Control over Real-Time Wireless Sensor and Actuator Networks. In *Proc. 17th IEEE International Conference on High Performance Computing and Communication*, 2015.

6 T. Blevins, M. Nixon, and W. Wojsznis. Event based control applied to wireless throttling valves. In *Proc. 1st International Conference on Event-based Control, Communication, and Signal Processing*, Krakow, Poland, 2015.

7 N. Cardoso de Castro, D. E. Quevedo, F. Garin, and C. Canudas de Wit. Energy-aware radio chip management for wireless control. *IEEE Transactions on Control Systems Technology*, 25(6):2121–2134, 2017.

8 Y. Wei et Al. RT-wifi: Real-time high-speed communication protocol for wireless cyber-physical control applications. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 140–149, Nashville, Tennessee, 2013.

9 G. Alderisi G. Patti and L. L. Bello. Introducing multi-level communication in the IEEE 802.15.4e protocol: The MultiChannel-LLDN. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, Barcelona, Spain, 2014.

10 J.W. Grizzle, C. Chevallereau, and C.L. Shih. HZD-based control of a five-link underactuated 3D bipedal robot. In *Proc. 47th IEEE Conference on Decision and Control*, pages 5206–5213, Cancún, Mexico, 2008.

11 T. Henningsson and A. Cervin. A simple model for the interference between event-based control loops using a shared medium. In *Proc. 49th IEEE Conference on Decision and Control*, pages 3240–3245, Atlanta, GA, USA, 2010.

12 D. Henriksson and A. Cervin. Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In *Proc. 44th IEEE Conference on Decision and Control*, pages 4469–4474, Seville, Spain, 2005.

13 B. Hensel, J. Ploennigs, V. Vasyutynskyy, and K. Kabitzsch. A simple PI controller tuning rule for sensor energy efficiency with level-crossing sampling. In *Proc. 9th IEEE International Multi-Conference on Systems, Signals and Devices*, pages 1–6, Chemnitz, Germany, 2012.

14 B. Hensel, V. Vasyutynskyy, J. Ploennigs, and K. Kabitzsch. An adaptive PI controller for room temperature control with level-crossing sampling. In *Proc. 2012 UKACC International Conference on Control*, pages 197–204, Cardiff, United Kingdom, 2012.

15 A. Leva, F. Terraneo, and S. Seva. Periodic event-based control with past measurements transmission. In *2017 3rd International Conference on Event-Based Control, Communication and Signal Processing (EBCCSP)*, pages 1–8, Madeira, Portugal, 2017.

16 Q. Liu, Z. Wang, X. He, and D. Zhou. A survey of event-based strategies on control and estimation. *Systems Science & Control Engineering: An Open Access Journal*, 2(1):90–97, 2014.

17 S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada. ASEBA: A modular architecture for event-based control of complex robots. *IEEE/ASME transactions on mechatronics*, 16(2):321–329, 2010.

18 M. Rabi and K.H. Johansson. Event-triggered strategies for industrial control over wireless networks. In *Proc. 4th Annual International Conference on Wireless Internet*, pages 34:1–34:7, Brussels, Belgium, 2008.

19 J. Sijs and M. Lazar. Event Based State Estimation With Time Synchronous Updates. *IEEE Transactions on Automatic Control*, 57(10):2650–2655, 2012.

20 J. Song, A.K. Mok, D. Chen, M. Nixon, et al. Challenges of wireless control in process industry. In *Workshop on Research Directions for Security and Networking in Critical Real-Time and Embedded Systems*, 2006.

21 F. Terraneo, A. Leva, and W. Fornaciari. A High-Performance, Energy-Efficient Node for a Wide Range of WSN Applications. In *Proc. 2016 International Conference on Embedded Wireless Systems and Networks*, Graz, 2016.

22 F. Terraneo, L. Rinaldi, M. Maggio, A. V. Papadopoulos, and A. Leva. FLOPSYNC-2: Efficient monotonic clock synchronisation. In *2014 IEEE Real-Time Systems Symposium*, pages 11–20, Rome, Italy, 2014.

23 F. Tramarin, A. K. Mok, and S. Han. Real-Time and Reliable Industrial Control Over Wireless LANs: Algorithms, Protocols, and Future Directions. *Proceedings of the IEEE*, 107(6):1027–1052, 2019.

24 X.M. Zhang, Q.L. Han, and Bao-Lin B.L. Zhang. An overview and deep investigation on sampled-data-based event-triggered control and filtering for networked systems. *IEEE Transactions on Industrial Informatics*, 13(1):4–16, 2016.

Real-Time Task Migration for Dynamic Resource Management in Many-Core Systems

Behnaz Pourmohseni
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
behnaz.pourmohseni@fau.de

Fedor Smirnov
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
fedor.smirnov@fau.de

Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
stefan.wildermann@fau.de

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
juergen.teich@fau.de

Abstract

Dynamic resource management strategies in embedded many-core systems rely on task migration to adapt the deployment (mapping) of applications dynamically, e.g., for thermal/power management or load balancing. In case of hard real-time applications, however, the current practice of on-line application adaptation is limited to reconfiguring the whole application between a set of statically computed mappings with statically verified timing guarantees. This heavily restricts the application's adaptability. To enable hard real-time task migrations in many-core systems without relying on a static analysis, this paper presents (i) a *predictable task migration mechanism* supported with (ii) a lightweight *migration timing analysis* and (iii) a lightweight *migration timing feasibility check* which can be applied on-line to bound on the worst-case temporal overhead of a migration and examine the admissibility of this overhead w.r.t. the hard real-time requirements of the application. For a variety of applications and many-core platforms, we experimentally demonstrate the feasibility of hard real-time task migrations, the lightness of the proposed timing analysis and feasibility check for on-line use, and the advantage of the proposed task migration approach over mapping reconfiguration as the state-of-the-art real-time adaptation approach for many-core systems.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Multicore architectures

Keywords and phrases Hard real-time, task migration, timing analysis, dynamic resource management, multi-core, many-core

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.5

Funding This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 - TRR 89 Invasive Computing.

1 Introduction

The ever-increasing number of applications hosted on a shared multi/many-core platform in modern embedded systems engenders a highly dynamic environment: Different applications are launched and terminated on demand and independently from each other, running applications are exposed to workload variation and fluctuating performance requirements, and platform resources may become unavailable unexpectedly, e.g., due to the emergence of

© Behnaz Pourmohseni, Fedor Smirnov, Stefan Wildermann, and Jürgen Teich;
licensed under Creative Commons License CC-BY

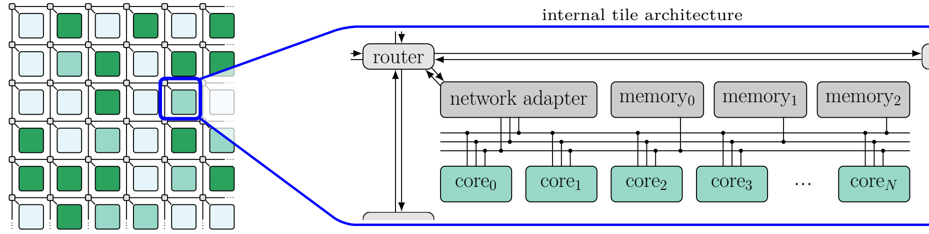


Figure 1 A heterogeneous tiled many-core architecture. Tiles are interconnected by a NoC. Each tile comprises a set of cores, a set of memories, and a network adapter, interconnected via buses.

thermal hot spots or hardware faults. Such events are typically addressed using dynamic resource management strategies which adapt the deployment of running applications. These strategies chiefly rely on *task migration* for rearranging the applications.

Migration-based resource management strategies can be viewed as an ensemble of two components: a migration policy and a migration mechanism. The *migration policy* determines *which* task(s) must be migrated *when* and *whereto*. A major factor taken into account during this selection process is the overhead associated with each migration option, e.g., the latency or the resource requirement of the migration process. These overheads are primarily a byproduct of the underlying *migration mechanism* which determines *how* a migration is performed. The choice of migration mechanism, in turn, depends on the target hardware architecture, particularly, its interconnection scheme and memory organization.

Many-core platforms, e.g. [8, 21, 36], are typically organized as a set of tiles with a Network-on-Chip (NoC) interconnection and a distributed No Remote Memory Access (NORMA) storage scheme for scalability [26], see, e.g., Fig. 1. Each tile comprises a set of cores, a set of memories, and a Network Adapter (NA), interconnected via a set of memory buses. This infrastructure enables the transmission of messages both between cores located on the same tile (*intra-tile* transmission) and between cores located on different tiles (*inter-tile* transmission). In the context of task migration, intra-tile task migrations are realized through the on-tile memories, oftentimes implicitly. The distributed memory scheme between tiles, however, necessitates inter-tile task migrations to be realized by explicit relocation of the task context between the source and destination tiles over the NoC.

Motivation. Existing works in the area of real-time task migration are either tailored to *soft* real-time constraints and try to reduce the number of deadline misses [1, 6], or assume a universal shared-memory scheme which, in the context of many-core systems, restricts their scope of applicability to intra-tile migrations only [19, 38]. Recently, *composable* many-core systems have emerged, primarily to cope with the immense systems dynamism and design complexity [2, 17, 41]. In a composable many-core system, e.g. [17], running applications are decoupled from each other using explicit reservation of resources (or resource budgets) required by each application so as to establish a spatial and/or temporal isolation between concurrent applications [2, 23]. This enables the worst-case temporal behavior of each application to be analyzed based on its reserved resources (or resource budgets), irrespective of the choice and behavior of the other applications that may run concurrently.

Contribution. In this paper, we exploit system composability to enable hard real-time task migrations without relying on a static timing analysis and verification. To that end, we present (i) a *predictable migration mechanism* which complies with the storage- and

communication schemes of many-core systems and can be employed for both intra- and inter-tile migrations, even in the case of migrations between cores of different types. We supply the proposed migration mechanism with (ii) a lightweight *migration timing analysis* which can be used on-line to calculate a safe bound on the worst-case latency of each migration process. To verify the real-time conformity of a migration, we then present (iii) a lightweight *migration timing feasibility check* which examines the admissibility of the migration latency w.r.t. the given hard real-time deadline of the application and the changes in its timing behavior during and after the migration. Our experimental results demonstrate the feasibility of hard real-time task migrations, the lightness of the proposed timing analysis and feasibility check for on-line use, and the advantage of the proposed task migration approach over the state-of-the-art hard real-time adaptation approach, namely, mapping reconfiguration.

2 Related Work

A large body of work exists on task migration in multi/many-core systems used for load balancing [4, 14, 22], temperature balancing [16, 24, 27], or fault resilience [3, 37]. They, however, either (i) rely on assumptions about the platform which do not necessarily apply to embedded many-core platforms, or (ii) disregard the temporal overhead of migration, making them inapplicable for hard real-time applications. For instance, in [1, 5, 19, 20, 30, 37], a globally shared-memory scheme is assumed for context migration while many-core systems typically manifest a distributed NORMA scheme [26]. Likewise, the migration approaches in [1, 12, 15, 27, 30] rely on a full/partial static replication of tasks on every memory in the system, which imposes an immense storage overhead that is often not tolerable in embedded many-core systems. From a predictability viewpoint, only a few existing migration approaches investigate the timing overhead of task migration [1, 6, 19, 38]. They, however, either assume soft real-time requirements and do not provide timing *guarantees* [1, 6] or investigate hard real-time task migration but rely on assumptions such as a globally shared-memory scheme which makes them inapplicable for inter-tile migrations in many-core systems [19, 38].

In the context of dynamic many-core systems, existing approaches [11, 32, 33, 40] for hard real-time application adaptation verify the admissibility of migration overhead using compute-intensive *static* timing analyses. Authors in [11] investigate real-time *system reconfigurations* between statically known system modes, each corresponding to a unique choice and deployment of active applications. Since the number of system modes and migrations per mode transition grows exponentially with the number of applications, this approach is generally not considered a viable solution for highly dynamic systems. To improve scalability, authors in [32, 33, 40] investigate per-application composable *mapping reconfigurations* in which each running application can be independently reconfigured between a set of *statically* computed mappings without affecting the other running applications.

In this paper, we present a task migration mechanism and timing analysis which, compared to mapping reconfiguration, enables a finer adaptation granularity as it empowers the real-time migration of *any subset* of an application's tasks *without* relying on a static analysis. Contrarily to existing migration solutions, our approach complies with the distributed memory scheme of embedded many-core systems. It is supported with a lightweight timing analysis and feasibility check which bound the worst-case temporal overhead of the migration processes at run time and examine the admissibility of this overhead w.r.t. the application deadline and the changes in its timing behavior *during* and *after* the migrations.

3 System Model

3.1 Platform Architecture

The target many-core platform is assumed to be organized as a set of (possibly heterogeneous) tiles interconnected by a Network-on-Chip (NoC), see, e.g., Fig. 1. Each tile comprises a set of homogeneous cores, memories, and a Network Adapter (NA), interconnected via buses.

Composability. The platform is assumed to be devoid of timing anomalies [35] and fully composable [2, 17], so that applications can share resources without affecting each other's worst-case timing behavior. Composability is established by means of exclusive reservation of resources (or reservation of periodic time budgets on resources) per application at its launch time. To establish this scheme, each potentially shared resource, i.e., core, bus, NoC link, and NA, must have a contentionless time-triggered arbitration policy, e.g., Time-Division Multiplexing (TDM) or Weighted Round-Robin (WRR). In this context, the worst-case timing behavior of each application can be analyzed based on its required resources (or resource budgets). As a result, as long as the reserved resource budgets of an application remain intact, its analyzed worst-case timing guarantees will hold, regardless of the presence and the behavior of other applications which utilize the remaining budget of these resources.

Memory Model. We consider a distributed NORMA scheme between tiles which is common for many-core systems [26]. Under this memory scheme, inter-tile data exchanges are realized by means of explicit message passing between communicating tiles over the NoC, while intra-tile data exchanges are realized through dedicated spaces in the memories on the respective tile. To achieve storage composability, the memory space in each tile is dynamically partitioned among tasks executed on it and messages produced and/or consumed on it.

NoC Model. The NoC is assumed to have a wormhole-switched- [29] and credit-based virtual-channel [7] flow control, see, e.g., the NoC in [18]. Under wormhole switching, packets are decomposed into so-called flits which are routed independently from each other in pipeline. Virtual channels provide multiple buffers per link which enables transmission preemption and composable link sharing among multiple communication flows. For each flow, the required bandwidth budget can be reserved on each link located on its transmission route, and its transfer latency can be analyzed based on its reserved budget, irrespective of the other flows.

3.2 Application and Mapping

We consider data-flow applications with a hard real-time constraint on their end-to-end latency (makespan), denoted as the application deadline. Each application is specified by an acyclic task graph (DAG) $G_P(T \cup M, E)$ where T denotes the set of tasks and M denotes the set of unicast messages, each exchanged between one pair of tasks. E is a set of directed edges which represent data dependencies among tasks and messages. For each task $t \in T$, the Worst-Case Execution Time (WCET) C_t per core type, the minimum interarrival time P_t, and the maximum context size B_t are given. For each message $m \in M$, the minimum interarrival time P_m and the maximum payload size B_m are given.

To execute an application, a so-called *mapping* of it on the platform is used which specifies (i) the binding and budget of the tasks on cores and (ii) the routing and budget of the inter-tile messages on the NoC. The Worst-Case Response Time (WCRT) L_t of each task $t \in T$ and the Worst-Case Traversal Time (WCTT) L_m of each message $m \in M$ are derived based on the budget reserved for each task (message) on its bound core (NoC route). For this purpose, we use the timing analysis from [31].

4 Real-Time Task Migration

Resource management in many-core systems, particularly, the migration of tasks, is typically controlled and operated by a so-called Run-time Manager (RM), see [39] for an overview. In the following, we consider a scenario where, during the execution of an application, task migration becomes necessary to address a run-time event, e.g., a thermal hot spot. Assume that the RM has selected a subset of the application's tasks for migration to different destinations. Before starting the migrations, the RM must first check the availability of resources required by each migrating task. These are (i) the target core, (ii) the post-migration NoC routes for inter-tile messages to/from the migrating task, and (iii) migration routes for data transfer between the source and destination tiles in case of inter-tile migrations.

In a non-real-time context, the RM performs the migrations after the availability of the required resources for all migrating tasks is verified. In a hard real-time context, however, the migrations can take place only after the RM also verifies that (iv) the timing overhead imposed during the migrations and (v) the changes in the timing behavior of the application after the migrations cannot lead to a violation of its real-time deadline. To enable this verification, in Section 4.1 we present a migration mechanism that enables the RM to migrate tasks in a predictable fashion and transparently to the application. In Section 4.2, we present a migration timing analysis which enables the RM to bound the worst-case latency of the steps involved in the migration of each task and, then, the end-to-end latency of the multi-task migration process. In Section 4.3, we present a migration timing feasibility check which enables the RM to verify the real-time conformity of the migrations w.r.t. the end-to-end migration latency and the changes in the timing behavior of the application during and after the migrations. Finally, we present an illustrative example in Section 4.4 and elaborate on the run-time overhead and complexity of our approach in Section 4.5.

4.1 Migration Mechanism

This section presents a task migration mechanism which enables the RM to perform task migrations in a predictable manner and transparently to the application. Our migration mechanism is non-preemptive. This enables migrations between heterogeneous cores using *fat binaries* without requiring source code modification and state transformation mechanisms which are typically not available in embedded systems. A fat binary comprises a set of binaries, one per Instruction Set Architecture (ISA), from which the fitting binary is selected at the migration destination, see [28]. We distinguish between intra- and inter-tile migrations:

Intra-Tile Task Migration. If a task is to be migrated between two cores on the same tile, the migration is realized implicitly via the memories on the tile. Here, the RM simply schedules the task for its next execution iteration (job) on the target core instead of the source core. The latency of this process can be safely bounded by the (known) worst-case context-switch latency L_{OS} of the operating system.

Inter-Tile Task Migration. Migrating a task between different tiles requires an explicit transfer of the task's dataset between the source and destination tiles. To that end, first the execution of the migrating task is suspended non-preemptively, i.e., after completing its current job. At the same time, its input/output (i/o) messages are suspended by blocking the injection of new messages into the NoC while allowing the already-injected messages to reach their destination node. The former ensures execution consistency between the jobs executed before the migration and the jobs executed after the migration, while the latter is crucial

to prevent communication inconsistencies that may arise, e.g., due to out-of-order delivery or even loss of input messages if they arrive at the old location *after* the migration process. Note that system services such as message forwarding or buffer reordering for resolving these issues are not typical for embedded systems. After the current job is completed and the i/o messages are suspended, the relocation process between the migration source- and destination tiles begins. In this step, the task's context, its unprocessed input messages, and its blocked output messages – all residing in the source tile's memory – are relocated to the destination tile. The task's execution is resumed after the relocation process has completed.

4.2 Migration Timing Analysis

This section presents a migration timing analysis that enables the RM to bound the worst-case end-to-end latency of migration processes. To that end, let $\hat{T} \subseteq T$ denote the set of tasks selected for inter-tile migration. Also, let function $M_{\mathrm{io}}(t)$ provide the set of input and output messages of task $t \in T$. For each task $t \in \hat{T}$, the worst-case migration latency consists of two components: (i) suspension latency $\delta_{\mathrm{susp}}(t)$ and (ii) relocation latency $\delta_{\mathrm{reloc}}(t)$. In the following, we present a lightweight timing analysis to bound the suspension- and relocation latency of each migrating task, and, subsequently, the end-to-end latency of the multi-task migration process for the two predominant cases of sequential and parallel migrations.

4.2.1 Suspension Latency

The suspension process of a migrating task $t \in \hat{T}$ – which begins after the current job of t has completed – involves two parallel operations: (i) storing the state of t in the tile memory and (ii) suspending the i/o messages of t. *State storage* is performed by the operating system. The latency of this process is bounded by the (known) worst-case context-switch latency L_{OS} of the operating system. *Communication suspension* is realized by blocking the injection of new input messages and output messages of the migrating task into the NoC and allowing the already-injected i/o messages to reach their destination. In the worst case, the suspension process is initiated right after the i/o messages are injected into in the NoC. Since each message m is guaranteed to be transmitted within its WCTT L_m, the worst-case latency for suspending all i/o messages of t can be bounded by the largest WCTT among its i/o messages. Taking into account the two parallel operations above, (i) and (ii), the worst-case suspension latency $\delta_{\mathrm{susp}}(t)$ of each migrating task $t \in \hat{T}$ can be bounded as:

$$\delta_{\mathrm{susp}}(t) = \max \left\{ L_{OS}, \max_{m \in M_{\mathrm{io}}(t)} \{L_m\} \right\} \tag{1}$$

4.2.2 Relocation Latency

The relocation of a migrating task $t \in \hat{T}$ begins only after t is suspended and involves the transfer of the *migration dataset* of t from the memory on the source tile to the destination tile. The migration dataset denotes the data required for a seamless resumption of t's execution at the destination tile. It contains t's context (code, state, etc.) of size B_t and its unprocessed input- and blocked output messages $m \in M_{\mathrm{io}}(t)$, residing in the source tile's memory. Thus, the size of the migration dataset for task $t \in \hat{T}$ is bounded by $B_{\mathrm{mig}}(t) = B_t + \sum_{m \in M_{\mathrm{io}}(t)} B_m$, where B_m denotes the maximum payload size of message m.

The migration dataset is transferred to the destination tile in three steps: (i) the NA on the source tile reads the dataset from the memory, decomposes it into flits, and injects the flits into the NoC. (ii) The flits are then transferred over the NoC to the destination tile.

Finally, (iii) the NA on the destination tile reconstructs the dataset from the flits and stores it in the memory. The worst-case relocation latency $\delta_{\text{reloc}}(t)$ of a task $t \in \hat{T}$ can be bounded using Eq. (2). Here, the first term bounds the latency of steps (i) and (iii), which we derive using the NA latency analysis from [31]. Note that the source and destination NAs have identical worst-case latencies, as they read/write the same amount of data $B_{\text{mig}}(t)$ from/to the memories. The second term in Eq. (2) bounds the NoC latency for transferring $B_{\text{mig}}(t)$ over the migration route $\rho_{\text{mig}}(t)$, which we derive using the NoC latency analysis from [33]. Both NA- and NoC analyses [31, 33] are lightweight and can be used on-line.

$$\delta_{\text{reloc}}(t) = 2 \times L_{\text{NA}}\Big(B_{\text{mig}}(t)\Big) + L_{\text{NoC}}\Big(B_{\text{mig}}(t), \rho_{\text{mig}}(t)\Big) \tag{2}$$

4.2.3 End-To-End Migration Latency

The end-to-end migration latency denotes the overall time overhead imposed on the regular execution of the application due to the migration of one or more tasks. It reflects the interval between the moment when the state storage of the first migrating task begins and the moment when the relocation processes for all migrating tasks are completed. In case of a single-task migration, the end-to-end migration latency is bounded by the sum of the suspension time $\delta_{\text{susp}}(t)$ and the relocation time $\delta_{\text{reloc}}(t)$ of that task t. If multiple tasks are to be migrated, the migrations may be performed (i) in parallel or (ii) sequentially. These two approaches enable the RM to draw a trade-off between the end-to-end migration latency and the amount of NoC budget that must be reserved for establishing the migration routes.

Parallel Migrations. In case of parallel migrations, for each migrating task t, a suspension latency $\delta_{\text{susp}}(t)$ and a relocation latency $\delta_{\text{reloc}}(t)$ is imposed. Thus, the end-to-end latency of parallel migrations can be bounded using Eq. (3). Note that parallel migrations are possible only if sufficient budget on NoC links is available so that the RM can reserve a migration route $\rho_{\text{mig}}(t)$ for each migrating task $t \in \hat{T}$. Congestion could then particularly occur when multiple migrating tasks have overlapping migration routes.

$$\delta^{\text{par}}_{\text{mig}}(\hat{T}) = \max_{t \in \hat{T}} \Big\{\delta_{\text{susp}}(t) + \delta_{\text{reloc}}(t)\Big\} \tag{3}$$

Sequential Migrations. In case of a sequential relocation of tasks, the end-to-end migration latency depends on the order in which the migrating tasks are relocated. Here, it may happen that the suspension of those tasks that are decided to be migrated first takes longer than the suspension of those that are decided to be migrated after the former. As a result, the latter suffer an idle time before the relocation of the former begins. Here, the worst-case scenario arises when (i) the task $t' \in \hat{T}$ chosen to be migrated first is the one with the highest WCRT, i.e., $L_{t'} = \max_{t \in \hat{T}}\{L_t\}$, (ii) the suspension request is issued right after t' starts its execution iteration, and (iii) at least one other migrating task $\tilde{t} \in \hat{T}$ has finished its execution iteration and updated its state in the memory prior to the suspension request. In this situation, $\tilde{t}$ undergoes the highest possible idle time before the relocation of the first migrating task t' begins. This idle time is guaranteed not to exceed the sum of t''s WCRT $L_{t'}$ and worst-case suspension time $\delta_{\text{susp}}(t')$. Thus, the worst-case migration latency for a sequential relocation of migrating tasks can be bounded as:

$$\delta^{\text{seq}}_{\text{mig}}(\hat{T}) = \max_{t \in \hat{T}} \Big\{L_t + \delta_{\text{susp}}(t)\Big\} + \sum_{t \in \hat{T}} \delta_{\text{reloc}}(t) \tag{4}$$

4.3 Migration Timing Feasibility Check

Task migration affects the temporal behavior of the application twofold: First, the regular execution of migrating tasks and the injection of their i/o messages into the NoC are suspended during the migration process. Second, the WCRT of each migrating task and the WCTT of its i/o messages may change after the migration; The WCRT of a task may change, e.g, if its pre- and post-migration cores are heterogeneous. The WCTT of a message may change, e.g., if its pre- and post-migration NoC routes have different lengths.

We present a lightweight migration timing feasibility check to enable the RM to examine whether performing a given set of migrations can lead to the violation of the application's deadline, taking into account the worst-case migration latency as calculated in Section 4.2 and the changing timing behavior of the application during and after the migrations. The migration timing feasibility check must verify that the application deadline will be respected by the end-to-end latency of each application input which either (i) arrives before the migrations and is processed by some migrating tasks before their migration and by some others after their migration or (ii) arrives during/after the migrations and is, therefore, processed by the migrating tasks after their migration. We examine the satisfaction of the given application deadline for both of these cases simultaneously by calculating a safe upper bound on the end-to-end latency of any application input as follows:

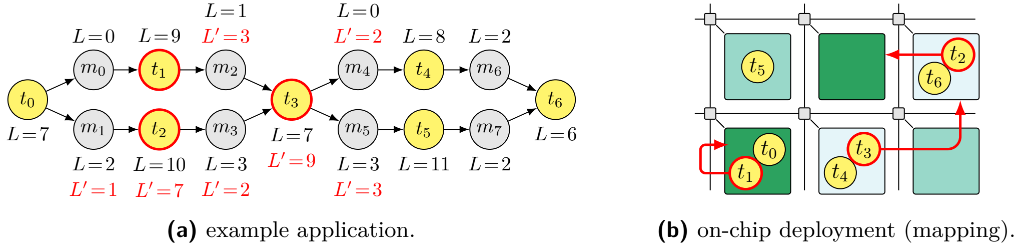
- **i** For each migrating task $t \in \hat{T}$, the post-migration WCRT L'_t is calculated using the response time analysis from [31]. For all other tasks $t \in T \backslash \hat{T}$, we consider $L'_t = L_t$.
- **ii** For each message $m \in M$ to/from the migrating tasks, the post-migration WCTT L'_m is calculated using the traversal time analysis from [31]. For other messages, $L'_m = L_m$.
- **iii** For each application task/message $x \in T \cup M$, a safe bound on x's pre- and post-migration latency is derived as $\hat{L}_x = \max\{L_x, L'_x\}$, referred to as the *compound latency* of x.
- **iv** The latency of the longest path in the application DAG is derived using the DFS algorithm [13] where the compound latency of each task/message is used as its weight. The result is referred to as the *compound application latency* and denoted by $\hat{L}_{\text{app}}$.

The compound application latency $\hat{L}_{\text{app}}$ provides a safe bound on the end-to-end latency of any application input whose processing may be affected by the migrations in question. Therefore, the RM can check the real-time conformity of the migrations by verifying that $\hat{L}_{\text{app}} + \delta_{\text{mig}}(\hat{T})$ does not exceed the given application deadline. Here, $\hat{L}_{\text{app}}$ bounds the end-to-end latency of the application and $\delta_{\text{mig}}(\hat{T})$ (derived in Section 4.2) bounds the end-to-end latency of the migrations.

4.4 Example

Consider the exemplary application depicted in Fig. 2a which is mapped on four tiles of a many-core architecture as shown in Fig. 2b. The application tasks t_0–t_6 communicate with each other via messages m_0–m_7. For brevity, the NoC routes of messages and the internal layout of tiles (including the binding of tasks to cores, the memories, and the NAs) are not depicted in Fig. 2b. The WCRT L_t of each task t and the WCTT L_m of each message m are also given in Fig. 2a. Assume a scenario where the RM has selected tasks t_1–t_3 for migration to the destinations indicated by red arrows in Fig. 2b. Task t_1 is selected for intra-tile migration, whereas tasks t_2 and t_3 are selected for inter-tile migration, thus, $\hat{T} = \{t_2, t_3\}$.

To check whether the migration of t_2 and t_3 can lead to the violation of the application's deadline, the RM first calculates the end-to-end latency of the migrations. Assuming a context-switch latency of $L_{OS} = 1$, Eq. (1) bounds the suspension latency of the migrating tasks as $\delta_{\text{susp}}(t_2) = \max\{1, \max\{2, 3\}\} = 3$ and $\delta_{\text{susp}}(t_3) = \max\{1, \max\{1, 3, 0, 3\}\} = 3$.



(a) example application. **(b)** on-chip deployment (mapping).

Figure 2 (a) Example application annotated with pre-/post-migration latencies of tasks and messages and (b) its pre-migration mapping on the chip, used in the illustrative example in Section 4.4.

Then, assuming relocation latencies of $\delta_{\text{reloc}}(t_2) = 4$ and $\delta_{\text{reloc}}(t_3) = 6$, the end-to-end migration latency of t_2 and t_3 is guaranteed not to exceed $\delta^{\text{par}}_{\text{mig}}(\hat{T}) = \max\{(3+4),(3+6)\} = 9$ in case of parallel migrations, or $\delta^{\text{seq}}_{\text{mig}}(\hat{T}) = \max\{(10+3),(7+3)\} + (4+6) = 23$ in case of sequential migrations, derived using Eq. (3) and Eq. (4), respectively.

For migration timing feasibility check, assume that the RM has derived – using the analysis from [31] – the post-migration WCRT L'_t of each migrating task $t \in \hat{T}$ and the post-migration WCTT L'_m of t's i/o messages $m \in M_{\text{io}}(t)$ as given in Fig. 2a. Based on these, the compound application latency is bounded to $\hat{L}_{\text{app}} = 53$, following steps (i)–(iv) in Section 4.3. Recall that $\hat{L}_{\text{app}} = 53$ bounds the end-to-end latency of application inputs that are affected by the migration process. In our example, this is the latency for an input that passes through t_0, m_1, t_2, and m_3 before the migrations, is blocked at the input buffer of t_3 prior to the migration process, is relocated with t_3 during the migrations, and passes through t_3, m_5, t_5, m_7, and t_6 after the migrations. Based on the latency bounds above, the RM performs the migrations only if the application deadline is at least $\delta^{\text{par}}_{\text{mig}}(\hat{T}) + \hat{L}_{\text{app}} = 9 + 53 = 62$ in case of parallel migrations, or $\delta^{\text{seq}}_{\text{mig}}(\hat{T}) + \hat{L}_{\text{app}} = 23 + 53 = 76$ in case of sequential migrations.

4.5 Run-Time Overhead and Complexity

Any analysis targeted for on-line use must be lightweight so as to introduce an acceptable overhead for the RM. In the following, we elaborate on the computational complexity of the proposed migration timing analysis and feasibility check. Note that the WCRT and WCTT analyses adopted from [31], and the NA- and NoC latency analyses adopted from [31] and [33], respectively, are constant-time non-iterative operations with a complexity of $O(1)$.

The migration timing analysis presented in Section 4.2 embodies a 2-level nested loop where the outer loop iterates through migrating tasks and the inner loop iterates through their i/o messages. Since each message is unicast (has one producer and one consumer, see Section 3.2), the inner loop can have a maximum total of $2\,|M|$ iterations, resulting in a linear time complexity of $O(|T| + 2\,|M|) = O(|T| + |M|)$ for the migration timing analysis.

For the migration timing feasibility check, the main compute overhead stems from the calculation of the compound application latency in steps (i)–(iv) in Section 4.3. Here, steps (i)–(iii) are implemented by simple loops with a computational complexity of $O(|T|)$, $O(|M|)$, and $O(|T| + |M|)$, respectively. Having the application DAG provided as adjacency lists, the DFS algorithm in step (iv) will have a complexity of $O(|T| + |M|)$. Therefore, the migration timing feasibility check presented in Section 4.3 has a linear time complexity of $O(|T| + |M|)$. When examining the real-time conformity of a (possibly multi-task) migration, the RM applies the migration timing analysis and the feasibility check in succession. This introduces a compute overhead of linear time complexity $O(|T| + |M|)$ for the RM, rendering the proposed migration timing analysis and feasibility check scalable for on-line use.

5 Experimental Results

For our experiments, we consider two heterogeneous tiled many-core architectures with 6×6 and 8×8 tiles, respectively. Each tile is composed of four homogeneous cores while each platform comprises tiles of three different core types. Every shared resource (core, bus, NA, and NoC link) has a WRR arbitration policy. For the NoC, the XY-routing algorithm [29] is used. We consider four hard real-time applications from areas of automotive (18 tasks, 21 messages), telecommunication (14 tasks, 20 messages), consumer (11 tasks, 12 messages), and networking (7 tasks, 9 messages) provided by the Embedded System Synthesis Benchmarks Suite (E3S) [10]. To obtain a set of mappings for each application per architecture, we use the OpenDSE framework [34] to perform a Design Space Exploration (DSE), employing the NSGA-II evolutionary algorithm [9] provided by the Opt4J optimization framework [25]. The DSE is performed over $1,000$ generations and retains a population of 100 mappings. It optimizes the mappings w.r.t. five design objectives to be minimized: (i) distance to the hard real-time application deadline (set to 80% of the aggregate interarriavl time of tasks and messages on the longest path) evaluated using the analysis from [31], (ii) energy consumption evaluated based on [10] for cores and [42] for buses/NoC links with wire lengths of 5 mm and 2 mm, respectively, and (iii)–(v) number of allocated cores from each of the three core types. The DSE provides a set of Pareto-optimal mappings V_i per application i.

In our experiments, we investigate the feasibility and the effectiveness of the proposed real-time task migration approach in a case study on adaptive thermal management of many-core systems. Consider the scenario in which a real-time application i is launched using one of its precomputed mappings $v \in V_i$. During the execution of the application, the RM identifies the emergence of a thermal hot spot around one of the cores in use by the application which, consequently, necessitates the evacuation of the thermally affected core while guaranteeing that the evacuation process will not lead to the violation of the application's deadline. For the evacuation, we consider two adaptation approaches:

(i) *Mapping Reconfiguration.* In this approach, the RM reconfigures the application to another one of its precomputed mappings which does not depend on the thermally affected core. To that end, the RM iterates through the mappings $v' \in V_i \setminus \{v\}$ and checks per mapping (i) the availability of its required cores and NoC routes, (ii) the availability of migration routes for the relocation of (potentially all) tasks, and (iii) the real-time conformity of the reconfiguration process. We implement this approach using the mapping reconfiguration mechanism and timing analysis from [33] which are developed based on a sequential migration of tasks. This approach represents the state of the art in hard real-time application adaptation. Here, the evacuation of the thermally affected core is considered successful iff a mapping is found which passes both the resource checks, (i) and (ii), and the timing check, (iii).

(ii) *Task Migration.* In this approach, the RM migrates only those tasks that are running on the thermally affected core. We implement this approach using the proposed migration mechanism, supported by our migration timing analysis and timing feasibility check for the worst-case timing verification of the migrations. For the sake of comparability with mapping reconfiguration, the migrations are performed sequentially. For a migration-based evacuation, the RM iterates through the platform tiles (excluding the heated tile) and checks for each candidate tile, (i) the availability of a free core, (ii) the availability of NoC routes for i/o messages of the migrating tasks after the migration, and (iii) the availability of a NoC route for the relocation of migrating tasks. If the availability of all required resources is verified,

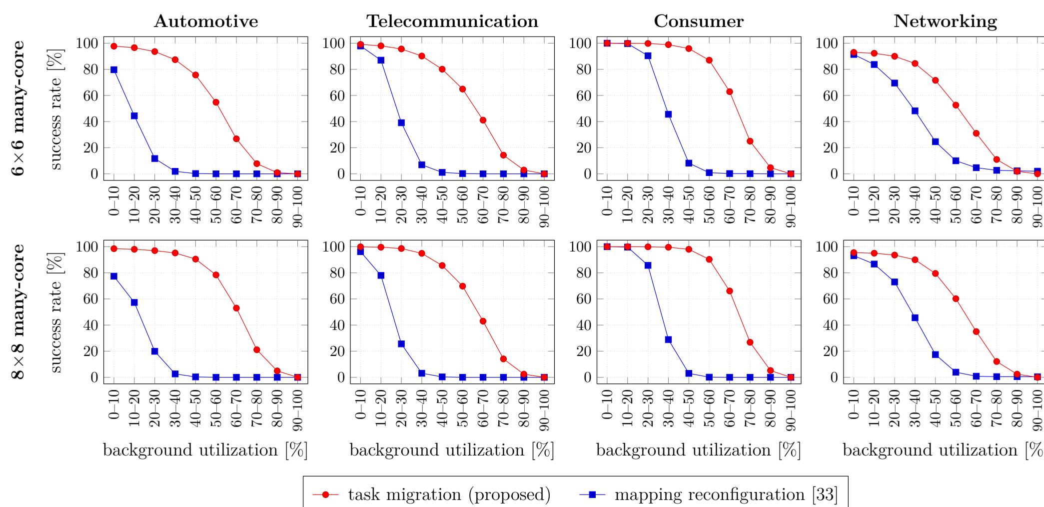


Figure 3 Success rate of task migration and mapping reconfiguration at different background utilization levels. The plots in each column (or row) correspond to one application (or architecture).

the RM performs (iv) the migration timing analysis and feasibility check. If the required resources are not available or the timing check is not passed, the RM continues its search through the remaining tiles. The evacuation is considered successful iff a destination tile is found with passes both the resource checks, (i)–(iii), and the timing check, (iv).

We perform the evacuation experiment for each mapping $v \in V_i$ of each application i as follows: First, application i is launched on an empty platform using mapping v. Then, we introduce additional (background) load into the system by iteratively occupying free resources (cores and NoC links) at random, thereby, generating different *background utilization levels*. At each utilization level, we then iterate through the cores in use by the application and, in each iteration, mark one core as an emerging hot spot so that its evacuation becomes necessary in near future. Then, for each investigated approach, i.e., mapping reconfiguration and task migration, we check whether the affected core can be evacuated successfully.

Evacuation Success. For each background utilization level, we record the evacuation success of each approach. Figure 3 illustrates the success rate of the two approaches versus background utilization level per application (plot column) on each architecture (plot row). The reported results are an average over five runs of DSE per application and architecture and 20 repetitions of the run-time thermal management experiment per DSE to incorporate diverse mixes of preoccupied resources for each background utilization level. The obtained results offer two major insights: First, the high success rate of task migration demonstrates the practicality of task migration also in a hard real-time context. Second, compared to mapping reconfiguration, task migration offers a substantially higher success rate, demonstrating its advantage over mapping reconfiguration as a real-time deployment adaptation approach. Among all applications and architectures, task migration exhibits an up to 95% higher success rate (35% on average), compared to mapping reconfiguration. This success difference roots in three advantages of task migration over mapping reconfiguration: Since it often involves the relocation of only a subset of the application's tasks, task migration (i) requires a smaller set of resources which increases its chances of passing the resource checks, (ii) imposes a lower timing overhead which increases its chances of passing the timing check, and, thanks to its lightweight timing analysis and feasibility check, (iii) enables the RM to consider all possible adaptation options instead of a restricted set of statically computed options.

Run-Time Overhead. During the RM's search for a destination tile, the application continues its regular execution. Thus, the overhead of the search process is not critical w.r.t. the real-time constraints. However, to fit for on-line use, this overhead – which is mainly due to the resource- and timing checks – must be acceptable. In Section 4.5, we demonstrated the scalability of the proposed analyses which were shown to exhibit a linear time complexity of $O(|T| + |M|)$. To assess their overhead in absolute time, in the thermal management experiment, we also record the time spent during the RM's search process before the first destination is found which passes both the resource- and the timing checks – performed on an Intel i7-4770 CPU at 3.4 GHz with 32 GiB of RAM. The records denote an average overhead of 1.08 ms (standard deviation of 0.16 ms) for the resource checks and 0.57 ms (standard deviation of 0.06 ms) for the timing check. According to the results, the overhead of the proposed migration timing analysis and feasibility check is by an average of 47 % lower than that of the resource check which verifies their lightness of for on-line use.

6 Conclusion

In this paper, we proposed a predictable migration mechanism supported with a migration timing analysis and feasibility check to enable hard real-time task migrations in composable many-core systems. The proposed migration mechanism complies with the distributed memory scheme of many-core systems, and its supporting analysis is lightweight and, therefore, applicable for on-line use. Experimental results demonstrate the feasibility of hard real-time task migrations, the lightness of the proposed timing analysis and feasibility check for on-line use, and the advantage of the proposed task migration mechanism over mapping reconfiguration as the state-of-the-art hard real-time adaptation approach.

References

1 Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP journal on embedded systems*, 2008(1):518904, 2007.

2 Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. Composability and predictability for independent application development, verification, and execution. In *Multiprocessor System-on-Chip*, pages 25–56. Springer, 2011.

3 Zaid Al-bayati, Brett H Meyer, and Haibo Zeng. Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 57–62, 2016.

4 Gabriel Marchesan Almeida, Sameer Varyani, Rémi Busseuil, Gilles Sassatelli, Pascal Benoit, Lionel Torres, Everton Alceu Carara, and Fernando Gehm Moraes. Evaluating the impact of task migration in multi-processor systems-on-chip. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design (SBCCI)*, pages 73–78, 2010.

5 Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 15–20, 2006.

6 Eduardo Wenzel Brião, Daniel Barcelos, Fabio Wronski, and Flávio Rech Wagner. Impact of task migration in NoC-based MPSoCs for soft real-time applications. In *Proceedings of the IFIP International Conference on Very Large Scale Integration*, pages 296–299, 2007.

7 William J Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 3(2):194–205, 1992.

8 Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey

Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013.

9 Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transaction on Evolutionary Computation (TEVC)*, 6(2):182–197, 2002.

10 Robert Dick. Embedded system synthesis benchmarks suite (E3S), 2010. URL: `http://ziyang.eecs.umich.edu/~dickrp/e3sdd/`.

11 Piotr Dziurzanski, Amit Kumar Singh, and Leandro Soares Indrusiak. Multi-criteria resource allocation in modal hard real-time systems. *EURASIP Journal on Embedded Systems*, 2017(1):30, 2017.

12 Ashaf El-Antably, Olivier Gruber, Frederic Rousseau, and Nicolas Fournel. Transparent and portable agent based task migration for data-flow applications on multi-tiled architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 183–192, 2015.

13 Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

14 Weiwei Fu, Tianzhou Chen, Chao Wang, and Li Liu. Optimizing memory access traffic via runtime thread migration for on-chip distributed memory systems. *The Journal of Supercomputing*, 69(3):1491–1516, 2014.

15 Laurent Gantel, Salah Layouni, Mohamed El Amine Benkhelifa, François Verdier, and Stéphanie Chauvet. Multiprocessor task migration implementation in a reconfigurable platform. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 362–367, 2009.

16 Yang Ge, Parth Malani, and Qinru Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 579–584, 2010.

17 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

18 Jan Heisswolf, Ralf König, Martin Kupper, and Jürgen Becker. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 39(8):2603–2622, 2013.

19 Robert Hilbrich and J Reinier Van Kampenhout. Partitioning and task transfer on NoC-based many-core processors in the avionics domain. *Journal Softwaretechnik-Trends*, 30(3):6, 2011.

20 Simon Holmbacka, Wictor Lund, Sebastien Lafond, and Johan Lilius. Task Migration for Dynamic Power and Performance Characteristics on Many-Core Distributed Operating Systems. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 310–317, 2013.

21 Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 108–109, 2010.

22 Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: distributed resource management for on-chip many-core systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, 2011.

23 H Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2 edition, 2011.

24 Zao Liu, Sheldon X-D Tan, Xin Huang, and Hai Wang. Task migrations for distributed thermal management considering transient effects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, 23(2):397–401, 2015.

25 Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1723–1730, 2011.

26 Guilherme Madalozzo, Liana Duenha, Rodolfo Azevedo, and Fernando G Moraes. Scalability evaluation in many-core systems due to the memory organization. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 396–399, 2016.

27 Fabrizio Mulas, David Atienza, Andrea Acquaviva, Salvatore Carta, Luca Benini, and Giovanni De Micheli. Thermal balancing policy for multiprocessor stream computing platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(12):1870–1882, 2009.

28 Peter Munk and Jan Richling. Migration-aware WCET estimation for heterogeneous multicores. *ACM SIGBED Review*, 11(3):22–25, 2014.

29 Lionel M Ni and Philip K McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

30 Michele Pittau, Andrea Alimonda, Salvatore Carta, and Andrea Acquaviva. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 59–64, 2007.

31 Behnaz Pourmohseni, Fedor Smirnov, Stefan Wildermann, and Jürgen Teich. Isolation-Aware Timing Analysis and Design Space Exploration for Predictable and Composable Many-Core Systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 12:1–12:24, 2019.

32 Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Predictable Run-Time Mapping Reconfiguration for Real-Time Applications on Many-Core Systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*, pages 148–157, 2017.

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(2):433–469, 2019.

34 Felix Reimann, Martin Lukasiewycz, Michael Glaß, and Fedor Smirnov. OpenDSE – open design space exploration framework, 2018. URL: `http://opendse.sourceforge.net/`.

35 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

36 Tilera Corporation. Tile Processor Architecture Overview for the TILE-Gx Series, 2012.

37 Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. *ACM SIGBED Review*, 6(3):6, 2009.

38 Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Predictable Task Migration for Locked Caches in Multi-Core Systems. *ACM SIGPLAN Notices*, 46(5):131–140, 2011.

39 Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Design Automation Conference (DAC)*, pages 1–10, 2013.

40 Pranav Tendulkar and Sander Stuijk. A case study into predictable and composable MPSoC reconfiguration. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 293–300, 2013.

41 Andreas Weichslgartner, Stefan Wildermann, Michael Glaß, and Jürgen Teich. *Invasive Computing for mapping parallel programs to many-core architectures.* Springer, 2018.

42 Pascal T Wolkotte, Gerard JM Smit, Nikolay Kavaldjiev, Jens E Becker, and Jürgen Becker. Energy model of networks-on-chip and a bus. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 82–85, 2005.