

34th Euromicro Conference on Real-Time Systems

ECRTS 2022, July 5–8, 2022, Modena, Italy

Edited by

Martina Maggio



Editors

Martina Maggio 

Universität des Saarlandes, Department of Computer Science, Saarbrücken, Germany
Lund University, Department of Automatic Control, Sweden
maggio@cs.uni-saarland.de

ACM Classification 2012

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

ISBN 978-3-95977-239-6

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/978-3-95977-239-6>.

Publication date

July, 2022

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 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/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/LIPIcs.ECRTS.2022.0

ISBN 978-3-95977-239-6

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

LIPICs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Marko Bertogna and Martina Maggio</i>	0:vii
Organizers	
.....	0:ix

Papers

Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm	
<i>Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, and Giacomo Travaglini</i>	1:1–1:15
RTScale: Sensitivity-Aware Adaptive Image Scaling for Real-Time Object Detection	
<i>Seonyeong Heo, Shinnung Jeong, and Hanjun Kim</i>	2:1–2:22
ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems	
<i>Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, and Claire Pagetti</i>	3:1–3:19
Using Quantile Regression in Neural Networks for Contention Prediction in Multicore Processors	
<i>Axel Brando, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla</i>	4:1–4:25
A Formal Link Between Response Time Analysis and Network Calculus	
<i>Pierre Roux, Sophie Quinton, and Marc Boyer</i>	5:1–5:22
Unikernel-Based Real-Time Virtualization Under Deferrable Servers: Analysis and Realization	
<i>Kuan-Hsun Chen, Mario Günzel, Boguslaw Jablkowski, Markus Buschhoff, and Jian-Jia Chen</i>	6:1–6:22
A Mathematical Comparison Between Response-Time Analysis and Real-Time Calculus for Fixed-Priority Preemptive Scheduling	
<i>Victor Pollex and Frank Slomka</i>	7:1–7:25
General Framework for Routing, Scheduling and Formal Timing Analysis in Deterministic Time-Aware Networks	
<i>Anais Finzi and Ramon Serna Oliver</i>	8:1–8:23
Correctness and Efficiency Criteria for the Multi-Phase Task Model	
<i>Rémi Meunier, Thomas Carle, and Thierry Monteil</i>	9:1–9:21
Overrun-Resilient Multiprocessor Real-Time Locking	
<i>Zelin Tong, Shareef Ahmed, and James H. Anderson</i>	10:1–10:25
Scheduling Offset-Free Systems Under FIFO Priority Protocol	
<i>Matheus Ladeira, Emmanuel Grolleau, Fabien Bonneval, Gautier Hattenberger, Yassine Ouhammou, and Yuri Hérouard</i>	11:1–11:19

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks <i>Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri</i>	12:1–12:22
Response-Time Analysis for Self-Suspending Tasks Under EDF Scheduling <i>Federico Aromolo, Alessandro Biondi, and Geoffrey Nelissen</i>	13:1–13:18
An Approach to Formally Specifying the Behaviour of Mixed-Criticality Systems <i>Alan Burns and Cliff B. Jones</i>	14:1–14:23
Achieving Isolation in Mixed-Criticality Industrial Edge Systems with Real-Time Containers <i>Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte</i>	15:1–15:23
Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds <i>Reza Miroshanlou, Mohamed Hassan, and Rodolfo Pellizzoni</i>	16:1–16:27
Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems <i>Mohamed Hossam and Mohamed Hassan</i>	17:1–17:23
RT-DFI: Optimizing Data-Flow Integrity for Real-Time Systems <i>Nicolas Bellec, Guillaume Hiet, Simon Rokicki, Frederic Tronel, and Isabelle Puaut</i>	18:1–18:24
Foundational Response-Time Analysis as Explainable Evidence of Timeliness <i>Marco Maida, Sergey Bozhko, and Björn B. Brandenburg</i>	19:1–19:25
Using Markov’s Inequality with Power-Of-k Function for Probabilistic WCET Estimation <i>Sergi Vilardell, Isabel Serra, Enrico Mezzetti, Jaume Abella, Francisco J. Cazorla, and Joan del Castillo</i>	20:1–20:24

■ Preface

Message from the Chairs

We welcome you to ECRTS 2022, in Modena, Italy. This is especially a pleasure given that ECRTS 2022 follows two editions that were forced to be virtual by the global pandemic.

Alongside RTSS and RTAS, ECRTS ranks as one of the top three international conferences on real-time systems, and is the premier European conference series on this topic. Given the lessons learned during the pandemic, this year ECRTS includes the possibility to participate online. We are delighted to have you join the first *hybrid* ECRTS, for an exciting program consisting of both scientific talks and opportunities for socializing and collaborative work.

ECRTS has been at the forefront of recent innovations in the real-time systems community such as artifact evaluation, open-access proceedings, and a flexible page limit. This year we have consolidated the experience, and repeated a double-blind submission process with flexible page limit, that does not constrain the authors, and allows them to put the effort into optimizing the content of their submission, rather than space utilization.

ECRTS 2022 received a total of 52 submissions from Asia, Europe, and North America. Each submission was reviewed by at least three expert members of the program committee and discussed at a virtual committee meeting that took place on April 5 and 6, 2022. The program committee accepted 19 papers for publication and presentation, which translates to an acceptance rate of 37%. In addition, on the scientific side, the ECRTS industrial challenge will be presented and discussed at the conference, following a long-lasting tradition of industrial challenges coming from the WATERS workshop.

A major conference such as ECRTS rests on many shoulders. First of all, we would like to thank the program committee members, for their hard work despite all the burdens of yet another challenging year. Similarly, thanks to all external and secondary reviewers, who provided many valuable perspectives and important feedback. We are especially grateful to those PC members and additional reviewers who went “above and beyond” serving as shepherds. We would also like to extend our thanks to the Artifact Evaluation Chairs, Matthias Becker and Angeliki Kritikakou, and their board of Artifact Evaluators for running the AE process, and to the Real-Time Pitches Chair, Timothy Bourke, for bringing fresh new ideas and discussions to the conference.

Finally, we would like to thank the ECRTS Executive Committee, Sebastian Altmeyer, Sophie Quinton, and Marcus Völp, for the outstanding service to the community, and the long-serving Euromicro Real-Time TC Chair Gerhard Fohler for developing ECRTS into what it is today, and for agreeing to give a retrospective talk on the history of ECRTS. Last but not least, we thank all authors for submitting to ECRTS 2022. Whether or not the submission was ultimately accepted for publication, we deeply appreciate your fine work and the tremendous effort and care that has gone into it; this conference would not be possible without you.

We are looking forward to an inspiring scientific program in Modena and online. Please join us in enjoying both the technical content and everything around it, especially with the return to in-person events.

Marko Bertogna
General Chair ECRTS 2022

Martina Maggio
Program Chair ECRTS 2022



■ Organizers

Euromicro Real-Time Technical Committee

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

General Chair

Marko Bertogna, Università degli Studi di Modena e Reggio Emilia, Italy

Program Chair

Martina Maggio, Universität des Saarlandes, Germany, and Lund University, Sweden

Artifact Evaluation Chairs

Angeliki Kritikakou, IRISA, Rennes, France
Matthias Becker KTH Royal Institute of Technology

Real-time pitches chair

Timothy Bourke, INRIA, France

Local Organization Team

Benjamin Rouxel, Università degli Studi di Modena e Reggio Emilia, Italy
Francesco Guaraldi, Università degli Studi di Modena e Reggio Emilia, Italy
Filippo Muzzini, Università degli Studi di Modena e Reggio Emilia, Italy

Program Committee

Sebastian Altmeyer, University of Augsburg, Germany
Sanjoy Baruah, Washington University in St. Louis, United States of America
Andrea Bastoni, TU Munich, Germany
Matthias Becker, KTH Royal Institute of Technology, Sweden
Marko Bertogna, Università degli Studi di Modena e Reggio Emilia, Italy
Marc Boyer, ONERA, France
Björn Brandenburg, Max Planck Institute for Software Systems (MPI-SWS), Germany
Daniel Bristot de Oliveira, Red Hat, Italy
Daniel Casini, Scuola Superiore Sant'Anna, Pisa, Italy
Francisco Cazorla, Barcelona Supercomputing Center, Spain
Anton Cervin, Lund University, Sweden
Thidapat Chantem, Virginia Tech, United States of America
Rolf Ernst, TU Braunschweig, Germany
Nathan Fisher, Wayne State University, United States of America
Gerhard Fohler, TU Kaiserslautern, Germany
Steve Goddard, University of Iowa, United States of America
Giovani Gracioli, Federal University of Santa Catarina, Brasil

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio



Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

0:x Organizers

Angeliki Kritikakou, University of Rennes 1, France
Risat Mahmud Pathan, Zenuity, Sweden
Renato Mancuso, Boston University, United States of America
Julio Medina, Universidad de Cantabria, Spain
Geoffrey Nelissen, Eindhoven University of Technology, The Netherlands
Alessandro Papadopoulos, Mälardalen University, Sweden
Rodolfo Pellizzoni, University of Waterloo, Canada
Linh Thi Xuan Phan, University of Pennsylvania, United States of America
Isabelle Puaut, INRIA, France
Sophie Quinton, INRIA, France
Jan Reineke, Universität des Saarlandes, Germany
Christine Rochange, University of Toulouse, France
Stefanos Skalistis, Collins Aerospace, Ireland
Leandro Soares Indrusiak, University of York, The United Kingdom
Marcus Völp, University of Luxembourg, Luxembourg
Georg von der Brüggen, TU Dortmund, Germany
Bryan Ward, Massachusetts Institute of Technology, United States of America
Dirk Ziegenbein, Bosch GmbH, Germany

Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm

Matteo Andreozzi ✉

Arm, Cambridge, United Kingdom

Giacomo Gabrielli ✉ 

Arm, Cambridge, United Kingdom

Balaji Venu ✉

Arm, Cambridge, United Kingdom

Giacomo Travaglini ✉

Arm, Cambridge, United Kingdom

Abstract

High-performance real-time systems are becoming increasingly common in several application domains, including automotive, robotics, and embedded. To meet the growing performance requirements of the emerging applications, these systems often adopt a heterogeneous System-on-Chip hardware architecture comprising multiple high-performance CPUs and one or more domain-specific accelerators. At the same time, the applications running on these systems are subject to stringent real-time and safety requirements. Due to the non-deterministic execution model of the compute elements involved and the co-location of the workloads, which leads to contention of the shared hardware resources, designing and orchestrating such applications is particularly challenging. In fact, the demand for novel methodologies, tools, and best practices to assist application designers working on high-performance real-time systems has never been stronger.

To stimulate innovation in this area, this document outlines an industrial case study from the automotive domain targeting an Arm-based hardware platform. The selected application is an augmented reality head-up display, which can be considered a representative example of a high-performance real-time use case. This case study will serve as the basis for a (multi-year) challenge involving real-time and embedded systems researchers across academia and industry that will be kicked off at the 34th Euromicro Conference on Real-Time Systems (ECRTS) 2022.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computing methodologies → Modeling and simulation

Keywords and phrases real-time, worst-case execution time

Digital Object Identifier 10.4230/LIPICs.ECRTS.2022.1

Acknowledgements The authors would like to thank Jay Cha (Arm) for his contribution to the definition of the case study, and Sophie Quinton (Inria), Martina Maggio (Saarland University) and Marko Bertogna (University of Modena) for their insightful comments and suggestions.

1 Introduction

As computing becomes ubiquitous, we observe increased interactions between devices and the physical world, which implies dealing more often with *real-time* and *safety* requirements. At the same time, the growing complexity of the end applications and the amount of data to be processed contribute to raising the *performance* requirements of the compute devices. These two trends lead to the proliferation of *high-performance real-time systems*.

Such systems are usually characterized by the co-location of multiple workloads on a single compute substrate, typically a heterogeneous System-on-Chip (SoC). This is done to improve the overall utilization of system resources by enabling their reuse (e.g., IO devices,



© Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, and Giacomo Travaglini; licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 1; pp. 1:1–1:15



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

hardware accelerators, etc.), and to improve the efficiency of data sharing across workloads. Workloads executing on such systems are defined as *mixed-criticality* [14]: each of them requires a potentially different *Level of Service* from the system, e.g., requiring a deadline to be met, a certain memory latency not to be exceeded, or bandwidth of requests being satisfied by a compute unit such as a GPU, with varying consequences, from soft recoverable errors to catastrophic failures, if those requirements are not met by the system at any point in time.

Co-locating multiple workloads with varying levels of criticality and priority comes at the cost of potential performance degradation due to the risk of interference between these workloads executing on shared resources, and thus increases the complexity of real-time analysis. The challenge in designing mixed-criticality systems is primarily to guarantee sufficient partitioning/isolation while still achieving high performance. Hence, it becomes necessary to be able to provision resources in a quantifiable and predictable way, regardless of whether the execution time of a workload may have non-deterministic external dependencies, such as non-deterministic data values or the arrival of non-deterministic events (e.g., interrupts). This is crucial for computing the *Worst-Case Execution Time* (WCET) for the real-time workloads being executed on the platform, and to ensure smooth and responsive operation of the general-purpose operating system (GPOS) workloads.

Addressing the above issues in a comprehensive way is one of the biggest challenges faced by system and application designers across various domains, in particular in the automotive, robotics, and Internet of Things (IoT) sectors. For this reason, we believe that it is very important to stimulate research across industry and academia around high-performance real-time themes. In this context, we would like to introduce the real-time research community to the Industrial Challenge associated with the Euromicro Conference on Real-Time Systems (ECRTS). Based on the success of the past editions of the challenge, which were part of the Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), a satellite workshop of ECRTS, we plan to follow a similar format: the challenge participants will be asked to address a specific set of initial questions, targeting approximately a 1-year timeline, and, based on the interest of the community and the reception of the initial set of activities, we will propose additional, more advanced, activities to be addressed in the following years. This document provides an outline of the scope and activities that we envision for the challenge. We propose an augmented reality head-up display application as a motivating case study, which we hope will provide an interesting testbed for innovative approaches in the areas of tools, methodologies and best practices to analyze high-performance real-time systems.

While the description of the case study provided in this document is a good starting point for the groups willing to start working on the challenge, we anticipate that further refinements to the definition of the case study and/or to the challenge activities will be released in the next months, together with deliverables including simulation tools, profiling data and reference input sets for the various software tasks. The web page <https://www.ecrts.org/arm-industrial-challenge/> will be used to share such updates and deliverables. If participants will have additional questions, we encourage them to submit them to the `#industrial-challenge` Discord channel – we will monitor the channel and address those questions in a timely fashion.

In the remainder of this document, Section 2 will describe the case study considered in detail. Section 3 will cover the key activities of the challenge, including a description of the related work; while some of the related work is mainly pertinent to the initial set of questions addressed to the challenge participants, we cover other related work that could be

relevant to follow-up questions, in order to expand the scope of the challenge. Section 4 will present the recommended platforms for the evaluation of the case study. The resources that will be provided to the challenge participants, which include analysis tools, pointers to the recommended software implementations for the main application tasks, and profiling data, will be covered in Section 5. We will conclude with final remarks in Section 6.

2 Case Study

The case study selected for the Industrial Challenge 2022 is an augmented reality head-up display application (AR HUD) for the automotive market, appropriately simplified to allow the study of its real-time aspects within the anticipated timeline and scope of the challenge.

AR HUDs extend the exterior view of the traffic conditions in front of the vehicle with virtual information (augmentations) for the driver. They are used to improve the situational awareness of drivers by displaying graphics that interact with the driver's field-of-view (FOV). The information provided is generated from real-time sensor data and typically includes advanced driver assistance system (ADAS) alerts and navigational cues overlaid on real-world objects. AR HUDs have started to appear in high-end cars and are expected to become a relatively common feature in the future due to the safety and comfort enhancements that they bring to the driving experience. At the same time, given their demanding compute and real-time requirements, AR HUDs are good examples of high-performance real-time applications where the interactions between the software tasks and the utilization of the shared hardware resources need to be carefully orchestrated to meet the desired functionality and performance goals.

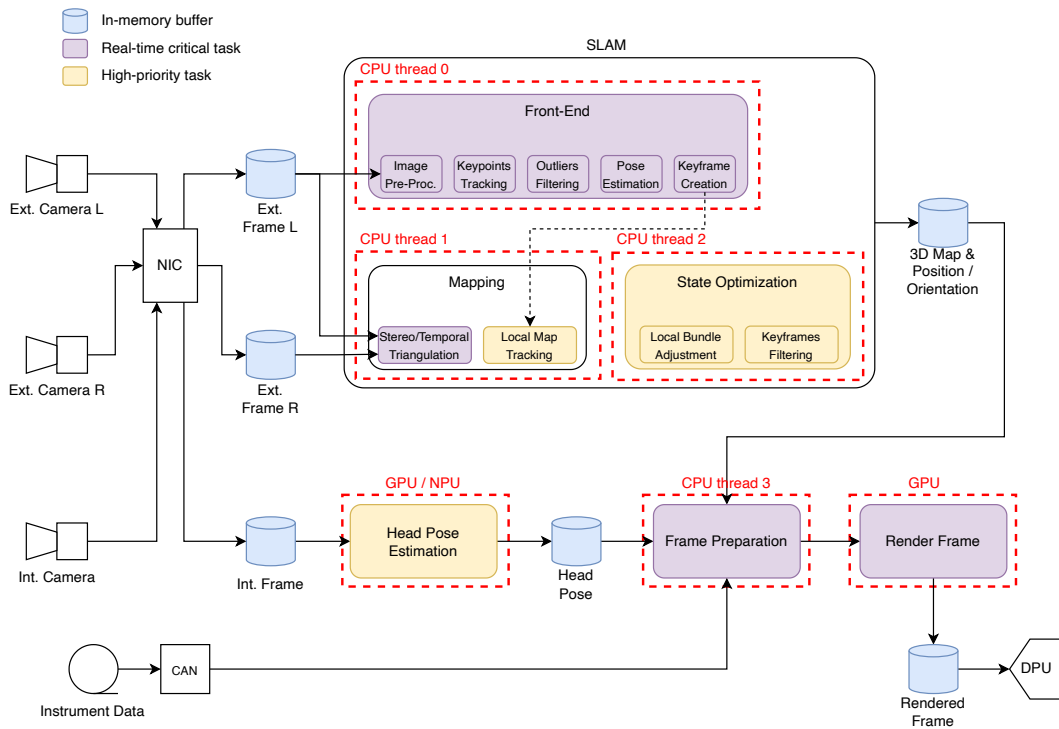


■ **Figure 1** An example of AR HUD (from [13]).

A key requirement for AR HUDs is the ability to project the images with enough positional accuracy to create the illusion that they appear as "fused" with the real world. This can be particularly challenging in driving scenarios due to the rapidly changing environment and the amount of sensor data to process and to present to the driver. In addition, to adjust the image to the driver's viewpoint, an eye tracking or head pose estimation function is normally used, which determines the appropriate amount of rotation/distortion to apply to the image frame. Another important requirement for AR HUDs is the ability to project images at a sufficient distance in front of the driver (around 10m): projecting images at longer distances reduces the accommodation time for the eyes between the real world and the HUD images; in addition, the ability of the human vision system to distinguish depth from other real-world

objects diminishes greatly beyond 7m – this factor can have implications on the complexity of the required eye tracking/head pose estimation function. Finally, high frame rates are necessary to avoid negatively impacting the user experience.

The implementation of an AR HUD system requires several functions, which can be decomposed into different software tasks. Figure 2 shows those functions, their decomposition into software tasks and a hypothetical mapping of such tasks onto the SoC compute resources (please refer to Section 2.5 for more details on the target hardware platform). The decomposition shows that we have a mix of real-time critical tasks and high-priority tasks, which can be mapped onto different concurrent CPU threads, thus making this case study non-trivial for real-time analysis. The *head pose estimation* task, implemented through a neural network, benefits from being mapped onto a high-throughput accelerator, either a GPU or a dedicated Neural Processing Unit (NPU). The following subsections will provide a detailed description of the tasks.



■ **Figure 2** Software tasks comprising the AR HUD case study and their hypothetical mapping onto the SoC compute resources. Purple blocks represent real-time critical tasks, while yellow blocks represent non-critical, high-priority, tasks. Red contours highlight the mapping to the compute blocks, i.e., CPU threads and GPU/NPU tasks. The arrows indicate the high-level data flow.

2.1 SLAM

A Visual Simultaneous Localization and Mapping (SLAM) function is required to determine the orientation and trajectory of the vehicle and to generate a map of its surroundings. The output of this function is then used to generate the HUD graphic images and to position them within the driver’s FOV.

The SLAM implementation selected for this case study is based on OV²SLAM [5], a high-performance feature-based SLAM supporting both monocular or stereo camera setups. While other SLAM techniques can offer improved accuracy, OV²SLAM is one of the techniques

with a publicly available implementation that aims at supporting real-time performance for a variety of realistic scenarios, (e.g., including autonomous driving), and can trade-off accuracy to maintain the required real-time performance.

For this case study, we assume a stereo camera setup: the SoC is connected to a pair of external cameras through Automotive Ethernet. The cameras have High-Definition (HD) resolution (1920x1080 pixels) and frame acquisition is expected to be performed at a relatively high target rate, in the range 30-60 frames per second (FPS). A Network Interface Card (NIC) receives new frames from the cameras and deposits them into a memory buffer, so that they can be read by the consumer tasks.

The *front-end* task includes the following sub-tasks:

- *image pre-processing*: a contrast enhancement technique is applied to all new frames to increase the dynamic range. Dynamic range in photography describes the ratio between the maximum and minimum measurable light intensities. Bright parts of the image can get much brighter, so the image seems to have more "depth" aiding the following stages of processing. The algorithm also limits the intensity changes due to exposure adaptation as the car drives through bright and shaded regions;
- *keypoint tracking*: an optical flow algorithm is applied to determine the *keypoints* and their motion, based on a pyramidal implementation of the inverse compositional Lucas-Kanade (LK) algorithm; keypoints are interesting portions of the images (eg: corners of objects in the image) that are tracked in consecutive frames. The number of keypoints are generally configurable in the algorithm.
- *outlier filtering*: outlier keypoints are identified by applying RANSAC filtering based on the epipolar constraint and removed in order to improve the accuracy of the camera pose estimation;
- *pose estimation*: this is performed by minimization of the 3D keypoints reprojection errors using a robust Huber cost function;
- *keyframe creation*: if the number of tracked 3D keypoints (i.e. the keypoints with prior information on their real 3D position) w.r.t. the last keyframe gets under a threshold or if a significant parallax is detected, a new keyframe is created. The scenario of number of tracked 3D keypoints falling below a threshold when compared to 3D keypoints in the last keyframe occurs when there is drastic changes in the scene while driving. This will be detected and a new keyframe needs be created for the new scene.

More details for these sub-tasks are available in [5]. The front-end pipeline is fully monocular, limiting all its operations to frames provided by the left camera, even if a stereo setup is available.

The *mapping* thread is responsible for processing every new *keyframe* to create new 3D map points by triangulation (both *stereo* & *temporal triangulation* with a stereo camera setup) and to perform *local map tracking* in order to minimize drift. These two sub-tasks have different real-time requirements: triangulation needs to operate at the full frame rate as it is critical for keeping accurate pose estimation in the front-end; the local map tracking operation, on the other hand, does not need to run at the full frame rate and it is executed and aborted if a new keyframe is available. However, it is beneficial to keep the local map tracking task as a higher priority task than, for instance, general-purpose or background tasks, as its frequency of execution has an impact on the overall SLAM accuracy.

The *state optimization* thread is responsible for running a *local bundle adjustment* (BA) pass, that is applied to refine the poses of the most recent keyframes and 3D map points' positions, and a *keyframes filtering* pass, that is applied to filter redundant keyframes in order to reduce the runtime of future instances of BA.

As with most high-performance real-time SLAM implementations, OV²SLAM leverages multi-threading to achieve real-time performance. As highlighted in Figure 2, the considered implementation of OV²SLAM relies on three CPU threads.

While OV²SLAM supports loop closure, this feature is disabled for this case study: due to the nature of the AR HUD application, the construction of a global map is, in fact, largely unnecessary. Loop closure and global map construction are more useful in scenarios where the user will revisit the same region of the map, e.g., for an AR wearable use case. However, while driving, this is more of a rare scenario and hence global map construction is not required.

The source code for the implementation of OV²SLAM that we plan to use as reference for the challenge is listed in Section 5.3.

2.2 Head Pose Estimation

Before being rendered on the AR HUD, images usually require some forms of correction to accommodate for the real-time position of the driver’s viewpoint. Such corrections are usually applied by relying on the output of a eye tracking/gaze estimation or head pose estimation function. For that, there are many factors affecting the choice of the specific algorithm used and its complexity. As anticipated at the beginning of this section, the complexity of these solutions depends on some high-level design parameters of the HUD, like the virtual image distance – longer distances require more complex display hardware, but at the same time can alleviate the complexity of the eye tracking function and allow for simpler methods to be used. In this case study, to keep the problem tractable, we assume that a long virtual image distance is indeed achievable by the HUD and that a head pose estimation method is adequate to solve the issue of determining the driver’s viewpoint.

The implementation of the head pose estimation function selected for this case study is Hopenet, which is based on a convolutional neural network (CNN) approach [16]. Hopenet requires a simple RGB monocular camera as input – we assume that another HD camera is installed inside the vehicle, directed towards the driving position, providing input frames via Automotive Ethernet to the SoC at the same target rate of the external cameras (30-60 FPS range).

As CNNs are amenable to hardware acceleration, either through a GPU or through a more dedicated Neural Processing Unit (NPU) or machine learning accelerator, the task should be mapped to one of those compute units, based on the specific platform selected by the challenge participants in their evaluation (Section 4).

In terms of real-time requirements, the head pose estimation task is considered a non-real-time critical, but high-priority, task, as highlighted in Figure 2. The accuracy of the head pose estimation is important, but a small number of frames can be dropped without affecting the general functionality of the application.

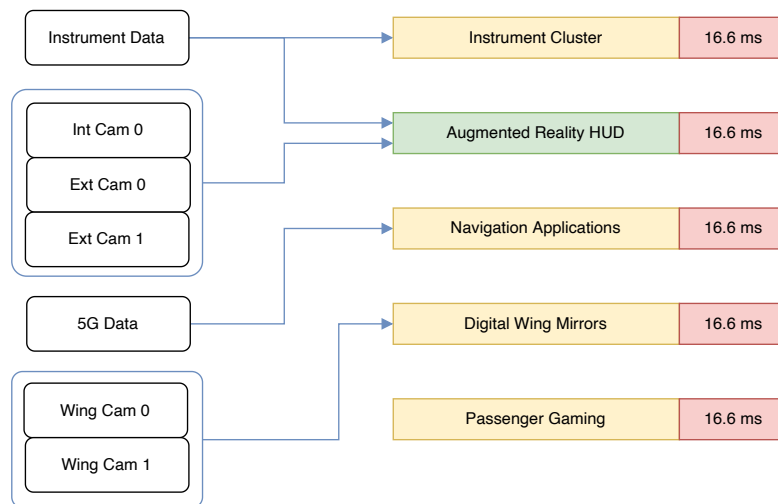
There are different versions of Hopenet publicly available, based on different input machine learning frameworks and resolutions/datasets. A reference implementation for Hopenet that we recommend is listed in Section 5.3.

2.3 Aggressor Workloads

While the software tasks required to implement the functionality of the AR HUD (summarized in Figure 2) constitute the main workload for the target system, as part of the challenge we will consider introducing other *aggressor workloads*, competing for the shared hardware resources, that will run alongside the main workload.

■ **Table 1** Parameters characterizing synthetic software tasks, for which reference implementations might not be available (e.g., proprietary/closed-source code bases). Modifications or additions to this list will be agreed with the challenge participants.

Parameter	Units
Compute intensity	FLOPS, IntOPS
Memory access bandwidth	MBps
Input memory buffer capacity	MB
Output memory buffer capacity	MB



■ **Figure 3** Example of co-located workloads and their deadlines (in red) for a modern digital cockpit.

2.4 Characterization of Tasks

In order to perform the activities described in Section 3, a characterization of the tasks of the considered case study will be required. For those tasks where a public software implementation is available, the characterization can be done by running/simulating the tasks on the selected evaluation platform and by either directly collecting the required statistics or by offline analysis of profiling information gathered during execution (see Section 4 for a description of the suggested evaluation platforms). This is the case, in particular, for the tasks belonging to the SLAM and Head Pose Estimation functions. The characteristics of the remaining tasks, including the aggressor ones, will be shared in the initial phase of the challenge. Different options will then be available on how to model the execution of such synthetic tasks on the evaluation platforms, that are partly discussed in Section 4.

Table 1 reports some key parameters that will be provided for each synthetic task.

2.5 Hardware Target

The target system for the considered case study is a platform comprising a cluster of high-performance Arm A-profile CPU cores, connected to a typical multi-level cache hierarchy configuration. The system will also feature a GPU and optionally an NPU.

The specific details of the hardware target will be shared in the initial phase of the challenge with the participants, and will take into account the specific evaluation platform chosen (Section 4).

3 Challenge Activities

The main tasks of the challenge are:

1. **Analysis of performance bounds:** Perform the response time analysis and worst-case execution time analysis following any methodology deemed suitable to the use case and platform considered. For this step, the challenge participants can choose to assume the absence of shared resources and other observable interference effects. The input for this first step is a description of the software tasks and their dependencies, commonly expressed as direct acyclic graphs (DAGs).
2. **Optimizations:** Perform one or more of the following optimization activities:
 - a. **Data-flow analysis:** Given the challenge use case decomposed into one or more software task DAGs, analyze its data-flows, resource usage and compute requirements, and work on one or more optimizations as follows.
 - b. **Scheduling:** Design one or more scheduling policies that can achieve better system utilization and data sharing between tasks.
 - c. **Resource mapping:** Efficient mapping of tasks to the various hardware components (processing nodes and resources) in the platform, in order to maximise efficiency and minimize contention.
 - d. **Shared resource interference monitoring and performance isolation:** Propose and/or implement shared resources monitoring strategies and design hardware and/or software techniques for shared resource contention mitigation.

3.1 Related Work

This section provides a short survey of real-time analysis techniques proposed in literature, in order to establish a common terminology and provide additional background to the challenge activities described at the top of this section.

3.1.1 Analysis of Performance Bounds

If shared resources in a system are required to provide a deterministic level of service, as defined in Section 1, then the performance of workloads executing on such a system can be found using various known methods, including static analysis, by measurement under worst-case conditions, or by means of formal tools. A static analysis attempts to estimate WCET without actually running the full software on the system. This approach can quickly estimate WCET, however requires the software to be profiled carefully and the traffic simulated on the system in order to come up with an accurate estimate [3, 12]. Another way of estimating performance bounds is by empirical means, i.e., by actually measuring it while running the mixed-criticality tasks on the system [20, 4]. This analysis requires due diligence during run-time for co-locating tasks that result in the highest amount of interference during their execution. We encourage the participants to make use of any tools they see best fit for this

challenge and/or they feel most comfortable with. We also encourage the participants to explore more than one technique or even a hybrid approach and to make recommendations based on the achieved results.

3.1.2 Data-flow Analysis

The data-flow model is simple and assumes that a workload can be broken into a series of tasks performed by or with the support of identifiable resources, that co-operate to achieve the higher-level objectives of such a workload in a non-trivial way. Workloads might be dependent on external events or inputs. When this applies, the mapping of tasks to specific resources might need to be determined dynamically at run-time. For tasks that can be executed in parallel (no immediate dependency between the tasks), assigning priorities is not a trivial task. Literature has demonstrated that a classic approach of assigning higher priorities based on the criticality levels and using them during scheduling decisions may result in poor system utilization [1, 19]. Novel run-time robustness mechanisms implemented in the OS/hypervisor layers that support graceful degradation of non-critical tasks partly addresses the system utilization problem [2, 1].

3.1.3 Analysis of Resource Sharing Policies

Shared resources contribute to the execution of a workload by providing one or more services to it. Some form of arbitration, either implicit or explicit, will regulate how users of resources are granted access to those, either in time or space (as in space of the resource) or both. If a resource is shared in time, for it to provide performance isolation, it must hold true that the service it provides to one of its users must be bounded without requiring knowledge of previously serviced users in time. If a resource is shared concurrently (in space), for it to provide performance isolation, it must hold true that the service it provides to one of its users must be bounded without requiring knowledge of concurrently serviced users.

Several techniques have been proposed in literature, e.g., restricting cache line evictions, cache coloring and partitioning, and also regulating memory traffic generated by a particular task in order to reduce the risk of interference between multiple workloads [21, 11, 8]. More recently, dedicated hardware support was added using FPGAs in order to strictly isolate the cores and avoid contention on shared resources such as last-level caches [6]. The focus on literature so far has been heavily around shared memory and bandwidth contention. We envisage that in future systems the problem will start surfacing around domain specific accelerators, that are usually shared across different tasks. This new class of accelerators comes with a different set of constraints due to the limited support for virtualization and preemption of tasks.

3.1.4 Monitoring of Shared Resource Interference

Monitoring shared resources provides insights into what causes contention on shared system resources and enables to implement contention mitigation strategies in the system. Upon determining the activities that make up a workload, we ask the participants how to implement or leverage a monitoring infrastructure that can enable observation of the shared resources involved in the computation of the system workloads. In detail, a monitor infrastructure should allow to determine the utilization of shared resources by their users to enable the setup and enforcement of service level agreements.

4 Evaluation Platforms

While the challenge activities can be addressed with different approaches and tools, we propose two evaluation platforms to be considered by the participants: either a virtual platform or a physical hardware development kit. It will be up to the participants to assess their preferred evaluation strategy for any of the activities. We will provide a comprehensive degree of support for the virtual platform approach, including a software starter kit to serve as baseline for the challenge, in order to streamline platform bring-up. Support for the hardware development kit will instead be provided on a best-effort basis.

Section 5 describes in more details what will be supplied to the participants over the course of the challenge.

Some details of the proposed platforms are reported below:

- Virtual platform: gem5 system-level simulator and AMBA Adaptive Traffic Profiles (ATP)
 - Link (gem5): <https://www.gem5.org/>
 - Link (ATP): <https://github.com/ARM-software/ATP-Engine>
 - Link (gem5 and AMBA ATP): <https://community.arm.com/arm-community-blogs/b/soc-design-and-simulation-blog/posts/amba-atp-engine-3-1-programmable-traffic-generation/>
- Hardware development kit: Jetson Xavier NX
 - Link: <https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-xavier-nx/>

As the selected use case will require compute intensive tasks, we propose the adoption of domain-specific accelerators to accelerate parts of it, as suggested in Figure 2. Ideally, we would like to enable modelling the full behaviour of the accelerators on both evaluation platforms. However, depending on the complexity of that approach, we will consider alternatives based on abstracting away the behaviour of the accelerator and other hardware components using traffic generators or approximate/synthetic models.

4.1 Virtual Platform

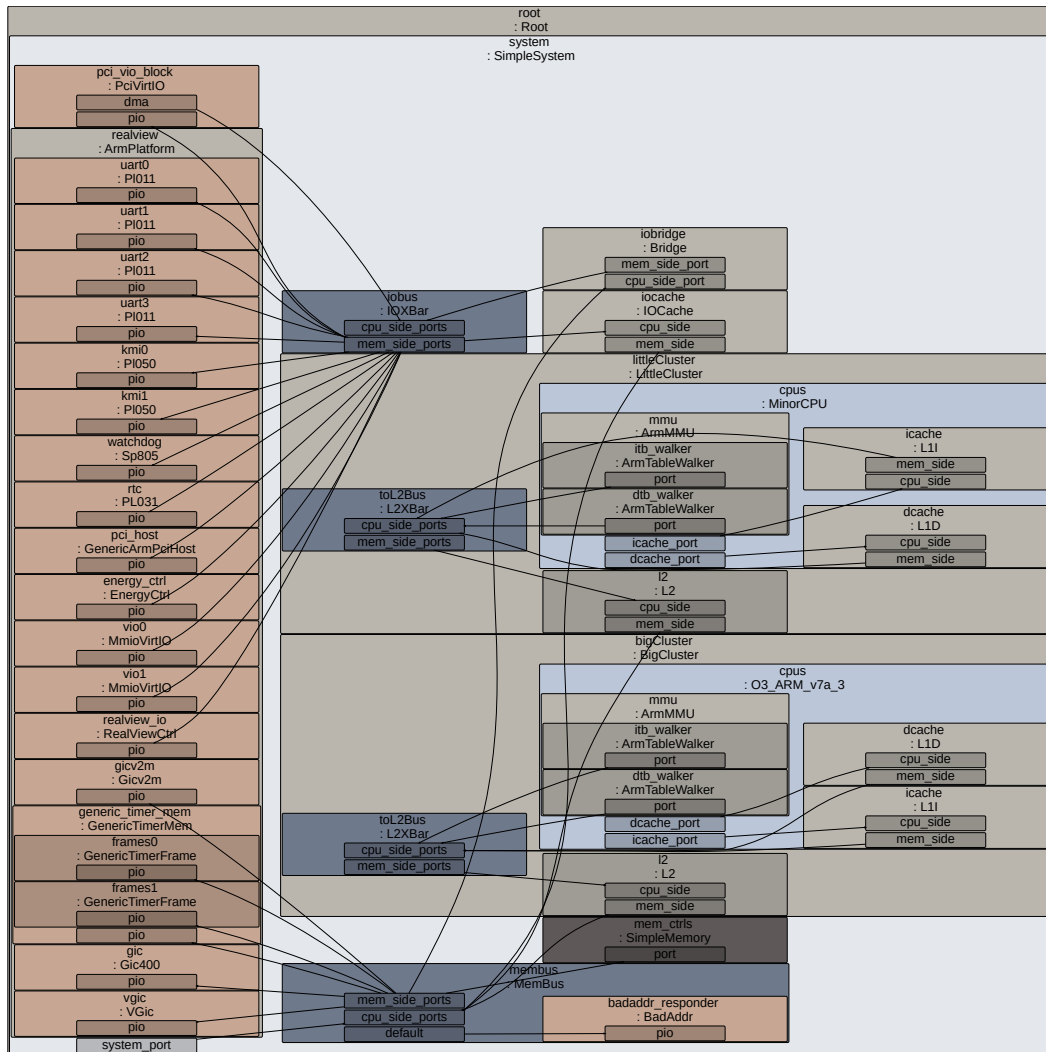
4.1.1 gem5

gem5 [9] is a modular event-driven simulation platform for computer architecture research, encompassing system-level architecture as well as processor micro-architecture modelling. It is widely used in both academia and industry for rapid early prototyping and/or design space exploration, and it has shown to be an effective tool for providing insights into the impact of system-level interactions when running complex workloads.

Its comprehensive model library (memories, IO devices, etc.) and the architectural support of Armv8-A features (up to Armv8.5-A) allows it to run unmodified complex workloads like Android and boot OSes from UEFI firmware implementations like TFA + u-boot / edkII [7].

Different CPU models, providing different degrees of abstractions and modelling fidelity, are provided in gem5, including two simple single-cycle-per-instruction models (*AtomicCPU*, *TimingCPU*), an in-order pipelined model (*MinorCPU*), and an out-of-order model (*O3CPU*). A memory system can be flexibly built out of caches and crossbars or through the Ruby framework, which provides even more flexible memory system modelling.

gem5 is conceptually a Python library written in C++: the simulated platform is configured in Python (configuration stage), but the instantiated Python models have a matching C++ implementation that gets executed at a later stage (execution stage), once



■ **Figure 4** An auto-generated diagram of a system modelled with gem5, including various processing elements, memories and devices.

the simulation is started. This allows to get the best of both worlds: the agility for prototyping and configuring a virtual system in Python, and the execution speed of C++ compiled code, which is crucial to reduce the simulation time for complex systems. Some example configuration platforms are provided within the repository itself (see scripts in `configs/example/arm/`). Those are meant to be starting points for building more complex configurations, and computer architects are expected to extend or adapt them to closely match the system under study.

4.1.2 AMBA Adaptive Traffic Profiles

The AMBA Adaptive Traffic Profiles (ATP) is a definition of the transaction characteristics of an hardware interface. It includes information on the types of transactions and the timing characteristics of those transactions. Traffic profiles can be used during system simulation to represent the behavior of a component. The simulation uses a traffic profile definition to

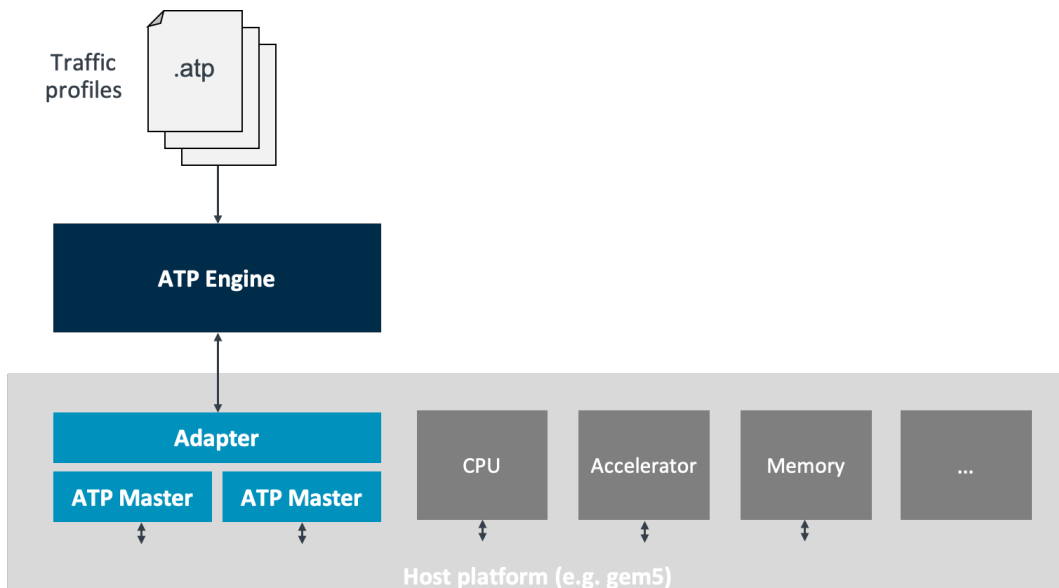
```

profile {
  type: READ
  master_id: "READS"
  fifo {
    start_fifo_level: EMPTY
    full_level: 2048
    TxnLimit: 30
    total_txn: 10000
    rate: "1GB/s"
  }
  pattern {
    address {
      base: 0
      increment: 64
    }
    size: 64
    wait_for: READ_RESP
  }
  name: "READ_EXAMPLE"
}
    
```

■ **Figure 5** An example of a simple .atp file

determine when a particular transaction should be issued, injecting synthetic traffic into the system under study. The AMBA ATP Engine is the open source implementation of the ATP specification. Its backbone is a lightweight FIFO model which injects transactions according to the provided traffic profile (specified through an .atp file).

The ATP Engine is plugged to gem5 (hosted solution) through an in-tree adaptor (see Figure 6) and this requires to build the ATP Engine as a gem5 loadable module (please follow the ATP Engine README.md guide). It is otherwise possible to build gem5 and ATP together with the meta-atp layer (link: <https://git.yoctoproject.org/meta-arm/tree/meta-atp/README.md>) from the meta-arm repository for Yocto.



■ **Figure 6** How to model heterogeneous systems by connecting the ATP engine to gem5

These items should be enough to build an evaluation platform using traffic profiles for the computing elements of the system. We won't provide functional NPU/GPU models; if challenge participants will be willing to add functional models of such accelerators to the starter kit simulation platform, prior work on integrating gem5 with approximated models [17, 15] can be used to facilitate the platform bring-up.

4.2 Hardware Development Kit

The hardware platform suggested as alternative to the virtual platform is the Jetson Xavier NX, whose SoC incorporates a 6-core NVIDIA Carmel Armv8.2 64-bit CPU, 384-core NVIDIA Volta GPU with 48 Tensor Cores, and two NVIDIA Deep Learning Accelerator (NVDLA) engines. The latter processing elements could be used to accelerate selected software tasks as specified in Figure 2.

5 Resources

This section provides a list of the resources, including tools, input data sets and profiling data, that will be provided for the challenge, and also the recommended open source implementations of the main software modules of the application in the considered case study.

5.1 Virtual Platform Starter Kit

For the virtual platform solution, we will supply:

- *gem5 starter kit*, including a specific system configuration (detailing platform composition).
- *AMBA ATP profiles*: traffic profiles for all tasks.

5.2 Hardware Development Starter Kit

For the hardware development kit, we will supply:

- *Software tasks source code*: source code for most of the CPU tasks; synthetic “busy-cycle” kernels for the remaining ones (e.g., aggressor tasks).

5.3 Recommended Open Source Software Implementations

- **OV²SLAM**:
Implementation of OV²SLAM available on GitHub at the following URL: <https://github.com/ov2slam/ov2slam>. This particular implementation targets CPUs, it leverages multi-threading, and it is written in portable modern C++; it has a few dependencies on widely available libraries and middle-ware (e.g., ROS [18]), which are described at the same URL.
- **Hopenet**:
Hopenet is available on GitHub at the following URL: <https://github.com/natanielruiz/deep-head-pose>. For this case study, we would recommend starting from Hopenet-lite, which is a lightweight version of Hopenet based on the simpler ShuffleNet V2 [10] network. The source code for Hopenet-lite is available on GitHub at the following URL: <https://github.com/OverEuro/deep-head-pose-lite>.

6 Conclusions

This document has described a high-performance real-time case study based on an augmented reality head-up display application from the automotive market. This application is a motivating example for the industrial challenge that will be kicked off at the ECRTS 2022 conference. An initial set of questions to prospective challenge participants has been presented, together with initial directions on how to carry out the activities on an Arm-based evaluation platform. Based on the experience from the past editions of the industrial challenge, we expect the definition of the challenge itself to evolve, based on further refinements of the use case and on feedback from the participants. The landing web page for the challenge (<https://www.ecrts.org/arm-industrial-challenge/>) will provide the latest information and it will be used to share deliverables with the real-time research community, including tools, input sets, profiling data, and pointers to reference software implementations. The `#industrial-challenge` Discord channel will be used to address questions from the challenge participants and the research community. We encourage everyone to reach out through the Discord channel or via email directly to the authors for clarifications, feedback and comments.

References

- 1 Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012. doi:10.1109/TC.2011.142.
- 2 S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, 2011. doi:10.1109/RTSS.2011.12.
- 3 Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified wcet analysis framework for multi-core platforms. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 99–108, 2012. doi:10.1109/RTAS.2012.26.
- 4 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, 2012. doi:10.1109/ECRTS.2012.31.
- 5 Maxime Ferrera, Alexandre Eudes, Julien Moras, Martial Sanfourche, and Guy Le Besnerais. Ov²slam : A fully online and versatile visual SLAM for real-time applications. *CoRR*, abs/2102.04060, 2021. arXiv:2102.04060.
- 6 Giovanni Gracioli, Rohan Tabish, Renato Mancuso, reza mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing mixed criticality applications on modern heterogeneous mp soc platforms. In *2019 ECRTS*, May 2019.
- 7 Adrian Herrera. Running trusted firmware-a on gem5. <https://community.arm.com/arm-research/b/articles/posts/running-trusted-firmware-a-on-gem5>, June 2020.
- 8 Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89, 2013. doi:10.1109/ECRTS.2013.19.
- 9 Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley

- Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannothe, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020. [arXiv:2007.03152](https://arxiv.org/abs/2007.03152).
- 10 Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design, 2018. [doi:10.48550/ARXIV.1807.11164](https://arxiv.org/abs/1807.11164).
 - 11 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013. [doi:10.1109/RTAS.2013.6531078](https://doi.org/10.1109/RTAS.2013.6531078).
 - 12 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014. [doi:10.1109/ECRTS.2014.20](https://doi.org/10.1109/ECRTS.2014.20).
 - 13 Panasonic automotive brings expansive, artificial intelligence-enhanced situational awareness to the driver experience with augmented reality head-up display. <https://na.panasonic.com/us/news/panasonic-automotive-brings-expansive-artificial-intelligence-enhanced-situational-awareness-driver>, January 2021.
 - 14 Michael Paulitsch, Oscar Medina Duarte, Hassen Karray, Kevin Mueller, Daniel Muench, and Jan Nowotsch. Mixed-criticality embedded systems – a balance ensuring partitioning and performance. In *2015 Euromicro Conference on Digital System Design*, pages 453–461, 2015. [doi:10.1109/DSD.2015.100](https://doi.org/10.1109/DSD.2015.100).
 - 15 Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. gem5-salam: A system architecture for llvm-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020. [doi:10.1109/MICRO50266.2020.00047](https://doi.org/10.1109/MICRO50266.2020.00047).
 - 16 Nataniel Ruiz, Eunji Chong, and James M. Rehg. Fine-grained head pose estimation without keypoints. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.
 - 17 Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. [doi:10.1109/MICRO.2016.7783751](https://doi.org/10.1109/MICRO.2016.7783751).
 - 18 Stanford Artificial Intelligence Laboratory et al. Robotic operating system. URL: <https://www.ros.org>.
 - 19 Sebastian Tobuschat, Moritz Neukirchner, Leonardo Ecco, and Rolf Ernst. Workload-aware shaping of shared resource accesses in mixed-criticality systems. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2014. [doi:10.1145/2656075.2656105](https://doi.org/10.1145/2656075.2656105).
 - 20 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008. [doi:10.1145/1347375.1347389](https://doi.org/10.1145/1347375.1347389).
 - 21 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. [doi:10.1109/RTAS.2013.6531079](https://doi.org/10.1109/RTAS.2013.6531079).

RTScale: Sensitivity-Aware Adaptive Image Scaling for Real-Time Object Detection

Seonyeong Heo ✉ 

Department of Information Technology and Electrical Engineering, ETH Zürich, Switzerland

Shinnung Jeong ✉

Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea

Hanjun Kim ✉

Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea

Abstract

Real-time object detection is crucial in autonomous driving. To avoid catastrophic accidents, an autonomous car should detect objects with multiple cameras and make decisions within a certain time limit. Object detection systems can meet the real-time constraint by dynamically downsampling input images to proper scales according to their time budget. However, simply applying the same scale to all the images from multiple cameras can cause unnecessary accuracy loss because downsampling can incur a significant accuracy loss for some images.

To reduce the accuracy loss while meeting the real-time constraint, this work proposes RTScale, a new adaptive real-time image scaling scheme that applies different scales to different images reflecting their sensitivities to the scaling and time budget. RTScale infers the sensitivities of multiple images from multiple cameras and determines an appropriate image scale for each image considering the real-time constraint. This work evaluates object detection accuracy and latency with RTScale for two driving datasets. The evaluation results show that RTScale can meet real-time constraints with minimal accuracy loss.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Parallel architectures; Software and its engineering → Real-time systems software; Computing methodologies → Neural networks; Computing methodologies → Object detection; Theory of computation → Scheduling algorithms

Keywords and phrases Real-time object detection, Dynamic neural network execution, Adaptive image scaling, Autonomous driving, Self-driving cars

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.2

Supplementary Material *Software (Source Code)*: <https://github.com/seonyheo/darknet>

Funding This work is supported by IITP-2020-0-01847, IITP-2020-0-01361, and IITP-2021-0-00853 through the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics.

Acknowledgements We thank the anonymous reviewers for their valuable feedback. We also thank the CoreLab members for their support and feedback during this work. (Corresponding author: Hanjun Kim)

1 Introduction

Real-time object detection in autonomous driving is crucial to avoid severe accidents. Autonomous cars have multiple cameras around their bodies [39] and use the cameras to detect objects on the road. Based on the object detection results, autonomous cars make decisions on how to control their braking system and steer their wheel. Since object detection provides essential visual information for autonomous cars, object detection must finish on time to make timely decisions. If autonomous cars fail to make decisions on time, they may hit pedestrians or other cars. Therefore, autonomous cars should detect objects on the road timely and accurately.



© Seonyeong Heo, Shinnung Jeong, and Hanjun Kim;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio; Article No. 2; pp. 2:1–2:22



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Recent advances in deep neural networks (DNNs) have brought real-time object detection into reality. With tens or hundreds of neural network layers, object detection networks predict regions on the input image (i.e., bounding boxes) that are likely to contain an object, and classify their object categories. In general, we call a whole end-to-end object detection network, including bounding box prediction and classification as an *object detector*.

Prior work [14, 13, 34, 17, 32, 36, 35, 6, 44] has proposed DNN architectures for object detection. Especially, one-stage object detectors such as SSD [32] and YOLO [34] enable fast and accurate object detection by integrating bounding box prediction and classification. For example, YOLO (You Only Look Once) achieves up to 45 frames per second with high accuracy [34]. Therefore, open-source autonomous driving platforms such as Autoware [24, 25] and Apollo [1] use YOLO-based networks for object detection.

Aside from designing a novel DNN architecture, various optimization techniques are available to enhance the detection speed of existing object detectors. One of the most popular approaches is model compression [16, 15, 28, 10], which reduces the computational cost by pruning parameters or using lower-precision numerics. However, the model compression techniques require fine-tuning the target network to minimize accuracy loss, so they could hardly reflect time-varying real-time constraints. Another approach is dynamic image scaling [8, 7], which can reduce the computational cost by dynamically downsampling input images. By adjusting the image size according to the time budget, dynamic image scaling can help satisfy dynamically changing real-time constraints.

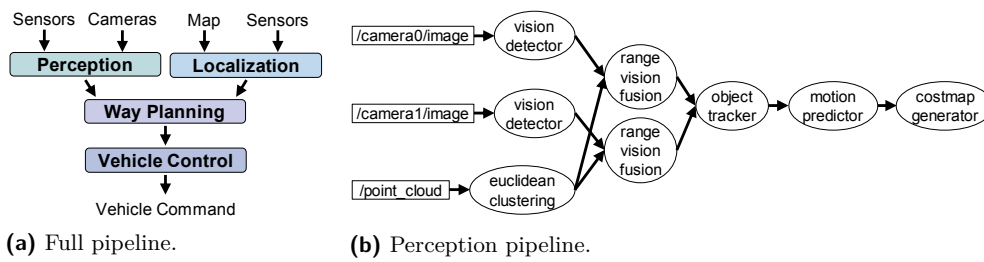
However, simply applying dynamic image scaling to object detection systems can cause unnecessary accuracy loss. With the state-of-the-art object detectors [6], we observe that each image has a different sensitivity to image scaling. Whereas image downsampling causes significant accuracy loss for some images, it barely causes accuracy loss for the other images. If an image only contains objects that are easy to detect even at a low scale, the image tends to be less sensitive to downsampling. Therefore, to reduce accuracy loss in image downsampling, it is necessary to consider the sensitivity in determining an appropriate scale for each image.

This work proposes RTScale, a new sensitivity-aware adaptive image scaling scheme for real-time object detection, which applies different scales to different images reflecting their sensitivities to image scaling and time budget. RTScale extends an existing object detector with a new scale sensitivity inference module and minimizes its overhead by reusing image features from the object detector. While offline, RTScale trains the sensitivity inference module to dynamically infer the impact of image scaling on the accuracy. While online, RTScale infers the scale sensitivities for multiple images from multiple cameras with the trained module, and determines appropriate image scales for the images considering the real-time constraint.

This work evaluates object detection accuracy and latency with two driving datasets: KITTI MOT [12] and BDD100K MOT [43]. This work implements RTScale on top of the state-of-the-art object detection framework [11]. The evaluation results show that RTScale can infer the sensitivity of images with low overhead and meet real-time constraints with minimal accuracy loss compared with another adaptive scaling scheme.

The contributions of this work are:

- Sensitivity-aware adaptive image scaling scheme for real-time object detection
- Sensitivity inference module which infers the scale sensitivity of an image based on its features
- Evaluation of the proposed approach with two real-world driving datasets



■ **Figure 1** Autonomous driving and perception pipeline.

2 Background and Motivation

2.1 Autonomous Driving and Object Detection

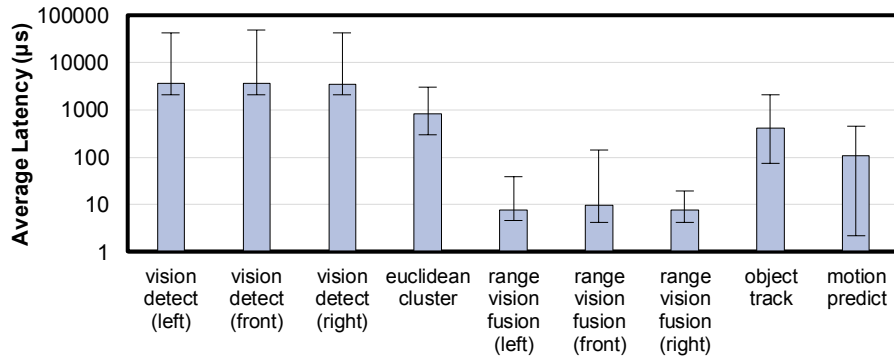
In general, autonomous driving consists of four primary jobs: perception, localization, way planning, and vehicle control. Autonomous driving systems first perceive their surrounding environment with cameras and sensors. At the same time, autonomous driving systems localize their locations using high-definition maps and sensors. Based on the perception and localization results, autonomous driving systems plan how to reach the destination avoiding obstacles on the road. Finally, autonomous driving systems determine how to control the vehicle according to the plan. Figure 1a briefly illustrates the autonomous driving pipeline.

Among the four primary jobs, perception plays an essential role in providing safe autonomous driving. Perception offers visual information on the road, such as whether pedestrians are near and how many cars are on the road. In practice, since high-definition maps only provide static information of the road, such as the locations of traffic lights, autonomous cars cannot solely rely on high-definition maps for safe autonomous driving. Through the perception process, autonomous cars can detect dynamic obstacles on the road and make the right decisions on their next movements. If autonomous cars cannot perceive the surrounding environment in real time, autonomous cars would fail to avoid obstacles on the road.

For perception, autonomous driving systems should process multiple camera images in practice. According to Tesla Model S owner’s manual [39], Tesla Model S cars use eight cameras for autonomous driving: one camera above the rear license plate, two cameras on each door pillar, two cameras on each front fender, and three cameras on the wind shield. According to prior work [29, 9], autonomous driving systems should finish the end-to-end processing within 100 ms. Therefore, autonomous driving systems should process all the images from multiple cameras within less than 100 ms.

Typically, autonomous driving systems implement perception with various computer vision algorithms such as object detection and object tracking. Figure 1b summarizes the perception pipeline process of Autoware [24, 25], one of the most popular open-source autonomous driving systems. When the system receives the images and point clouds from cameras and LIDAR, the system detects objects with the images and the point clouds. Then, the system fuses the objects from the images with the objects from the point clouds. Next, the system applies object tracking and motion prediction algorithms to the fused objects. Finally, the system obtains a cost map for way planning, which indicates the drivable area around the autonomous car.

In the perception process, object detection becomes the performance bottleneck incurring the most computational overhead. Figure 2 shows the profiling result of each task in the perception pipeline. For profiling, this work instruments time measuring code into the



■ **Figure 2** Latency profile of tasks in the perception pipeline in Figure 1b. (416, 416) image scale is used in object detection. Note that the y-axis is logarithmic.

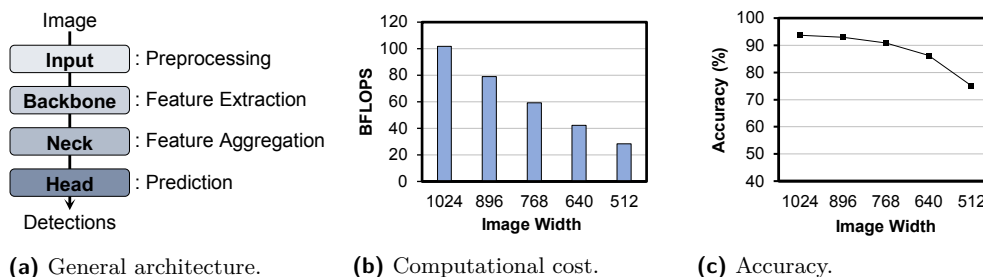
Autoware perception module [2]. Note that this work only measures the latency of inference, excluding queuing delay and communication time. In the graph, each error bar indicates the maximum and minimum latency of each task. The graph shows that (vision) object detection takes 4 ms while the other tasks take less than 1 ms on average. Therefore, the profiling result demonstrates that object detection takes the longest time among the different tasks in the perception pipeline.

2.2 Existing Object Detection Networks

In recent years, deep neural networks (DNNs) have remarkably enhanced both speed and accuracy of object detection. Figure 3a briefly describes the general structure of object detectors. An ordinary object detector comprises four parts: input, backbone, neck, and head [6]. The input part takes an image as an input and preprocesses the image. The backbone part contains a deep convolutional network such as ResNet [18] to extract the features of the input image. The neck part typically contains a small network that collects the features from different backbone stages such as FPN [30]. Finally, the head part predicts the location and category of objects.

There are two primary types of object detection networks: two-stage networks (e.g., R-CNN series networks [14, 13, 36, 17]) and one-stage networks (e.g., YOLO and SSD series networks [34, 35, 6, 32]). The major difference between two-stage and one-stage networks is on whether bounding box prediction and classification are separate or not. In the head part, two-stage networks find bounding boxes first and then classify objects in the bounding boxes. On the other hand, one stage networks predict bounding boxes and class probabilities together.

In general, one-stage networks are more compact than two-stage networks. For example, YOLO [34] with GoogLeNet [37] consists of 26 layers while Faster R-CNN with FPN [30] and ResNet-50 [18] consists of 213 layers in total. Since one-stage networks are more compact and faster than two-stage networks, most autonomous driving systems adopt one-stage networks like YOLO and SSD. For example, Autoware [24], a popular open-source autonomous driving system, allows to use either YOLO or SSD for object detection. Another open-source system, Apollo [1] also uses Apollo-OD for object detection which originates from YOLO networks.



■ **Figure 3** General architecture of existing object detectors and computational cost and accuracy of an existing object detector [6] with different image scales.

2.3 Dynamic Image Scaling

Dynamic image scaling is one of the optimization techniques that can accelerate object detection. It can enhance object detection speed by reducing the scale of the input image dynamically. Figure 3 shows how the computational cost changes as the image size changes on top of the existing deep learning framework [11]. Note that we downsample the input image to have the widths on the x-axis. In the graph, BFLOPS indicates the number of floating-point operations in billions. The graph shows that the computational cost decreases almost linearly as the image size decreases.

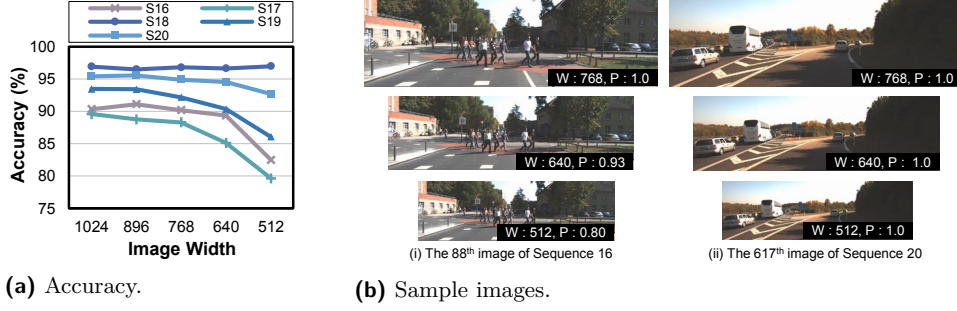
However, downsampling an image often incurs accuracy loss. Figure 3 also shows how the accuracy changes as the image size changes for the KITTI MOT dataset [12]. Here, we use mean average precision with IoU threshold = 0.5 (mAP_{50}) as the accuracy metric, which is commonly used to evaluate object detectors. The graph shows that the accuracy of the object detector tends to decrease as the image size decreases.

The interesting observation is that each image has a different sensitivity to image scaling in terms of accuracy. In other words, some images barely lose accuracy in downsampling while other images do not. In this work, *sensitivity* indicates how much image downsampling affects object detection accuracy for an image. If an image is highly sensitive to downsampling, it implies that the image would lose accuracy a lot in downsampling. This work will formally define the sensitivity in Section 4.

Figure 4a shows how the accuracy for each image sequence changes as the image size changes for the KITTI MOT dataset. The graph shows that some image sequence (S18) loses almost no point, but another image sequence (S17) loses 9.95 points at the minimum scale. Therefore, this work considers the images in S17 are more sensitive to image scaling than the images in S18. Thus, it is necessary to consider the sensitivity to image downsampling to reduce accuracy loss.

This work also observes that the sensitivity of an image differs according to the features of the image. Figure 4b is the collection of the sample images from the KITTI MOT dataset. The first image from S16 seems more complicated than the second image from S20. Whereas the first image contains objects that are difficult to detect (e.g., pedestrians on the road), the second image only contains simple objects (e.g., cars on the road). The difference may result in different sensitivities of S16 and S20. Since S16 has more complicated images than S20, S16 shows higher sensitivity than S20 as Figure 4a shows. Therefore, as the features of an image affect its sensitivity, we can infer the sensitivity based on the image.

Prior work [7] designs a dynamic image scaling scheme to enhance the accuracy of object detection, but the approach is not aware of real-time constraints nor sensitivity. It only predicts an optimal scale for a given image from a single image stream regardless of real-time



■ **Figure 4** Object detection accuracy with different image scales for each image sequence in the KITTI MOT validation dataset.

constraints. The approach cannot find the optimal scales for multiple image streams that respect the time constraint. Therefore, the existing approach is not suitable for real-time object detection with multiple image streams.

This work proposes RTScale, which enables real-time object detection with a new adaptive image scaling scheme considering the scale sensitivity of multiple input images. RTScale predicts the sensitivity of each image and determines the appropriate scales of images based on real-time constraints and sensitivities. In this way, RTScale can reduce accuracy loss in image downsampling while satisfying real-time constraints.

3 Problem Statement

When an autonomous car drives, the car receives N images from its N cameras in a fixed interval. When the images arrive, the object detector processes the images with M processing units and makes decisions based on the results. Let $I_{i,j}$ denote the image frame of the j -th camera at the i -th interval. Then, at the i -th interval, the object detector conducts object detection tasks $\tau_{i,1}, \dots, \tau_{i,N}$ for $I_{i,1}, \dots, I_{i,N}$. To make a timely decision, the object detector must finish processing the N images within deadline D_i at each interval. Note that the relative deadline can differ across intervals depending on dynamic execution environment.

Before processing the images, the scheduler determines the scales of the images and schedules the object detection tasks for the images. In this paper, we define an image scale as (w, h) where w and h are the width and height of the image. The object detector maintains the set of scales S and chooses an appropriate scale from the set for each image. In other words, at the i -th interval, the scheduler determines $s_{i,j}$ as one of the scales in S for $j = 1, \dots, N$ where $s_{i,j}$ denotes the image scale of $I_{i,j}$. The reason for assuming the predefined set of scales is that handling arbitrary image scales would incur considerable resizing overhead in practice because the system needs to configure the network layers every time if the input shape is arbitrary.

This work formulates the problem as follows.

Problem Statement. The problem is to find the optimal scales of input images, $\{s_{i,j}\}_{j \in [1,N]}$, and schedule the object detection tasks for the images, $\{\tau_{i,j}\}_{j \in [1,N]}$, at each interval i while satisfying the following constraint.

Deadline Constraint. The object detection tasks for the images finish before the deadline, i.e., for each $j = 1, \dots, N$,

$$f_{i,j} \leq r_{i,j} + D_i$$

where $r_{i,j}$ and $f_{i,j}$ are the release time and completion time of $\tau_{i,j}$, respectively.

■ **Table 1** Notation for the problem.

No.	Description	No.	Description
N	Number of cameras	$s_{i,j}$	Scale of $I_{i,j}$ for processing ($s_{i,j} \in S$)
M	Number of processing units	$\tau_{i,j}$	Object detection task for $I_{i,j}$
S	Set of scales to choose from	$r_{i,j}$	Release time of $\tau_{i,j}$
D_i	Relative deadline at the i -th interval	$f_{i,j}$	Completion time of $\tau_{i,j}$
$I_{i,j}$	The j -th image at the i -th interval	$\rho_{i,j}$	Scale sensitivity of $I_{i,j}$
$T[k]$	Latency of processing an image at $S[k]$		

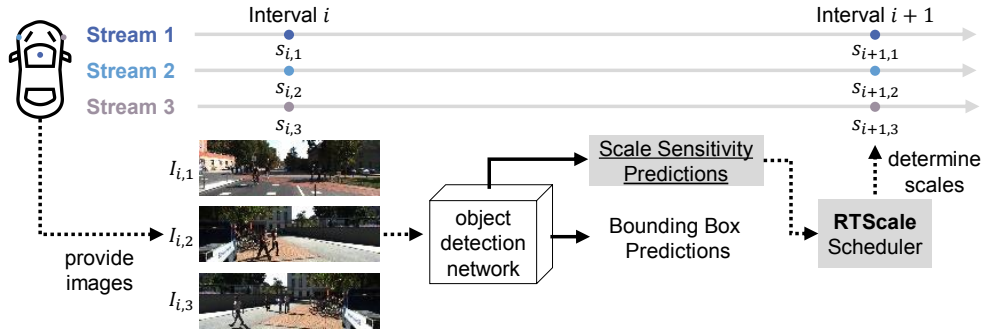
The ultimate goal of this work is to maximize object detection accuracy while satisfying the deadline constraint. If we only consider the deadline constraint, forcing a very low scale to the images could be a solution. However, it is not desirable because aggressive image downsampling can cause a huge accuracy loss. Since each image has a different sensitivity to image scaling as shown in Section 2, it is essential to minimize accuracy loss by considering the sensitivity.

This work uses four assumptions for the problem: (i) The frame interval is longer than or equal to the relative deadline. (ii) The worst-case execution time of processing an image at a scale is given. Based on prior work [5, 20, 21, 19], it is possible to estimate the worst-case execution time of an object detector. Especially, [19] presents a layer-level worst-case execution time model for general neural networks. Therefore, we can obtain the worst-case execution time of the target object detector by combining the estimation models of its network layers. (iii) Every image is equally important in terms of autonomous driving as in Autoware [24]. In other words, each image provides equally meaningful information for autonomous driving. (iv) The time to schedule the object detection tasks is negligible compared with the time to detect objects in images. To justify the fourth assumption, we measure the scheduling overhead in the experiment and observe that scheduling takes less than $10^{-5}\%$ of the total execution time as shown in Section 5.2.4.

4 Design

While most existing object detectors process images at a fixed scale, RTScale provides sensitivity-aware adaptive image scaling for real-time object detection. Figure 5 briefly illustrates the overall object detection process with RTScale. RTScale extends an existing object detector with a new lightweight sensitivity inference module that infers the scale sensitivity of an input image. Since the sensitivity inference module reuses the features extracted by the object detector, the module can predict scale sensitivity with a few convolution layers only. While offline, RTScale trains the sensitivity inference module with the ground-truth scale sensitivities of training datasets. While online, RTScale determines the scales of the next images based on the real-time constraints and the sensitivity prediction results of the current images.

In this paper, we assume a YOLO-series object detector [6] as a baseline object detector because it is one of the most popular object detectors in open-source autonomous driving platforms [24, 25, 1]. However, note that the proposed method can apply to any object detector as well because it is based on the common characteristics of ordinary existing object detectors.



■ **Figure 5** Overall object detection process with RTScale.

4.1 Scale Sensitivity Inference

This work defines *scale sensitivity* of an image ρ as the ratio of the accuracy obtained at the maximum scale to the accuracy obtained at the minimum scale, i.e.,

$$\rho = \frac{A[s^{max}]}{A[s^{min}]} \quad (1)$$

where $A[s]$ indicates the object detection accuracy of the image when detecting objects at a scale s , and s^{max} and s^{min} are the maximum and minimum scales in S , respectively.

The meaning of scale sensitivity is the accuracy loss ratio at a minimum scale. If an image loses accuracy a lot at a minimum scale, the image will have a high scale sensitivity. On the other hand, if an image loses little accuracy at a minimum scale, the image will have a low scale sensitivity. It is possible for a scale sensitivity to be smaller than one because object detectors sometimes can detect objects better at a lower scale. For example, if an image contains a very large object like a train, it may be better to process the image at a lower scale [7].

The scale sensitivity definition presupposes a monotonic relationship between image scale and object detection accuracy. This work has tried different metrics for the sensitivity to reflect the non-monotonic relationship between image scale and object detection accuracy as shown in Figure 4a. However, the sensitivity inference problem becomes too complex for the inference module to be trained. To simplify the sensitivity inference module, this work assumes that image scale and object detection are in a monotonic relationship.

Network Extension. This work extends an existing object detector to infer the scale sensitivity of an image along with bounding box prediction and classification. This work designs the scale sensitivity inference module to exploit the rich features from the existing object detector for scale sensitivity inference. As explained in Section 2, ordinary object detectors have a backbone network to extract the features from an input image and use the features to infer bounding box locations and categories. Since the sensitivity inference module also needs distinct features from the same input image, this work reduces the complexity of the sensitivity inference module by sharing the feature maps from the backbone network.

Table 2 describes the detailed architecture of the sensitivity inference module. The module applies two pairs of 3×3 and 1×1 convolution layers with the Leaky ReLU activation function [33] to the feature maps from the backbone network. Next, the module applies (global) average pooling to the output feature maps and obtains 512 features per image. Here, the global average pooling enables the module to get the same size of features for different image scales. Then, the module obtains the final sensitivity prediction by applying a dense layer with the linear activation function.

■ **Table 2** The architecture of sensitivity inference module (Input: Feature maps from the backbone network of which shape is $(N, 1024, H, W)$, Output: Normalized scale sensitivity).

Type	Kernel	Padding	Activation	Output Dim.
Convolution	3×3	1	Leaky	$(N, 512, H, W)$
Convolution	1×1	0	Leaky	$(N, 512, H, W)$
Convolution	3×3	1	Leaky	$(N, 512, H, W)$
Convolution	1×1	0	Leaky	$(N, 512, H, W)$
Average pooling	Global	-	-	$(N, 512, 1, 1)$
Dense	-	-	Linear	(N)

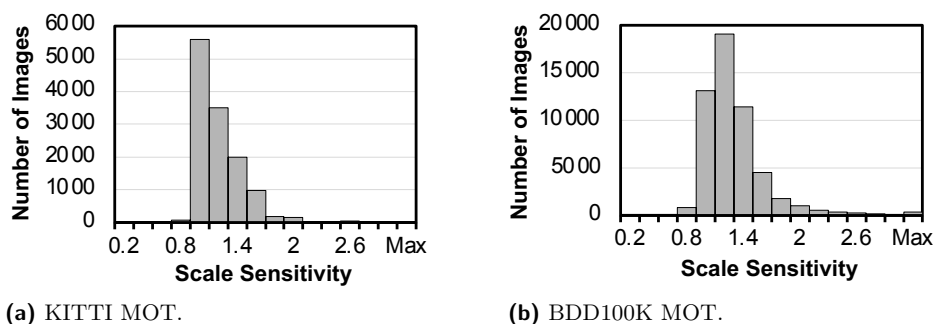
Ground-truth Label Generation. To train the scale sensitivity module, this work calculates the ground-truth sensitivities of the images from train datasets. For the minimum and maximum scales in S , this work records the object detection accuracy of the baseline object detector for each image in the train datasets. Then, this work calculates the ground-truth sensitivity of each image using (1).

This work uses the F1 score as the accuracy metric for scale sensitivity calculation to consider both precision and recall. Precision is insufficient to evaluate the accuracy for a *single* image because precision may favor low-resolution images with a smaller number of *valid* predictions (i.e., the total number of true positives and false positives of which confidence scores are larger than a threshold). In general, the total number of valid predictions tends to decrease as the image scale decreases. Thus, if there are few valid predictions of an image, its precision becomes undesirably high. Therefore, it is necessary to consider recall also to mitigate the problem. Note that mean average precision (mAP) is not applicable here because it is an accuracy metric for an entire dataset, not for a single image.

$$A[s] = \frac{2}{\frac{1}{pr[s]} + \frac{1}{rc[s]}} \quad (2)$$

Equation 2 is the definition of the F1 score where $pr[s]$ and $rc[s]$ denote the precision and recall of an image obtained at a scale s , respectively. In object detection, we regard a bounding box prediction is true positive if the predicted bounding box overlaps with a true bounding box with $\text{IoU} > 0.5$ and the predicted category is same as the category of the true bounding box. Here, IoU is the abbreviation of ‘‘Intersection over Union’’, which indicates the ratio of the overlap area of two bounding boxes to the total area of two bounding boxes.

To facilitate deep learning, this work normalizes the ground-truth sensitivity values with (3). In the equation, $\hat{\rho}^{min}$ and $\hat{\rho}^{max}$ indicate the minimum and maximum scale sensitivities, respectively. Note that $\hat{\rho}^{min}$ and $\hat{\rho}^{max}$ depend on the train datasets. Figure 6 shows the



■ **Figure 6** Scale sensitivity distribution of each driving dataset.

scale sensitivity distributions of two driving datasets, KITTI MOT [12] and BDD100K MOT [43]. The graphs show that BDD100K MOT has a broader distribution of scale sensitivity than KITTI MOT. Thus, this work uses a smaller $\hat{\rho}^{min}$ and a larger $\hat{\rho}^{max}$ for BDD100K MOT compared with KITTI MOT. In inference, this work denormalizes the predicted scale sensitivity for use in scheduling.

$$\rho^{norm} = (\rho - \hat{\rho}^{min}) / (\hat{\rho}^{max} - \hat{\rho}^{min}) \quad (3)$$

Training: This work regards scale sensitivity inference as a linear regression problem and trains the sensitivity inference module using a smooth L1 loss function with the ground-truth labels. Equation 4 is the definition of the loss function where y is the predicted sensitivity and ρ^{norm} is the normalized ground-truth sensitivity. This work accumulates the loss for each image when the mini-batch size is larger than one.

$$L(y, \rho^{norm}) = \begin{cases} 0.5 \times (y - \rho^{norm})^2 & \text{if } |y - \rho^{norm}| < 1 \\ |y - \rho^{norm}| - 0.5 & \text{otherwise} \end{cases} \quad (4)$$

4.2 Scheduling

The RTScale scheduler determines the scales of the next images considering both the real-time constraint and the sensitivity prediction of the current images. It is based on the assumption that two consecutive frames from the same image stream have similar image features. That is, two consecutive images in the same image stream would have similar scale sensitivities. We consider the assumption is reasonable because prior work on video object detection [48, 49, 47, 46] is also based on the assumption.

The basic idea for determining the scales is to minimize the expected accuracy loss relative to the accuracy obtained at the maximum scale. This work calculates the relative accuracy loss of a scale with scale sensitivity. Let $Q(k_1, k_2)$ be the ratio of the k_2 -th smallest scale over the k_1 -th smallest scale in S . For example, $Q(1, |S|)$ is the ratio of the maximum scale over the minimum scale in S . According to the definition,

$$Q(1, |S|) = \prod_{k \in [1, |S|-1]} Q(k, k+1)$$

Then, this work defines the expected accuracy gain of the k -th smallest scale relative to the minimum scale as follows:

$$gain(k, \rho) = \rho^{\log_{Q(1, |S|)} Q(1, k)} \quad (5)$$

Here, if $Q(1, 2) \simeq \dots \simeq Q(|S| - 1, |S|)$, we can simplify (5) as follows:

$$gain(k, \rho) = \rho^{\frac{k-1}{|S|-1}} \quad (6)$$

Finally, we define the expected loss of the k -th smallest scale relative to the maximum scale as follows:

$$loss(k, \rho) = \begin{cases} \frac{gain(|S|, \rho)}{gain(k, \rho)} = \rho^{\frac{|S|-k}{|S|-1}}, & \text{if } k \geq 1 \\ \infty, & \text{otherwise} \end{cases} \quad (7)$$

This work designs a scheduling algorithm that gradually finds the scale and schedule of each image that meet the time constraint for M (identical) processing units. After processing $I_{i,1}, I_{i,2}, \dots, I_{i,N}$ at the i -th interval, the object detector invokes the scheduler with the sensitivity predictions of the images. The algorithm takes the set of sensitivities

■ **Algorithm 1** RTScale scheduling algorithm.

Input : Sensitivities of the previous frames ρ_1, \dots, ρ_N
Relative deadline D

Output : Schedule of the next frames SC

```

1 if  $\lceil N/M \rceil \cdot T[1] > D$  then
2   | Return with an error
3 end
4  $SC \leftarrow \text{initialize}(\{\rho_j\}_{j \in [1, N]})$ 
5 while  $SC.\text{makespan} > D$  do
6   |  $j^* \leftarrow \arg \min_j \text{loss}(SC[j].\text{scale} - 1, SC[j].\text{sensitivity})$ 
7   |  $SC[j^*].\text{scale} \leftarrow SC[j^*].\text{scale} - 1$ 
8   |  $SC.\text{update}(j^*)$ 
9 end
10  $SC.\text{optimize}()$ 

```

■ **Algorithm 2** $\text{initialize}(\rho_1, \rho_2, \dots, \rho_N)$.

Input : Scale sensitivities ρ_1, \dots, ρ_N

Output : Initial schedule SC

```

1 for  $j \in [1, N]$  do
2   |  $SC[j].\text{id} \leftarrow j$ 
3   |  $SC[j].\text{sensitivity} \leftarrow \rho_j$ 
4   |  $SC[j].\text{scale} \leftarrow |S|$  // Maximum scale
5 end
6  $SC \leftarrow \text{Sort } SC \text{ in descending order of sensitivity}$ 
7  $W \leftarrow \{0, 0, \dots, 0\}$  // Workload of each unit
8 for  $j \in [1, N]$  do
9   |  $p^* \leftarrow j \bmod M$ 
10  |  $SC[j].\text{proc} \leftarrow p^*$ 
11  |  $SC[j].\text{track} \leftarrow W$ 
12  |  $W[p^*] \leftarrow W[p^*] + T[|S|]$ 
13 end
14  $SC.\text{makespan} \leftarrow \max_{p \in [1, M]} W[p]$ 

```

$\rho_{i,1}, \rho_{i,2}, \dots, \rho_{i,N}$ and the relative deadline for the next interval D_{i+1} as its inputs. Considering both the sensitivities and the relative deadline, the algorithm determines the scales of the next images $s_{i+1,1}, s_{i+1,2}, \dots, s_{i+1,N}$ and the assignment of processing units.

Algorithm 1 is the pseudo code of the RTScale scheduling algorithm. First, the algorithm checks whether the given images are schedulable or not by comparing the minimum possible *makespan* and the relative deadline (Line 1 to Line 3). After the test, the algorithm generates an initial schedule setting all the image scales as the maximum scale (Line 4). Then, the algorithm gradually lowers the scales until the makespan does not exceed the given relative deadline (Line 5 to Line 9). At every iteration, the algorithm finds the scale with the minimum expected loss, lowers the scale, and updates the schedule to reflect the change. Note that the algorithm assumes that S is sorted in advance.

Algorithm 2 and Algorithm 3 show how to find the minimum makespan schedule in detail. Since the minimum makespan scheduling problem is known as strongly NP-hard [40], the algorithms are based on a $\frac{4}{3}$ approximation algorithm which sorts the set of tasks in descending order of latency and greedily assigns each task to the processing unit with the minimum workload. Rather than calculating the entire minimum makespan schedule at every iteration, this work optimizes the algorithms to incrementally update the minimum

■ **Algorithm 3** $SC.update(j^*)$.

Input : Index of the target image j^*

- 1 $W \leftarrow SC[j^*].track$
- 2 **for** $j \in [j^*, N]$ **do**
- 3 $p^* \leftarrow \arg \min_p W[p]$
- 4 $SC[j].proc \leftarrow p^*$
- 5 $SC[j].track \leftarrow W$
- 6 $W[p^*] \leftarrow W[p^*] + T[SC[j].scale]$
- 7 **end**
- 8 $SC.makespan \leftarrow \max_{p \in [1, M]} W[p]$

makespan schedule. After sorted once in initialization, the order of the tasks remains the same as the main loop iterates. Therefore, the algorithms can simply update the necessary part of the schedule only (Line 2 and Line 7 in Algorithm 3).

After determining the schedule, Algorithm 1 optimizes the schedule as finalization. Since the algorithm reduces the scale of each image based on its expected accuracy loss, each processing unit may have spare time until the deadline. Therefore, the algorithm checks the amount of slack for each processing unit and increases the scale of each image if possible in the descending order of scale sensitivity.

The underlying principle of the algorithm is to lower image scales to satisfy the deadline constraint while minimizing its total accuracy loss. By reducing the scale of an image that has the minimal accuracy loss until satisfying the time constraint, the algorithm will obtain the maximal accuracy product of the scaled images that respects the time constraint.

The computation complexity of the scheduling algorithm is $O(N^2KM)$ where $K = |S|$. In the algorithm, the main loop for determining the next scales dominates the computation complexity of the algorithm. In the worst case that all images require the minimum scale, the outer loop iterates $N(K - 1)$ times, and the computation complexity of the **update** function is $O(NM)$. Therefore, the computation complexity of the algorithm is $O(N^2KM)$.

5 Evaluation

5.1 Experimental Setup

This work implements RTScale on top of the Darknet deep learning framework from prior work [11]. This work extends the state-of-the-art object detector [6] with the scale inference module in Section 4.1. Since the original object detector is supposed to use a single image scale, this work modifies the deep learning framework to dynamically change the scales of input images as the scheduler directs. Furthermore, this work enables the deep learning framework to support multiple image streams and multiple processing units for inference.

To show the effectiveness of RTScale, this work compares different image scaling schemes on top of the framework:

- **AvgScale**: chooses the maximum scale for each processing unit that does not violate the deadline constraint, i.e., choose the maximum scale such that $N_p \cdot T[k] \leq D_i$ where N_p indicates the number of images assigned to the p -th processing unit.
- **RTScale-pred**: chooses the scales using the proposed scheduling algorithm based on scale sensitivity predictions.
- **RTScale-gt**: chooses the scales using the proposed scheduling algorithm with ground-truth scale sensitivities.

This work evaluates the image scaling schemes with two driving datasets, KITTI MOT [12] and BDD100K MOT [43]. The reason for using the multiple object tracking datasets is that existing object detection datasets do not provide multiple streams of consecutive image frames. Existing object detection datasets only provide key frames for training and testing. Therefore, this work uses the multiple object tracking datasets for evaluation. Note that the multiple object tracking datasets only consider movable objects such as cars and pedestrians, unlike ordinary object detection datasets.

Since the KITTI MOT dataset does not contain a validation set, this work divides the train set of the KITTI MOT dataset for training and validation. Among 21 image sequences of the train set, this work uses 16 sequences for training and the other five sequences for validation. In addition, this work supplements the small volume of the KITTI MOT train set with the KITTI object detection dataset. In the case of BDD100K MOT, this work samples one image frame every other five frames from the train set to avoid overfitting from having too many similar images.

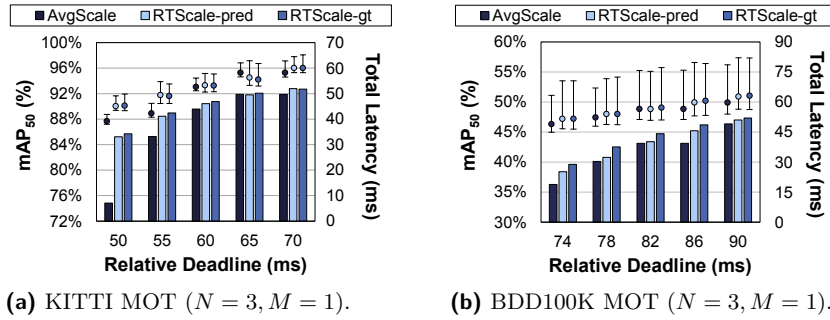
This work first trains the baseline object detector for each dataset with multiple image scales. The framework dynamically scales the input images by randomly choosing a scaling factor between $[1/1.4, 1.4]$ every 10 iterations. This work uses pretrained network parameters for the first 137 layers of the baseline object detector. This work uses 0.001 as the learning rate for the datasets and divides the rate by 10 after 80% and 90% of the total iterations as prior work [6].

After training the baseline object detector, this work trains the sensitivity inference module for each dataset. This work generates ground-truth sensitivity labels for each dataset and filters several outliers to facilitate deep learning. This work uses (0.6, 2.8) and (0.3, 4.0) as (ρ^{min}, ρ^{max}) in (3) to normalize the ground-truth sensitivity values for KITTI MOT and BDD100K MOT, respectively. For KITTI MOT, this work uses a larger learning rate considering the small volume of the KITTI MOT train set. Similar to the baseline detector, this work divides the learning rate by 10 after 80% and 90% of the total iterations.

For evaluation, this work generates three artificial image streams with the validation images of each dataset. This work divides a set of image sequences into three image streams. Each stream in KITTI MOT and BDD100K MOT has 693 and 11,329 images, respectively. For evaluating with multiple processing units, this work further divides three image streams from KITTI MOT into six image streams. Although each image stream includes real-world road images, the image streams are not from the cameras on the same vehicle. It might not

■ **Table 3** Datasets and training parameters.

Information		KITTI MOT	BDD100K MOT
Number of train images		12,900	55,616
Number of validation images		2,079	33,987
Original image size		1242×375	1280×720
Parameter		KITTI MOT	BDD100K MOT
Baseline	Number of iterations	16000	16000
	Batch size	64	64
	Learning rate	0.001	0.001
Module	Number of iterations	70000	70000
	Batch size	2	2
	Learning rate	0.001	0.0001
Image scales		$1024 \times 288, 896 \times 256,$ $768 \times 224, 640 \times 192,$ 512×160	$768 \times 416, 704 \times 384,$ $640 \times 352, 576 \times 320,$ 512×288



■ **Figure 7** Object detection accuracy and total latency with a single processing unit (N : Number of image streams, M : Number of GPUs).

be ideal, but it is the best possible option because there is no available multi-camera driving dataset. This work conducts experiments with two Intel(R) Xeon(R) Silver 4210 CPUs and up to three NVIDIA GeForce RTX 2080 Ti GPUs using CUDA 10.0 and cuDNN 7.6.4.

5.2 Results

5.2.1 Accuracy and Latency

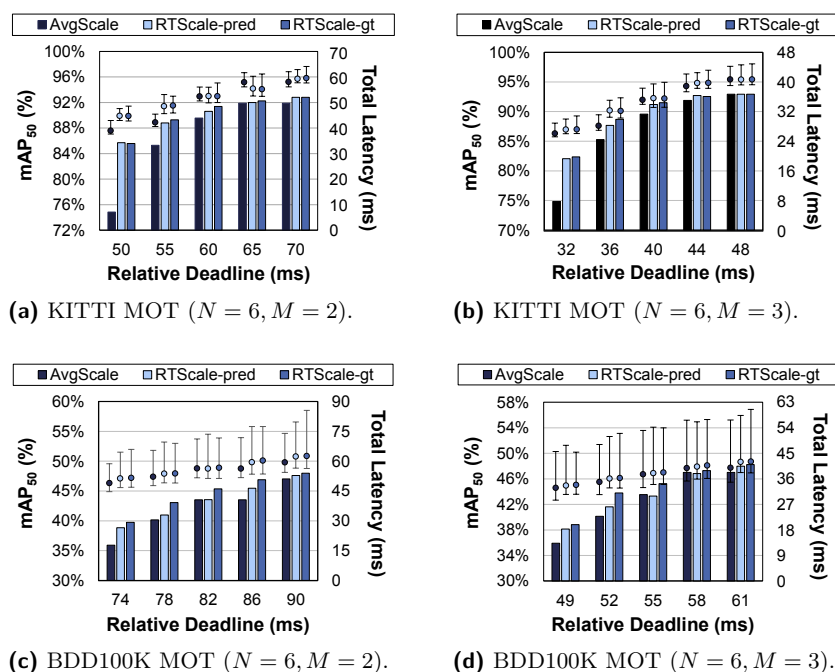
This work measures the accuracy and latency of the object detector for a given relative deadline with different image scaling schemes. This work evaluates object detection accuracy with the mean average precision metric with IoU threshold = 0.5 using the evaluation code of the Darknet framework. In addition, this work measures the total latency for processing all the images from image streams within an interval. This work determines relative deadlines considering the worst-case object detection latency of processing an image at each scale.

Figure 7 shows the accuracy and latency of the object detector for KITTI MOT and BDD100K MOT with one processing unit. In the figures, the rectangular bars indicate the accuracy of the object detector and the error bars indicate the range of object detection latency during all the intervals. For different relative deadlines, all the image scaling schemes meet the deadline constraints because they always choose the appropriate set of scales that would not violate the deadline constraint.

Overall, with the same relative deadline, RTScale-pred and RTScale-gt obtain better accuracy than AvgScale. As shown in the graphs, RTScale-pred and RTScale-gt enhance the object detection accuracy by 10.4 and 10.8 points at most. Since RTScale-pred and RTScale-gt determine the scales of the images considering scale sensitivity, RTScale-pred and RTScale-gt can reduce accuracy loss from image downsampling compared with AvgScale.

The amount of accuracy gain with RTScale tends to decrease as the relative deadline increases. It is because the accuracy gap between two similar scales tends to be smaller for the higher scales. In Figure 4a, the object detector obtains 5.48 more points for S17 with the image width of 768 than the image width of 640. On the other hand, the accuracy gap between the image widths of 1024 and 896 is only 0.83 points. The result implies that there are less opportunities to enhance accuracy through adaptive image scaling when the higher scales are available (i.e., when the relative deadline is large).

In general, RTScale-gt obtains better accuracy than RTScale-pred because RTScale-gt uses ground-truth scale sensitivities in scheduling. Therefore, RTScale-gt can better predict expected accuracy loss than RTScale-pred. In the experiment, RTScale-gt gains at most 1.8 more points in accuracy compared with RTScale-pred. Nevertheless, RTScale-pred performs almost as good as RTScale-gt in the experiments.



■ **Figure 8** Object detection accuracy and total latency with multiple processing units (N : Number of image streams, M : Number of GPUs).

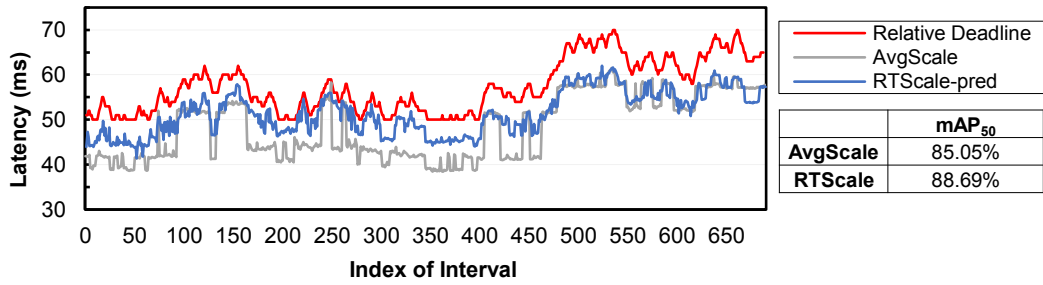
Figure 8 shows the accuracy and latency of the object detector for KITTI MOT and BDD100K MOT with two and three processing units. RTScale assigns each task to a certain processing unit regarding the scale sensitivity. For example, RTScale can have a processing unit dedicated to a highly sensitive image. Thus, RTScale can better utilize the multiple processing units to reduce accuracy loss than AvgScale. As shown in the graphs, RTScale-pred and RTScale-gt enhance the object detection accuracy by 10.9 and 10.7 points at most with the two processing units.

Interestingly, RTScale-pred outperforms RTScale-gt in some cases. It is because choosing the maximum scale is not always the best, even for the images with scale sensitivity larger than one. This work observes that detecting objects at a medium scale sometimes results in the best accuracy. Since RTScale calculates the scale sensitivity regarding the maximum and minimum scales only, RTScale sometimes fails to predict the expected accuracy loss correctly. However, predicting the non-monotonic tendency of accuracy change is too complicated for deep neural networks to learn. Therefore, RTScale only considers the monotonic tendency of accuracy change.

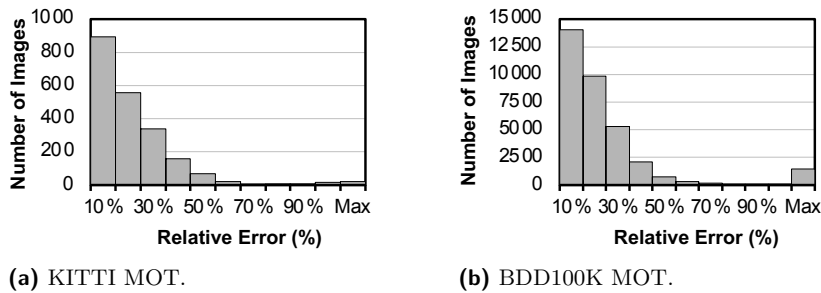
Furthermore, RTScale may not perform to the best because the datasets only provide a few frames per second. Although the object detector can process three images within 100 ms, the actual time interval between two consecutive images in the datasets is much longer. It means that two images may not be similar to each other. It can hinder RTScale from performing to the best because the scale sensitivities of the two images may not be similar to each other, which is different from our assumption.

5.2.2 Dynamic Deadline Adaptation

This work evaluates how the object detector can well adapt to dynamically changing relative deadline with the KITTI MOT dataset. This work randomly generates the sequence of relative deadlines and provides a different relative deadline to the object detector at every



■ **Figure 9** Object detection accuracy and latency when changing the relative deadline at every interval.



■ **Figure 10** Relative error distribution of scale sensitivity predictions.

interval. This work randomly chooses an integer within $[50, 70]$ for a relative deadline. Here, since each image stream of the KITTI MOT dataset contains 693 images, the total number of intervals is 693, accordingly.

Figure 9 shows how the latency of the object detector changes according to the relative deadline. In the figure, the red line indicates the relative deadline at each iteration, the grey line indicates the latency of the object detector with AvgScale, and the blue line indicates the latency of the object detector with RTScale-pred. Figure 9 also provides the accuracy of the object detector with AvgScale and RTScale-pred.

The evaluation result shows that RTScale better adapts to the deadline compared with AvgScale because RTScale applies different scales for the images. Although the two image scaling schemes enable the object detector to meet the deadline constraint at every interval, the latency with RTScale changes more smoothly according to the given deadline. Furthermore, RTScale obtains 3.64 points higher object detection accuracy compared with AvgScale. The result shows that RTScale can well adapt to the dynamic deadline while enhancing object detection accuracy.

5.2.3 Scale Sensitivity Inference

This work also evaluates how accurately the scale sensitivity inference module can predict scale sensitivity. This work calculates the relative errors of the predicted scale sensitivity with the ground-truth scale sensitivity for the validation set. Figure 10a and Figure 10b are the histograms that show the distributions of relative errors for KITTI MOT and BDD100K MOT. On average, the scale sensitivity module obtains 17% and 21% relative errors for KITTI MOT and BDD100K MOT, respectively.



■ **Figure 11** Sample object detection results with the image scales used in detection (TP: True Positive, FP: False Positive).

Figure 7 clearly shows the consequence of inaccurate scale sensitivity inference. Since the sensitivity module relatively performs worse for BDD100K MOT than KITTI MOT, the gap between RTScale-pred and RTScale-gt is larger for BDD100K MOT. While the largest gap between RTScale-pred and RTScale-gt is 0.5 points for KITTI MOT, the largest gap between RTScale-pred and RTScale-gt is 1.8 points for BDD100K MOT. It implies that the errors in scale sensitivity predictions can degrade accuracy.

However, the relative errors of scale sensitivity predictions do not directly cause accuracy degradation because the scheduler determines the scales of images by comparing the expected accuracy losses of the images. That is, the scheduler considers the relative differences in their sensitivities. Therefore, if the predicted sensitivities of the images show a similar difference to the ground-truth sensitivities, RTScale-pred still can find an effective solution as RTScale-gt. As a result, RTScale-pred could obtain a comparable accuracy with RTScale-gt in the experiments.

This work also visualizes the object detection results of sample images from the KITTI MOT to show how the quality of object detection results differs with image scales. Figure 11 shows the object detection results of the same images with different relative deadlines. In the figure, each image includes the image scale used in object detection. For readability, this work resizes the images to have the same size. In addition, the color of a bounding box indicates the category of the object. Overall, the object detection results with a high scale have more true positives than the results with a lower scale. For example, in the case of the middle image in the figure, the object detector fails to detect a hidden car on the right side at the lowest scale.

5.2.4 Memory and Scheduling Overhead

This work measures the system-level memory and scheduling overhead compared with the original object detection system. RTScale requires extra memory because the system needs to store the parameters for the scale sensitivity inference module and maintain the multiple descriptors for each convolution layer to avoid the resizing overhead. However, the total memory overhead is at most 2.10 % only compared with the amount of memory that the original system uses. Furthermore, RTScale incurs little scheduling overhead with the small N and M compared with the total latency of processing images.

■ **Table 4** Memory and scheduling overhead of RTScale.

Configuration	$N = 3, M = 1$	$N = 6, M = 2$	$N = 6, M = 3$
Memory Overhead	0.32 %	1.54 %	2.10 %
Scheduling Overhead	$< 10^{-5}$ %	$< 10^{-5}$ %	$< 10^{-5}$ %

6 Related Work

6.1 Real-time Object Detection

Previous work [22, 19, 38] has studied real-time object detection to finish object detection within a given deadline.

Jang et al. [22] design a real-time object detector for autonomous driving considering the end-to-end delay of object detection. Jang et al. observe that existing object detectors suffer from severe time lags, even excluding inference time. Jang et al. address that the time lags come from queuing delay, pipeline imbalance, and resource contention (especially on integrated CPU-GPU platforms). To reduce the end-to-end delay of an object detector, Jang et al. propose three optimization techniques: on-demand capture, zero-slack pipeline, and contention-free pipeline. Since the prior work targets to optimize end-to-end delay rather than inference time, this work can apply RTScale to R-TOD to further optimize end-to-end object detection latency, including inference time.

S³DNN [45] is a system solution for executing multiple DNN workloads in real time. S³DNN guarantees real-time performance with two main techniques: (i) system-level data fusion and (ii) supervised streaming and scheduling. First, S³DNN fuses multiple DNN workloads into one DNN instance to enhance resource utilization within the memory limit. Second, S³DNN enables streaming processing of multiple GPU kernels from different DNN instances and schedules the kernels considering concurrency benefits. The main objective of S³DNN is to enhance the throughput of DNN workloads, which might degrade pipeline efficiency in autonomous driving. Nevertheless, this work can employ its techniques to optimize throughput because the techniques are orthogonal to the approach of this work.

Heo et al. [19] propose multi-path neural networks that can adapt to time-varying time constraints by dynamically changing their execution paths. According to the time constraints, the multi-path neural network can skip layers, generate a different number of region proposals, and switch to another branch. Heo et al. also provide the worst-case execution time model for deep neural networks considering the worst-case memory contention. Although the prior work shows that the multi-path neural network can well adapt to the time constraints, it requires designing an elaborate multi-path neural network to minimize accuracy loss.

IntPred [38] is an object detection scheme that reduces computational cost with an interpolation prediction. Since objects move slowly across consecutive frames, IntPred only runs an object detector for a partial set of image frames. Based on the interpolation prediction, IntPred adjusts the location of the objects for subsequent image frames. By avoiding processing every image frame, IntPred can reduce object detection time and power consumption. However, it can be risky to skip image frames because objects can suddenly appear in the skipped frames, especially in the street images that change dynamically.

6.2 Real-time DNN Inference

Rather than focusing on object detection, other work [42, 26, 31, 27, 23, 41, 3, 4] proposes general real-time DNN frameworks.

Recent work [42, 26, 31] has proposed to use the concept of *early exiting* for real-time DNN inference, which allows a neural network to generate the output early. Yao et al. [42] suggest controlling the number of network stages with early exiting to provide intelligent real-time edge services. Liu et al. [31] extend the previous work [42] to consider the criticality of data within a scene. Kim et al. [26] propose a hierarchical neural network that can provide abstract classifications before final concrete classification. Similar to RTScale, the recent work exploits the architecture of existing networks to provide real-time inference. However, the work requires designing a network with multiple exits in advance for dynamic latency adjustment. Therefore, the network cannot easily adapt to different run-time environments.

Lee et al. [27] introduce SubFlow, which enables real-time inference and training by dynamically pruning neurons. SubFlow dynamically generates subgraphs according to dynamic time constraints. By ranking neurons of each layer considering their importance, SubFlow can find the best subgraph that satisfies a given time constraint. Lee et al. propose time-bound inference and training of convolutional and fully-connected layers. Since its dynamic model compression approach is orthogonal to adaptive image scaling, RTScale is applicable in combination with SubFlow. However, SubFlow only supports convolution and fully-connected layers for now, while most object detectors include other types of layers.

DART [41] is a pipeline-based DNN scheduling framework, which provides a deterministic response time for processing multiple DNN models. DART minimizes the response time using data-level parallelism, allocating tasks into multiple CPUs and GPUs. DART utilizes two types of data-level parallelism, inter-node pipelining and intra-node data parallelism, to overcome the resource limitation of local accelerators and exploit multiple processing units efficiently. DART provides a time-predictable DNN execution for multiple processing units.

DeepRT [23] and PredJoule [4] utilize DVFS (Dynamic Voltage-Frequency Scaling) to satisfy time constraints and optimize energy consumption in neural network execution. They dynamically change the DVFS configuration according to time constraints and energy consumption. DeepRT also employs dynamic model compression to reduce the computational cost of executing deep neural networks on mobile devices. PredJoule finds the optimal DVFS configuration considering the performance and energy characteristics of different layers.

ApNet [3] is a timing-predictable runtime system for DNN workloads, which applies approximation to neural networks to satisfy real-time requirements. ApNet chooses an appropriate approximation strategy for each layer based on how the resource utilization of the target device changes with different approximation strategies. In addition, ApNet designs a runtime system that can enhance resource sharing and concurrency via approximation.

Although the existing approaches [23, 4, 3] for real-time DNN inference allow each inference task to finish within a deadline, the approaches either require the system support or incur a relatively large accuracy loss.

7 Conclusion

This work proposes RTScale, which enables real-time object detection through adaptive image scaling while minimizing accuracy loss. Based on the observation that each image has a different sensitivity to image scaling with regard to object detection accuracy, RTScale finds appropriate scales for images from multiple streams considering both scale sensitivity and real-time constraint. RTScale enables existing object detectors to infer the sensitivity by adding a few layers for sensitivity inference. This work evaluates RTScale with other image scaling schemes with two popular driving datasets. The evaluation results show that RTScale can meet real-time constraints with minimal accuracy loss.


References

- 1 Apollo. <https://apollo.auto/index.html>.
- 2 Autoware.ai Core Perception Github Repository. https://github.com/Autoware-AI/core_perception, May 2021.
- 3 Soroush Bateni and Cong Liu. ApNet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018. doi:10.1109/RTSS.2018.00017.
- 4 Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. PredJoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, December 2018. doi:10.1109/RTSS.2018.00020.
- 5 Adam Betts and Alastair Donaldson. Estimating the wcet of gpu-accelerated applications using hybrid analysis. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013. doi:10.1109/ECRTS.2013.29.
- 6 Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal speed and accuracy of object detection, 2020. doi:10.48550/arXiv.2004.10934.
- 7 Ting-Wu Chin, Ruizhou Ding, and Diana Marculescu. AdaScale: Towards real-time video object detection using adaptive scaling. In *Proceedings of Machine Learning and Systems 2019*, pages 431–441. 2019.
- 8 Ting-Wu Chin, Chia-Lin Yu, Matthew Halpern, Hasan Genc, Shiao-Li Tsao, and Vijay Janapa Reddi. Domain-specific approximation for object detection. *IEEE Micro*, 38(1):31–40, 2018. doi:10.1109/MM.2018.112130335.
- 9 Hiroyuki Chishiro, Kazutoshi Suito, Tsutomu Ito, Seiya Maeda, Takuya Azumi, Kenji Funaoka, and Shinpei Kato. Towards heterogeneous computing platforms for autonomous driving. In *2019 IEEE International Conference on Embedded Software and Systems (ICCESS)*, 2019. doi:10.1109/ICCESS.2019.8782446.
- 10 Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. In *Proceedings of the 5th International Conference on Learning Representations, ICLR '17*, 2017.
- 11 Darknet. <https://github.com/AlexeyAB/darknet>.
- 12 Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012. doi:10.1109/CVPR.2012.6248074.
- 13 Ross Girshick. Fast R-CNN. *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015. doi:10.1109/ICCV.2015.169.
- 14 Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014. doi:10.1109/CVPR.2014.81.
- 15 Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15*. JMLR.org, 2015.
- 16 Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, NIPS'15*, Cambridge, MA, USA, 2015. MIT Press.
- 17 Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask R-CNN. *2017 IEEE International Conference on Computer Vision (ICCV)*, October 2017. doi:10.1109/ICCV.2017.322.
- 18 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. doi:10.1109/CVPR.2016.90.
- 19 Seonyeong Heo, Sungjun Cho, Youngsok Kim, and Hanjun Kim. Real-time object detection system with multi-path neural networks. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020. doi:10.1109/RTAS48715.2020.000-8.

- 20 Vesa Hirvisalo. On static timing analysis of gpu kernels. In *14th International Workshop on Worst-Case Execution Time Analysis*, OpenAccess Series in Informatics (OASISs), 2014.
- 21 Yijie Huangfu and Wei Zhang. Static wcet analysis of gpus with predictable warp scheduling. In *2017 IEEE International Symposium on Real-Time Computing (ISORC)*, 2017. doi:10.1109/ISORC.2017.24.
- 22 Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil Dutt, and Jong-Chan Kim. R-TOD: Real-time object detector with minimized end-to-end delay for autonomous driving. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 191–204, 2020. doi:10.1109/RTSS49844.2020.00027.
- 23 Woochul Kang and Jaeyong Chung. DeepRT: Predictable deep learning inference for cyber-physical systems. *Real-Time Systems*, 55(1):106–135, January 2019. doi:10.1007/s11241-018-9314-y.
- 24 Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015. doi:10.1109/MM.2015.133.
- 25 Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS '18*, pages 287–296. IEEE Press, 2018. doi:10.1109/ICCPS.2018.00035.
- 26 Jung-Eun Kim, Richard Bradford, Man-Ki Yoon, and Zhong Shao. ABC: Abstract prediction before concreteness. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1103–1108, 2020. doi:10.23919/DATE48585.2020.9116479.
- 27 Seulki Lee and Shahriar Nirjon. SubFlow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020. doi:10.1109/RTAS48715.2020.00-20.
- 28 Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *Proceedings of the 5th International Conference on Learning Representations, ICLR '17*, 2017.
- 29 Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*. Association for Computing Machinery, 2018. doi:10.1145/3173162.3173191.
- 30 Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. doi:10.1109/CVPR.2017.106.
- 31 Shengzhong Liu, Shuochao Yao, Xinzhe Fu, Huajie Shao, Rohan Tabish, Simon Yu, Ayoosh Bansal, Heechul Yun, Lui Sha, and Tarek Abdelzaher. Real-time task scheduling for machine perception in intelligent cyber-physical systems. *IEEE Transactions on Computers*, 2021. doi:10.1109/TC.2021.3106496.
- 32 Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*. Springer International Publishing, 2016. doi:10.1007/978-3-319-46448-0_2.
- 33 Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.
- 34 Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016. doi:10.1109/CVPR.2016.91.

- 35 Joseph Redmon and Ali Farhadi. YOLO9000: Better, faster, stronger. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017. doi:10.1109/CVPR.2017.690.
- 36 Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6), 2017. doi:10.1109/TPAMI.2016.2577031.
- 37 Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. doi:10.1109/CVPR.2015.7298594.
- 38 Hamid Tabani, Matteo Fusi, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. IntPred: Flexible, fast, and accurate object detection for autonomous driving systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20*. Association for Computing Machinery, 2020. doi:10.1145/3341105.3373918.
- 39 Tesla Model S Owners Manual. https://www.tesla.com/sites/default/files/model_s_owners_manual_north_america_en_us.pdf, April 2020.
- 40 Vijay V. Vazirani. *Approximation Algorithms*. Springer Publishing Company, Incorporated, 2010. doi:10.1007/978-3-662-04565-7.
- 41 Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405, 2019. doi:10.1109/RTSS46320.2019.00042.
- 42 Shuochao Yao, Yifan Hao, Yiran Zhao, Huajie Shao, Dongxin Liu, Shengzhong Liu, Tianshi Wang, Jinyang Li, and Tarek Abdelzaher. Scheduling real-time deep learning services as imprecise computations. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2020. doi:10.1109/RTCSA50079.2020.9203676.
- 43 Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving dataset for heterogeneous multitask learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. doi:10.1109/CVPR42600.2020.00271.
- 44 Xiaofan Zhang, Haoming Lu, Cong Hao, Jiachen Li, Bowen Cheng, Yuhong Li, Kyle Rupnow, Jinjun Xiong, Thomas Huang, Honghui Shi, Wen-Mei Hwu, and Deming Chen. SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. In *Proceedings of Machine Learning and Systems 2020*, pages 216–229. 2020.
- 45 Husheng Zhou, Soroush Bateni, and Cong Liu. S³DNN: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201, 2018. doi:10.1109/RTAS.2018.00028.
- 46 Menglong Zhu and Mason Liu. Mobile video object detection with temporally-aware feature maps. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5686–5695, 2018. doi:10.1109/CVPR.2018.00596.
- 47 Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. Towards high performance video object detection. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7210–7218, 2018. doi:10.1109/CVPR.2018.00753.
- 48 Xizhou Zhu, Yujie Wang, Jifeng Dai, Lu Yuan, and Yichen Wei. Flow-guided feature aggregation for video object detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 408–417, 2017. doi:10.1109/ICCV.2017.52.
- 49 Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4141–4150, 2017. doi:10.1109/CVPR.2017.441.

ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems

Iryna De Albuquerque Silva ✉ 

ONERA, Toulouse, France

Thomas Carle ✉ 

IRIT – Univ Toulouse 3 – CNRS, France

Adrien Gauffriau ✉

Airbus, Toulouse, France

Claire Pagetti ✉ 

ONERA, Toulouse, France

Abstract

Machine learning applications have been gaining considerable attention in the field of safety-critical systems. Nonetheless, there is up to now no accepted development process that reaches classical safety confidence levels. This is the reason why we have developed a generic programming framework called ACETONE that is compliant with safety objectives (including traceability and WCET computation) for machine learning. More practically, the framework generates C code from a detailed description of off-line trained feed-forward deep neural networks that preserves the semantics of the original trained model and for which the WCET can be assessed with OTAWA. We have compared our results with Keras2c and uTVM with static runtime on a realistic set of benchmarks.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Software notations and tools

Keywords and phrases Real-time safety-critical systems, Worst Case Execution Time analysis, Artificial Neural Networks implementation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.3

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.8.1.6>

Funding This project received funding from the French “Investing for the Future – PIA3” program within the Artificial and Natural Intelligence Toulouse Institute (ANITI).

1 Introduction

The use of artificial intelligence approaches is already of vital importance in many research areas. In particular, when embedded in aircraft systems, intelligent algorithms could help in tasks such as navigation, predictive maintenance and air traffic control, improving safety and saving environmental resources. Nonetheless, not much progress has been made in embedding machine learning solutions in safety-critical systems as most of those applications do not reach classical safety confidence levels and are not implemented with accepted development process [2, 5]. The scope of this work is the safe real-time implementation of neural networks on embedded platforms.

Context. We focus on safety-critical domains and in particular on aeronautics that is subject to *certification*. The question of how to safely and reliably implement a neural network on an adequate hardware is of vital importance. Indeed, certification requirements, in particular those of the DO 178-C [14]¹, impose strong guarantees on the quality of the code and expect the designer to:

¹ Classical guidance for the implementation process of the software items



3:2 Predictable Code for Machine Learning Applications

- ensure *traceability* between the requirements and the (source) code;
- compute the *WCET (Worst Case Execution Time)* [39] for each piece of code;
- run *intensive testing* to verify the compliance of the implementation to the requirements.

This includes unit tests to verify both that the executable provides the intended function and there is no hidden unintended function (by activating all the branches of the code). The purpose is thus to provide a programming framework compliant with these objectives for machine learning. This work is restricted to *off-line trained feed-forward deep neural networks* (referred to simply as neural networks or DNN subsequently). The off-line design of such neural networks is done by defining the structure (that is the number and the type of layers), choosing the training data set and using a learning framework such as Tensorflow [1] or PyTorch [28]. The result of the design is called the *inference model*: it comprises a neural network with its parameters (e.g. weights, biases, activation functions or kernels). The implementation – the part we focus on – consists in coding the inference model in an adequate programming language and porting the code on the target hardware.

Contributions. The first challenge brought by the implementation is the *semantic preservation*: the reproducibility of the behavior observed when executing the *inference model* within the training tool on the target hardware. Thus our first contribution (see section 2) is to formally define the semantics of DNN (by extending and formalizing existing works of the literature) and explaining the challenges brought by the current frameworks. Indeed, the training tools such as Tensorflow/Keras or PyTorch do not encode the basic operations, such as convolutions (and thus matrices operations) in the same manner.

The second challenge is *predictability*: the capacity to assess the worst-case execution time (WCET) of a sequential code. In the ML literature, most of the implementations are done on GPUs or TPUs with a runtime engine such as Tensorflow that interprets the *computation graph*, i.e., a graph describing the mathematical structure of the neural network. Such an interpreter uses dynamic memory and scheduling allocation and as we focus on safety-critical domain – and more specifically avionics –, such an approach is not practical for two reasons. First, the hardware targets that are compatible with certification are not those mentioned earlier. We thus focus on general purpose multi-core commercial off-the-shelf (COTS) hardware such as the T1042 from NXP, the Coolidge from Kalray [18] or the Keystone from Texas Instrument [37] (used in the experiments). Second, the programs, including the application, the RTOS and the runtime, must be *predictable*. There are some initiatives to make such runtime predictable such as eIQ and KaNN. However, there is still a large amount of work and proof to show the capability to compute a WCET for these tools. This is the reason why we target a more classical static approach which consists in generating an equivalent C code to execute the model (no interpretation) such as proposed in [8]. Our second contribution is the development of ACETONE (Avionics C code generator for Neural Networks), a framework that generates a real-time C code *semantically equivalent* to the inference model (see section 3) and that fits the aeronautic requirements. We made a particular effort on the software architecture to make the framework:

- *modular*: it is very simple to add new DNN structures, new types of layers or new refinement of the existing ones.
- *very easy to use*: a person non familiar with our framework can very quickly generate their C code and port them on their target;
- *extremely traceable*: looking at the generated C code, it is humanly possible to trace back to the original exported DNN model, which is an expected property from the DO-178C;

- *predictable*: we used a static WCET analyzer of the literature, OTAWA [4] developed at the University of Toulouse, to assess the WCET of the code. This means that the C code is expected to run sequentially on a single core (no parallelization targeted in this paper), all the memory allocations are static and the schedule (here the sequence of executions) is also static. The compilation of the C code to a binary must also use the flag `-O0` (no compiler optimization). These restrictions are important to keep in mind to understand the philosophy of the code generation.

The last contribution is a thorough evaluation of our framework together with a comparison with state of the art C code generator frameworks, namely Keras2c [10] and uTVM with static C runtime [35]. Section 4 details the methodology: we have selected a set of representative benchmarks and identified a set of criteria to assess the quality of a code (in accordance with the DO-178C objectives listed above). Section 5 gives the results of the experiments. We were able to assess most of our criteria for all the benchmarks and frameworks. In particular, we have ported the binary on an Arm Cortex-A15 of the Keystone [37] to compare the measured and worst-case execution times. Overall, in terms of performance, we are comparable to and even slightly better than the other frameworks. This stems, for uTVM, from the restrictions needed for predictability and the compilation with `-O0`. In that sense, our implementations are optimal with respect to our criteria.

2 Reminder on Deep Neural Networks

We focus on the inference of off-line trained feed-forward Deep Neural Networks (DNN). More precisely, we consider convolutional neural networks (CNN) and multi-perceptron (or fully-connected) neural networks.

2.1 Functions performed by DNN

There are multiple ways to define DNNs: directed graphs, computational graphs or simply the mathematical functions transforming the input into the output. The latter is the way we propose to explain the computations needed to be done by the C code. The input of those functions can be seen as a multi-dimensional vector also called *tensor*. Subsequently, we will only consider 1D-, 2D- and 3D-tensors but to save space, we only provide definitions for 3D. We only consider inference with one input (no batch).

► **Definition 1 (Tensor)**. A 3D-tensor T is represented by its size (n_h, n_w, n_c) where n_h is the height, n_w the width and n_c the number of channels (or feature maps). We denote by T_{x_1, x_2, x_3} the value of T for the indices x_1, x_2, x_3 . We denote by $T[s_{11} : s_{21}, \dots, s_{1k} : s_{2k}]$ the slice of T of all the values $T_{s_{11}+x_1, \dots, s_{1k}+x_k}$ with $i \in [1, k]$ and $x_i \in [1, s_{2i} - s_{1i}]$.

► **Definition 2 (Feed-forward Deep Neural Network)**. A feed-forward neural network $N = \langle l_1, \dots, l_n \rangle$ is a succession of layers l_i taking as input the output of the previous layer l_{i-1} . The first layer takes the input tensor. A layer can be of type $l \in \{\text{act, bias, padd, conv, pool, batch norm, flat, dense}\}$ where *act* is an activation, *padd* is a padding, *bias* is a bias adding, *conv* is a convolution, *pool* is a pooling, *batch norm* is normalization, *flat* is flattening and *dense* is a perceptron. A layer comes with a set of parameters (e.g. weights or stride).

► **Definition 3 (Function associated to a DNN)**. The function f_N computed by a DNN $N = \langle l_1, \dots, l_n \rangle$ is the composition of the functions computed by each layer $f_N = f_{l_n} \circ \dots \circ f_{l_1}$.

The semantics of each function is given in [38]. We give the definition, with mathematical equations, of the main layers used in the use cases depicted in section 4.1.

► **Definition 4** (Activation function). Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function (e.g. ReLu, sigmoid). The activation function \mathcal{A}_f applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{A}_f(I)$ of size (n_h, n_w, n_c) defined by $O_{x,y,z} = f(I_{x,y,z})$ for all $x \leq n_h$, $y \leq n_w$ and $z \leq n_c$. We could also write $O_{x,y,z} = \text{map}(I, f)$.

► **Definition 5** (Bias layer associated function). Let B be a 3D-tensor of size (n_h, n_w, n_c) . The bias function \mathcal{B}_B applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{B}_B(I)$ of size (n_h, n_w, n_c) defined by $O_{x,y,z} = I_{x,y,z} + B_{x,y,z}$ for all $x \leq n_h$, $y \leq n_w$ and $z \leq n_c$.

► **Definition 6** (Padding layer associated function). Let $p = (p_t, p_b, p_l, p_r)$ be a 4-tuple of integers representing the padding to be applied on each border of a 3D-tensor. The padding function \mathcal{P}_p applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}_p(I)$ of size (o_h, o_w, o_c) with $o_h = n_h + p_t + p_b$, $o_w = n_w + p_l + p_r$ and $o_c = n_c$ such that

$$O_{x,y,z} = \begin{cases} 0 & \text{if } x \leq p_t \text{ or } x > n_h + p_t \text{ or } y \leq p_l \text{ or } y > n_w + p_l \\ I_{x-p_t, y-p_l, z} & \text{otherwise} \end{cases}$$

► **Definition 7** (2D-convolution associated function). Let K be a vector of 3D-tensors $[K^1, K^2, \dots, K^{\text{nb_kernel}}]$ representing the kernels of the convolution. Each kernel K^i is of size (f_h, f_w, f_c) . Let $s = (s_h, s_w)$ be the stride parameter with s_h and s_w two integers. The 2D-convolution² $\mathcal{C}_{K,s}$ applied to a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{C}_{K,s}(I)$ of size (o_h, o_w, o_c) with $o_h = \lfloor \frac{n_h - f_h}{s_h} + 1 \rfloor$, $o_w = \lfloor \frac{n_w - f_w}{s_w} + 1 \rfloor$ and $o_c = \text{nb_kernel}$. We have $O_{x,y,z} = \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} \sum_{m=1}^{f_c} K_{i,j,m}^z \cdot I_{s_h \cdot (x-1) + i, s_w \cdot (y-1) + j, m}$ for all $x \leq o_h$, $y \leq o_w$ and $z \leq o_c$. Note that also we must have $f_c = n_c$ thus, convolutions are often applied on 3D-tensors on which padding has been applied first to fit the sizes. See definition 14.

► **Definition 8** (Pooling layer associated function). Let $s = (s_h, s_w)$ be the stride parameters, let $k = (k_h, k_w)$ be the height and width of the window and let $f : \mathbb{R}^{k_h \cdot k_w} \rightarrow \mathbb{R}$ be a function (e.g. max or average). The pooling applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}\text{ool}_{k,s,f}(I)$ of size (o_h, o_w, o_c) with $o_h = \lfloor \frac{n_h - k_h}{s_h} + 1 \rfloor$, $o_w = \lfloor \frac{n_w - k_w}{s_w} + 1 \rfloor$ and $o_c = n_c$ with $O_{x,y,z} = f(I[s_h \cdot (x-1) + 1 : s_h \cdot (x-1) + k_h + 1][s_w \cdot (y-1) + 1 : s_w \cdot (y-1) + k_w + 1][z])$.

► **Definition 9** (Batch norm layer associated function). Let γ be 1D-tensor of size n_c be the scale, let β be 1D-tensor of size n_c be the offset, let μ be 1D-tensor of size n_c be the mean (on the batch fixed during the training), let V be 1D-tensor of size n_c be the variance (on the batch fixed during the training), let ϵ be a float used to ensure no division per 0. The batch norm applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{B}\mathcal{N}_{\gamma,\beta,\mu,\sigma,\epsilon}(I)$ of size (n_h, n_w, n_c) with $O_{x,y,z} = \frac{\gamma_z}{\sqrt{V_z + \epsilon}} \cdot I_{x,y,z} + \left(\beta_z - \frac{\mu_z}{\sqrt{V_z + \epsilon}} \right)$. We often denote by $\alpha_z = \frac{\gamma_z}{\sqrt{V_z + \epsilon}}$ and $\mathcal{B}_z = \left(\beta_z - \frac{\mu_z \cdot \gamma_z}{\sqrt{V_z + \epsilon}} \right)$, so that $O_{x,y,z} = \alpha_z \cdot I_{x,y,z} + \mathcal{B}_z$.

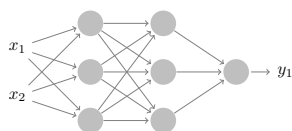
► **Definition 10** (Flattening layer). The flattening layer applied to a 3D-tensor I of size (n_h, n_w, n_c) outputs the 1D-tensor $O = \mathcal{F}\text{lat}(I)$ of size $n_o = n_h \times n_w \times n_c$ such that $O_x = I_{x \bmod n_w, \lfloor \frac{x \bmod (n_h \cdot n_w)}{n_w} \rfloor, \lfloor \frac{x}{n_h \cdot n_w} \rfloor}$.

► **Definition 11** (Dense layer). Let W be a 2D-tensor of size (n_o, n_i) (for the weights) and B be a 1D-tensor of size n_o (for the biases). The dense layer applied to a 1D-tensor I of size n_i outputs the 1D-tensor of size n_o $O = \mathcal{D}\text{ense}(I) = W \cdot I + B$, i.e. $O_x = \sum_{k=1}^{n_i} W_{x,k} \cdot I_k + B_x$.

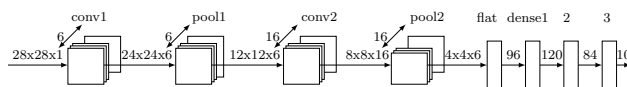
² There may be an additional parameter, that is the *dilatation* supported by the code generation and not detailed here.

Then, we can define easily the function associated to a DNN from those basic functions.

► **Example 12** (Multi-perceptron / fully-connected neural network). A *fully-connected neural network* is a succession of *dense and activation layers*. The function associated to the DNN of figure 1 is $N = f_{l_3} \circ f_{l_2} \circ f_{l_1} = \mathcal{A}_3(W_3 \cdot (\mathcal{A}_2(W_2 \cdot (\mathcal{A}_1(W_1 \cdot I + B_1)) + B_2)) + B_3)$. Its structure corresponds to 2 hidden layers with 3 neurons each, 2 inputs and 1 output. Short notation: (2, 3, 3).



■ **Figure 1** Fully-connected NN.



■ **Figure 2** LeNet-5 CNN.

► **Example 13** (LeNet-5). The LeNet-5 [23] model is the basic CNN developed for handwritten digits images recognition. We used the pre-trained LeNet-5 from Keras which is shown in figure 2. Such a graphical representation is classical to highlight the layers and the number of feature maps.

The size of the input / output tensors are shown on the figure. The first 2D-convolution *conv1* takes inputs of size $28 \times 28 \times 1$, is composed of 6 kernels K^i of size $5 \times 5 \times 1$ and of a stride $s = (1, 1)$. The activation function *tanh* is applied to the outputs. The first pooling layer *pool1* is an average pooling with stride $s = (2, 2)$ and window $k = (2, 2)$. The second 2D-convolution *conv2* is composed of 16 kernels K^i of size $5 \times 5 \times 6$ and of a stride $s = (1, 1)$. The activation function *tanh* is applied to the outputs. The second pooling layer *pool2* is an average pooling with stride $s = (2, 2)$ and window $k = (2, 2)$. The 3D-tensor of size $6 \times 6 \times 4$ is flattened in a 1D-tensor of size 96. There are three dense layers with respectively $(n_i, n_o) = (96, 120)$, $(n_i, n_o) = (120, 84)$ and $(n_i, n_o) = (84, 10)$. The two first dense layers apply the activation function *tanh* and the last one a *softmax*. Thus the function associated to this LeNet-5 is: $N = \mathcal{A}_{softmax} \circ f_{dense3} \circ \mathcal{A}_{tanh} \circ f_{dense2} \circ \mathcal{A}_{tanh} \circ f_{dense1} \circ f_{flat} \circ f_{pool2} \circ \mathcal{A}_{tanh} \circ f_{conv2} \circ f_{pool1} \circ \mathcal{A}_{tanh} \circ f_{conv1}$.

2.2 Semantics-preserving model transformation

At this stage, it is acceptable to transform the DNN model as long as the semantics is preserved. This can be interesting when it yields an improvement of the implementation. We list here some transformations worth to be made before coding.

► **Definition 14** (Extended 2D-convolution layers). *In the literature, convolutions usually integrate other parameters than those listed in definition 7. Indeed, a convolution is often defined together with the padding, the activation function and even in some cases with a bias. We thus denote by $\mathcal{C}_{p,K,s,B,f} = \mathcal{A}_f \circ \mathcal{B}_B \circ \mathcal{C}_{K,s} \circ \mathcal{P}_p$ (all combinations by removing a function work). Note that this is common to consider bias B in convolution where $B_{x_1,y_1,z} = B_{x_2,y_2,z}$ with $x_1 \neq x_2$ and $y_1 \neq y_2$.*

► **Property 1** (Well-balanced 2D-convolution layers). *It is usual to have the output height and width equal to the input height and width, i.e. $o_h = n_h$ and $o_w = n_w$. In that case, we must have $n_h = \lfloor \frac{n_h + p_t + p_b - f_h}{s_h} + 1 \rfloor$ and $n_w = \lfloor \frac{n_w + p_l + p_r - f_w}{s_w} + 1 \rfloor$. The padding should also satisfy $p_t + p_b - f_h = -1$ and $p_l + p_r - f_w = -1$.*

► **Property 2** (Portability issue between training frameworks). *We remark that for a given kernel size, several solutions may exist to the equations of property 1. For instance, with a kernel size of (5, 5) and stride of 1, 4 different paddings for each dimension satisfy the equations. Thus classical frameworks like Tensorflow or PyTorch have different strategies (thus imply different semantics) when implementing a convolution that preserves the input size for height and width.*

► **Property 3** (Max pooling and ReLu activation layers). *Applying a ReLu activation layer before a max pooling layer is semantically equivalent to applying the ReLu activation layer after the max pooling layer. However, the number of operations is reduced when applying the ReLu activation after if the stride $s = (s_h, s_w)$ of the pooling satisfies $s_h > 1$ or $s_w > 1$.*

Proof. Let us assume that the input tensor I is of size (n_h, n_w, n_c) and that we do the ReLu before the pooling. Then we will do $O_{x,y,z} = \max(\text{ReLu}(I[s_h \cdot (x-1) + 1 : s_h \cdot (x-1) + k_h + 1][s_w \cdot (y-1) + 1 : s_w \cdot (y-1) + k_w + 1][z])))$ thus ReLu will be applied $n_h \times n_w \times n_c$ times. On the contrary, if the ReLu is done after the pooling, we will do $O_{x,y,z} = \text{ReLu}(\max(I[s_h \cdot (x-1) + 1 : s_h \cdot (x-1) + k_h + 1][s_w \cdot (y-1) + 1 : s_w \cdot (y-1) + k_w + 1][z])))$ thus the ReLu will be applied $o_h \times o_w \times o_c$ times. Note also that $\max(\max(x_i), 0) = \max(\max((x_i, 0))$, thus the semantics is preserved. ◀

► **Property 4** (Merging a batch norm with a convolution). *Applying a batch norm layer after a convolution layer is semantically equivalent to applying a single convolution with modified kernels and bias. This reduces the number of operations and saves memory bandwidth required for storing intermediate tensors.*

Proof. Let suppose that the input tensor I is of size (n_h, n_w, n_c) and that we have a convolution layer $C_{p,K,s,B,f}$ followed by a batch-norm layer $\mathcal{BN}_{\alpha,B}$. The output tensor is $O = \mathcal{BN}_{\alpha,B}(C_{p,K,s,B,f}(I))$.

$$\begin{aligned} O_{x,y,z} &= f\left(\alpha_z \cdot \left(\sum_{i=1}^{f_h} \sum_{j=1}^{f_w} \sum_{m=1}^{f_c} K_{i,j,m}^z \cdot I_{s_h \cdot (x-1) + i, s_h \cdot (y-1) + j, m} + B_{x,y,z}\right) + \mathcal{B}_z\right) \\ &= f\left(\sum_{i=1}^{f_h} \sum_{j=1}^{f_w} \sum_{m=1}^{f_c} \alpha_z \cdot K_{i,j,m}^z \cdot I_{s_h \cdot (x-1) + i, s_h \cdot (y-1) + j, m} + \alpha \cdot B_{x,y,z} + \mathcal{B}_z\right) \end{aligned}$$

This is the equation of a convolution $C'_{p,\alpha,K,s,\alpha.B+\mathcal{B},f}$ ◀

2.3 Model description for the code generation

Once a model has been trained, validated and possibly optimized with semantics-preserving transformations, its detailed description can be exported from the learning framework. As we want to generate the inference associated code, we assume the DNN representation to be cleaned from any irrelevant training-related feature (e.g. loss). A first challenge brought by the implementation is the *semantic preservation*: the reproducibility of the behavior observed at the end of the design when executing the *inference model* within the training tool and on the hardware target. Even though the semantics is clear in the literature, the training tools do not encode the (default) operations³ in the same manner. This is particularly true for convolutions, where some implementations start from the top left and some from the bottom right of the matrix, or compute the padding in a different way. This has been observed in [24] and could be reproduced by experimenting with the frameworks. There are lots of works tackling the interoperability among frameworks, by proposing conversion tools [24] or

³ when not specifying in detail the parameters, which could be very tricky

defining open source formats such as protobuf [15], Onnx [3] or Nnef [38] (Neural Network Exchange Format). A description must contain all the necessary information to encode the same behaviour: this includes the number of layers, the type of every layer, the parameters of each layer including the activation function specification and anything required to reproduce the behaviour. So far, Nnef is the most adequate format as it contains the necessary elements to reconstruct most of the semantics of a model. We currently use a degraded version of Nnef in Json to allow full text description (and not binary) to help the debugging but as future work we will comply with the Nnef.

3 C back-end

We have developed a Python prototype to generate C code. We do not detail the front-end which first imports the json description file, focusing instead on the back-end. We reuse the semantics of definition 3 considering every layer as an independent programming function for the code generation. The *forward-pass* for inference then consists in calling each layer function in the correct order with the accurate parameters and inputs.

3.1 Software architecture

The C back-end is composed of a library of functions and other model-dependent files. This library is, to a certain extent, hard-coded as the bodies of functions needed for inference are defined in the Python prototype and the corresponding C files will be generated whenever needed. The model-dependent files refer to the weights, biases and auxiliary parameters that are also written as C files.

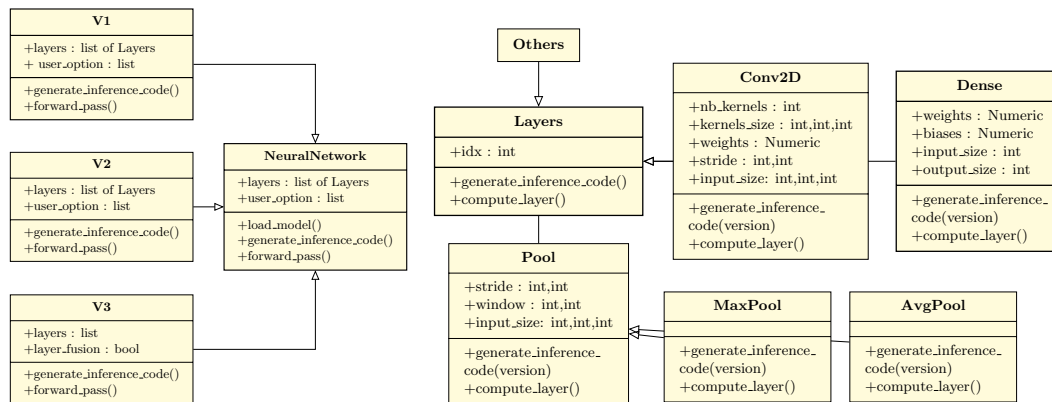


Figure 3 Software architecture

Figure 4 Software architecture of *layers*.

– several versions.

Figure 3 shows the software architecture as an Uml diagram. There are several compilation strategies named V1, V2 and V3. We decided to proceed like that in order to allow a design space exploration (DSE): our goal was to understand what is the most suited approach for a given model and hardware. The main class *NeuralNetworks* contains two variables: *layers* that contains the list of *Layers* (another class defined hereafter) and *user_option* that captures the options chosen by the user for the generation, such as applying semantics-preserving transformations or selecting the version. That class defines three methods (in addition to the classical *init*): *load_model* which imports the json DNN description; *forward_pass*

that concatenates the layers to encode the DNN function as the composition of layers and *generate_inference_code* which generates the C code. All the classes V1, V2 and V3 inherit from the *NeuralNetworks* class.

Figure 4 shows the *Layers* class which is inherited by several sub-classes, one per type of layer. The main class stores the *idx* of the layer and basically defines two abstract methods. The first is *generate_inference_code*, that implements the semantics of the layer in C language, and the second is *compute_layer*, that actually executes the functions of the layer, mainly for debugging and evaluation.

For each type of layer, we define the parameters (e.g. the weights and biases for *dense*) as variables and the methods (*generate_inference_code* and *compute_layer*) are refined. We did not detail all the layers (*others* grouping the missing ones).

The prototype supports all the layers defined in section 2.1 and the *ReLU*, *Hyperbolic Tangent*, *Sigmoid* and *Linear* activation functions.

3.2 Version 1 – generic inference function

A layer is defined as a data type, a *struct* statement, whose fields encode the parameters (e.g., type or input size). Every layer has the same definition and their particular parameters will be defined as constants in a header file. The first hidden layer of the fully-connected neural network of example 12 is a *Dense* layer depicted in listing 1.

■ **Listing 1** A Dense layer – see Definition 11.

```
double biases_Dense_01[3] = {0.07543805986642838, 0.025200579315423965, 0.03704497963190079};
...
struct Layer net[nb_layers]{
  [1] = {
    .layer_type = Dense,
    .layer_size = ll_size, // output_size
    .weights = weights_Dense_01,
    .biases = biases_Dense_01,
    .actv_function = relu,
    .pad = 0x0,
    ... }, ... };
```

layer_type and *actv_function* are function pointers to the aforementioned functions of the C library. The other fields are pointers, mostly to arrays which are also written to C source files. In the example, the *biases* field points to a static double array of size 3 shown in the listing. Unnecessary fields point to null. The whole network is treated as an array, an indexed linear sequence, of these structures.

Afterwards, an *inference* function is defined (see listing 2). It is a generic function, i.e. identical for every DNN (whether fully-connected or not), responsible for connecting the layers. It simply consists in 2 nested *for* loops (one ranging over the number of layers and the second ranging over the number of operations to be done for the current layer).

■ **Listing 2** Inference – see Definition 3.

```
for (int i=1; i < nb_layers; ++i) {
  net[i].layer_type(i, output_pre, output_cur);
  for (int j = 0; j < net[i].layer_size; ++j){
    output_pre[j] = output_cur[j]; } }
```

This logic of having generic definitions for the layers functions leads to a helpful simplicity in terms of execution and code size. However, using function pointers leads the WCET analysis tool to consider that each call made in the loop is a call to the most expensive function (or to the same function in worst context), which can be very pessimistic.

3.3 Version 2 – inlined inference function

The second version keeps the definition and declaration of layers as was done for Version 1. What changed is the *inference function* which is optimized by in-lining the programming functions for layers and activations, i.e., directly writing their body to the C file. The only parameters stored in a header C file are the weights and biases, since loops bounds are now hard-coded, meaning that the inference function is no longer generic. The Listing 3 gives part of the *inference function* for the first *dense* layer of example 12.

■ **Listing 3** Dense layer in-lined code of the inference function.

```
for (int i = 0; i < 3; ++i) { // Dense_1
  dotproduct = 0;
  for (int j = 0; j < 2; ++j) {
    dotproduct += output_pre[j] * weights_Dense_01[(i + 3*j)];
    dotproduct += biases_Dense_01[i];
    output_cur[i] = dotproduct > 0 ? dotproduct : 0; } ...
```

The straightforward effect of this optimization is improving time performance since we eliminate the function-call and struct parsing overheads, however it comes at the cost of using more instruction space, as we duplicate code, producing larger source files, which can be prohibitive in an embedded environment. Nonetheless, OTAWA produces more precise estimation for the WCET since we are able to provide the correct context in which layers are executed with no overestimation for loop bounds.

3.4 Version 3 – unrolled inference function

The third version is completely different and we reuse a philosophy of full in-lining (with loop unrolling) that can be seen *à la Scade* [9]. In particular, there is no declaration of layers and parameters as was done in listing 1. Listing 4 presents the beginning of the instructions to deal with the first *dense* layer of example 12.

■ **Listing 4** Dense layer code with in-lining and loop-unrolling.

```
dotproduct = 0; // Dense_1
dotproduct += nn_input[0] * -1.0743303298950195;
dotproduct += nn_input[1] * 0.8140403032302856;
dotproduct += 0.07543805986642838;
output_cur[0] = dotproduct > 0 ? dotproduct : 0;
dotproduct = 0;
dotproduct += nn_input[0] * -0.18220123648643494;
dotproduct += nn_input[1] * 0.7036496996879578;
dotproduct += 0.025200579315423965;
output_cur[1] = dotproduct > 0 ? dotproduct : 0;
```

The main advantages of this optimization are the elimination of computational overhead due to branching on the termination condition and the delay of reading data from memory, since everything needed for the layers operations is self contained in a C source file. Similarly to Version 2, we have the capacity of doing a better instruction pipelining. Additionally, we remove incertitude about the execution path, which is advantageous for the WCET analysis. However, it worsens the drawback already identified in the V2: the instruction space becomes huge and for large DNNs, the approach is not sustainable.

4 Comparative approach for C code generation frameworks

In order to test in practice the advantages and limitations of our framework, as well as its behavior compared to the other frameworks in the literature, we have defined the following methodology. We have selected a set of representative benchmarks (section 4.1) of the literature compliant with our restrictions (e.g. feed-forward DNN with restricted types of layers). The idea was to consider a large test campaign by varying several parameters (number

and type of layers, data type of parameters, type of activation). We then define three criteria to assess the quality of implementation in accordance with the DO-178C requirements (see section 4.2). In particular, not all criteria require the same level of test campaign: computing the WCET needs to be done once whereas the measurements need to be repeated several times. Finally, we introduce the two code generation frameworks selected for comparison (see section 4.3).

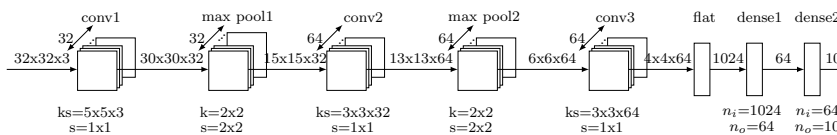
4.1 Benchmark description

Fully-connected networks – ACAS-Xu experience. The first models correspond to the classical fully-connected networks as shown in the example 12. We rely in particular on the airborne collision avoidance system for unmanned aircraft (ACAS-Xu) [27]. The ACAS-Xu system takes five input variables, i.e., information from sensors measurements, and computes five action advisories, represented by scores. The original design relies on a set of off-line computed lookup tables (LUT) to make avoidance decisions. Some work [19, 12] proposed to replace those LUT with some surrogate neural networks in order to reduce the memory footprint and thus to improve the execution time. We consider several DNN models with various structures, all with a ReLU activation function in hidden layers, linear activation for output layer and floating-point single precision (FP32) data type:

- regular structures with the same number of neurons per layer. We consider 7 hidden layers with *reg50* (50 neurons per layer), *reg100* (100 neurons per layer) and *reg200* (200 neurons per layer);
- decreasing structures with *decr128* (5 hidden layers of size (128, 128, 64, 32, 16)) and *decr256* (6 hidden layers of size (256, 256, 128, 64, 32, 16));

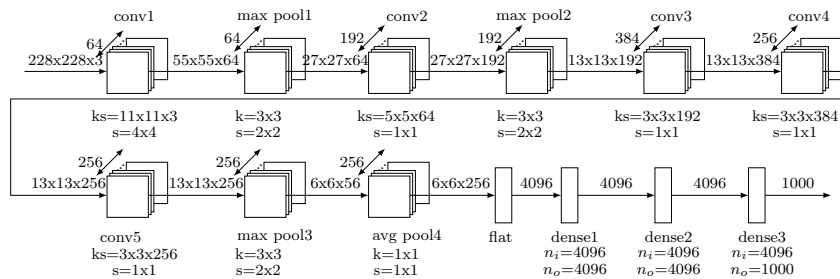
LeNet-5. The LeNet-5 model [23] refers to the feed-forward convolutional neural network introduced in the example 13. It is one of the earliest models of this type and is known for promoting the development of deep learning with the introduction of the back-propagation algorithm. Although this model is simple, it contains the main basic layers: convolution, pooling and dense layers. All the layers have the same tangent hyperbolic activation function, except for the last one, where a softmax is performed. Thus, it has 44,426 trainable parameters to stock and an inference pass executes 572,504 floating-point operations (FLOPs).

CifarNet. CifarNet was first introduced in [20] and was for a long time the state-of-the-art model used to solve the object classification problem on the Cifar-10 dataset, which consists of 32 x 32 RGB images of 10 classes. CifarNet is composed of three convolutional layers, and its pooling layers, followed by two dense layers (see figure 5). The ReLU activation function is applied to all the layers. The main difference with LeNet-5 is that it has a three-dimensional input and the convolutional layers have additional parameters such as padding and a non equal to 1 stride, which adds some complexity in terms of computation. With this configuration the number of trainable parameters increases to 122,570 alongside with 9,18 million FLOPs for inference.



■ Figure 5 CifarNet CNN.

AlexNet. The AlexNet architecture was first defined in [21] and is considered as one of the most influential works in computer vision. Indeed, thanks to the use of convolution layers and GPUs to accelerate deep learning, it achieved a considerably improved performance over other methods in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of 2012. The ImageNet dataset [13] is composed of 256 x 256 RGB images categorized under 1000 object class categories. AlexNet has five convolution layers, three pooling layers and three dense layers (see figure 6⁴) with approximately 61 million tunable parameters and 1,64 billion FLOPs. Additionally, this model uses the ReLU activation function, which was presented as novelty and proved to be more efficient in learning phase than the, at the time, standard hyperbolic tangent [21].



■ **Figure 6** AlexNet CNN.

4.2 Criteria of comparison

We have identified three criteria of comparison that correspond to the most important avionics constraints to be respected.

Semantic preservation. To validate the correctness of the code generation, we need to prove the *semantic preservation*, that is the capacity to reproduce the inference observed in the training tool on the target. To do so, we could have used formal methods (such as Coq [36] as was done for Velus [6]) but instead, we chose to review the code generated and run a large campaign of tests. This technique may be less sound but is indeed an acknowledged way in the certification standard DO178C [14]. The semantic preservation is assessed by comparing the predictions of the C code with those provided by the training framework.

► **Definition 15** (Semantic preservation). *Let $x = (x_1, x_2, \dots, x_n)$ be a vector representing the training framework outputs for a given set of inputs and $\tilde{x} = (\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_n)$ be the vector of outputs of the C code execution. We define the absolute error as*

$$\|\tilde{x} - x\|_{\infty} = \max_{0 \leq i \leq n} |(\tilde{x} - x)_i|$$

This norm asserts a maximum bound on the error observed for a given testing sample.

Measured and Worst Case Execution Time. For each C code, we want to assess both its performance and its predictability. The performance is evaluated by executing the code on an Arm Cortex-A15 of the keystone [37] and measuring the execution time.

► **Definition 16** (Measured execution time). *To obtain the measured execution time, we run a sample (i.e. an input) 50 times and store the average observed time.*

⁴ There is a pre-processing that consists of a *scaling* from 256 x 256 to 228 x 228 of the input

In order to assess the predictability, we compile each C code for a *lpc2138* Arm-based target, and compute its WCET with OTAWA [4]. The choice of the hardware (*lpc2138* Arm-based target) was dictated by the libraries available in the OTAWA framework. Even though it is not representative of the Arm Cortex-A15 of the Keystone, the comparison between the WCETs of the various versions still provides valuable insights on how the shape of the generated code impacts the level of precision that can be achieved during the analysis. Despite its limitations OTAWA is open-source and presently maintained, thus up to date with current WCET calculation techniques. We did not experiment with other static timing analysis tools that may or may not have the same limitations.

Memory layout of executable. Because it is important to efficiently use the resources in order to be predictable and efficient, we also analyze the memory layout of the C executable. The memory space is segmented into discrete blocks with specific purposes. We mainly focus on the stack, data, BSS and text segments. The stack segment contains all the data needed by a function call, including the arguments passed to the routine and its local variables. The data segment contains the explicitly initialized global variables and static local variables, its size does not change at runtime. Uninitialized variable data are stored in the BSS segment. Lastly, the text segment contains the executable instructions and constant variable that can not be modified.

4.3 Others C back-end frameworks

We chose two open-source frameworks from the TinyML [31] domain that were developed with nearly the same objectives as ACETONE.

Keras2C. As explained in [10], the Keras2c back-end was developed to address real-time applications and not to optimize the code for speed. Indeed, the generated C code layout is very similar to Version 1 presented in Section 3.2, where the programming functions describing the layers are generic and all the mutable data are passed into and out of each function during the inference execution. Thus, in terms of timing analysis, Keras2c presents the same downside as our first version, which is an overestimated WCET due to the inability of passing the context in which a function is called when there are multiple occurrences of it.

Another drawback observed in this framework is the declaration of the weights of layers as local variables initialized in the core of the function. Thus weight arrays are always allocated on the stack. In case of large networks, heavy arrays are then stored within the stack and such an approach is not at all recommended. Moreover, these arrays shall be initialized in each function with a memcopy from the reference one declared by the compiler in the text segment. This is not ideal for the computation time that is waste to copy weights for each layer and each inference. We preferred a zero copy strategy using static variable declaration for the weights, this saves space in the stack and computation time.

To avoid declaring to heavy array on the stack, Keras2c chooses to use dynamic memory allocation when working with large neural networks, which implies additional certification challenges in terms of verification and is not at all suited for WCET analysis.

MicroTVM with static C runtime. The TVM compiler [7] outputs a *model execution graph* – encoded as json – and simplified parameters. In order to execute the model, the TVM runtime has to rebuild this graph in memory, load the parameters, and then call the operator implementations in the correct order by parsing the computation graph. This is the principle of a graph interpreter / executor that we also found in Tensorflow.

uTVM [11] is a runtime developed to execute graphs produced by TVM on bare-metal targets. The code generation flow remains mostly the same, specific changes are needed in the runtime in order to avoid the usage of traditional operating-systems abstractions and support standalone model inference. The main parts of uTVM process are:

1. the production of a relay module depending of the training framework;
2. compilation, where TVM implements each operator into tensor intermediate representation followed by code generation;
3. integration of the generated code along with TVM C runtime library, into a user-supplied binary project;
4. and deployment, when a binary is built and inference can be run.

A drawback of this graph executor logic in an avionics context is the amount of memory overhead required in parsing the json, a dynamic scheduling and a dynamic memory allocation, which we are not able to analyze.

To bypass these limitations, [35] provided a patch to uTVM that relies on a static scheduling and memory allocation. We call that framework *uTVM with static C runtime* or static uTVM subsequently. It uses the relay module produced by TVM and generates a dedicated C source code that calls the generated operator implementations directly, eliminating the need of a graph json parsing, and which is able to execute the model statically. By doing minor changes in this static uTVM, we were able to proceed to a timing analysis of the inference model and could observe that the generated code when analyzed with OTAWA is very similar to our Version 2 (Section 3.3).

5 Experiments

This section summarizes the results when assessing the criteria for the different frameworks and the different benchmarks. We have in addition to the benchmarks identified before, considered VGG-16 [34]. Unfortunately, we only manage to generate the C code for V1 and V2 and analyze the semantic preservation. Other frameworks and analyses were not able to handle such a large network (138.36 million parameters).

Semantic preservation. We use the formula of definition 15 to compute the maximal observed error over 1000 tests when the generated code were executed on a x86 target. The three versions (V1, V2, V3) encode the same semantics, so no need to make them all appear. For our tool and for Keras2c, the reference was Keras and for static uTVM it was Tensorflow Lite. The results using single-precision FP are shown in table 1. We can note that all the frameworks produce very similar results with errors in the order of 10^{-6} , which is considered acceptable. For the ACAS-Xu regular models, using the learnt parameters (weights and biases) present in the lookup tables led to values larger than 1 (around 10^5) in outputs. This had an influence in the floating point precision which in turn affected our semantic preservation assessment, so we proceeded to use random initialized parameters and have normalized outputs instead.

Measured and Worst Case Execution Time. We measured the inference time on the Arm Cortex-A15 (implementing the ARMv7 architecture) of the keystone. For all experiments, caches were activated and we put data and code sections in the DDR. We used the flag *mfloat-abi=hard* in order to use the *neon floating point unit* of the processor. C codes were compiled without any optimization level (-O0). Table 2 shows the results where the measured execution times (MET) are computed following definition 16.

■ **Table 1** Results for the semantic preservation in FP32 precision.

Maximum error									
Framework	ACAS-Xu	ACAS-Xu	ACAS-Xu	ACAS-Xu	ACAS-Xu	LeNet-5	CifarNet	AlexNet	VGG-16
	<i>reg50</i>	<i>reg100</i>	<i>reg200</i>	<i>decr128</i>	<i>decr256</i>				
Ours (V1)	2.0265e-06	1.4305e-06	4.7683e-07	1.4305e-06	5.9604e-07	1.7881e-06	6.1988e-06	2.142e-06	4.7087e-06
Keras2C	2.0265e-06	1.4305e-06	4.7683e-07	8.34465e-07	9.5367e-07	2.0265e-06	5.6028e-06	–	–
uTVM static	1.6689e-06	9.5367e-07	1.1921e-07	2.3842e-07	2.3842e-07	1.9073e-06	4.2915e-06	–	–

■ **Table 2** Measured execution times on the Arm with -O0 flag.

Execution time [cycles]					
Framework	ACAS-Xu	ACAS-Xu	ACAS-Xu	LeNet-5	CifarNet
	<i>reg50</i>	<i>decr128</i>	<i>decr256</i>		
Ours (V1)	381 439	888 190	3 975 111	23 934 418	464 386 831
Ours (V2)	243 195	533 767	2 339 851	12 186 378	233 450 428
Ours (V3)	357 483	650 895	6 466 297	–	–
Keras2C	499 315	1 104 134	4 977 515	25 786 401	642 390 830
uTVM static	416 796	681 708	2 677 785	10 201 249	193 599 362

■ **Table 3** Measured execution times on the Arm with -O3 flag.

Execution time [cycles]		
Framework	ACAS-Xu	CifarNet
	<i>decr256</i>	
Ours (V2)	441 992	53 773 643
Keras2C	2 117 467	273 594 356
uTVM static	291 609	69 022 625

Among our versions, V2 produces the best MET. V2 has even a better MET than Keras2c and static uTVM for fully-connected networks (ACAS), and is slightly slower than uTVM for CNNs. Indeed, for the latter, static uTVM performs additional optimization (e.g. on tensor operations). On fully-connected networks, the tensor operations are basic matrix multiplications that do not require any optimization techniques. Keras2c has the worst MET for all benchmarks: we attribute that to the strategy to allocate weights tensors on the stack that adds a *memcpy* overhead at each layer (copy the weights from *.text* to stack).

Outside the avionics world, performance is looked for and thus inference codes are generally compiled with the -O3 option. Calling for this option enables the utilization of *Single Input Multiple Data (SIMD)* instructions on the keystone. We thus also compiled two benchmarks with this flag to observe the impact. The results are given in table 3. First, for all versions, the MET is greatly reduced, due to the SIMD instructions well adapted to these algorithms. Keras2c has the same drawback due to copy of weights on the stack. Since -O3 only optimizes the computation of tensor operations, the time to copy data remains the same. Thus, the difference between Keras2c and two others remains high. Secondly, V2 has best MET for CNN and worst with fully-connected network. We do not try to optimize the utilization of SIMD instruction (array organization), thus we also believe this is not the case of static uTVM. This would require a dedicated back-end for floating point unit of Arm.

■ **Table 4** WCET given by OTAWA for different benchmarks.

WCET [cycles]							
Framework	ACAS-Xu	ACAS-Xu	ACAS-Xu	ACAS-Xu	ACAS-Xu	LeNet-5	CifarNet
	<i>reg50</i>	<i>reg100</i>	<i>reg200</i>	<i>decr128</i>	<i>decr256</i>		
Ours (V1)	8 025 404	21 288 195	84 655 395	26 092 073	121 206 406	6 881 827 044	361 743 738 250
Ours (V2)	5 617 830	13 971 737	55 122 437	6 128 253	24 461 227	165 718 749	3 018 534 290
Keras2c	5 033 535	19 692 951	79 383 490	36 838 054	112 237 358	1 160 385 934	97 959 064 345
static uTVM	4 008 298	15 711 232	58 832 502	6 765 413	27 015 092	113 449 651	3 215 754 680

Table 4 shows the WCET of the benchmarks. OTAWA requires flow-fact information, that is information about the control flow: loop bounds and addresses of targets for indirect function calls (function pointers). Obtaining this information for our generated code was

easy (and making this process automatic is part of future work). For Keras2c and uTVM, we had to first modify the generated code to analyze only the inference code (as we did for our code), and to leave the initialization functions out of the WCET. OTAWA was not able to provide a WCET bound for V3 nor for AlexNet and VGG-16 architectures, because those binaries are too large and it runs out of memory during the analysis.

Looking at Table 4, we observe that shape of the C code has a significant impact on the WCET bound. This is not simply a question of performance optimizations, but also of the capacity to provide precise flow-fact information to the analyzer. C codes that employ function pointers (V1 and Keras2c) overall get larger WCETs than the others, because we were unable to provide contextual information about the layers function calls. When all layers perform an equivalent number of operations (the ACAS-Xu regular structures), this impact is reduced. For the other cases, the pessimism appears clearly such as for *decr256*. Indeed, although *decr256* performs less computations than *reg200*, as attested by the WCET of V2 and static uTVM, the WCETs for V1 and Keras2c are significantly higher than the ones of the *reg200*.

OTAWA assumes that each call to a layer function is a call to the worst layer. In V2, the layers are implemented as a sequence of separate loops, and in static uTVM as a sequence of separate instructions calling the layer functions. Consequently, OTAWA is able to benefit from the detailed flow-fact information for these versions.

Memory layout of executable. We analyzed the memory layout of the generated codes when compiled to ARM Cortex-A15. For the sake of simplicity, we only present the results obtained for the ACAS-Xu *reg50* model as the same trend is observed for the other models. From Table 5 it is possible to understand how different the memory usage of the different frameworks is.

■ **Table 5** Memory layout of the executable generated for ACAS-Xu *reg50*.

Benchmark	Size of memory segments [bytes]			
	stack	.data	.bss	.text
Ours (V1)	240	66 548	708	17 004
Ours (V2)	158	65 860	708	18 556
Ours (V3)	140	2 444	708	1 603 980
Keras2c	129 280	–	2 840	12 744 060
static uTVM	210	–	2 808	12 688 208

In our work, we privileged writing all parameters as constants to statically allocate all memory at compile time and better use the stack, which is also translated in the data segment size. The non-initialized data basically corresponds to the outputs. Additionally, in V3 we observe that the text segment is bigger since all constants are directly written in the C source code. We notice that, V1 and V2 are very efficient in terms of .text and stack size compared to Keras 2c or static uTVM. Because Keras2c allocates all the weights tensors on the stack, the stack size is higher than other versions. Moreover, weights shall also be present in the .text segment. We notice, that the stack size is much higher than the size required for storing weights. In addition Keras 2c allocates work arrays that are not used for computing dense layer. Our stack measurement is coherent with stack information given by *gcc* compiler. For uTVM, we explain the size of the .text by all tensor operations functions embedded in the TVM library. In our version, we only embedded necessary tensor operations function.

6 Related Work

We found plenty of frameworks that provide the possibility to run neural networks. Most of them rely on an inference engine that dynamically explores a computation graph. Without ignoring them, we decided to focus the related work on tools that are more adapted to avionics constraints.

Generic C code generator frameworks. The first work [8] is guided by avionics constraints as well and, in order to provide an efficient implementation of DNN inference models, the authors developed an automatic code generator that allows preserving semantics of the trained machine learning model. However, the code generation tool is not extensively described nor made available.

The second is Keras2c [10]. This method consists in a library to convert Keras models into real-time compatible C code, supporting a wide range of layers and relying only on C standard library functions. In the section 5, we have extensively compared our results with Keras2c. The study of [29] also investigates a predictable implementation of neural networks for safety-critical cyber-physical systems. They embed the Keras2c code on Patmos, a time-predictable processor, which is part of the larger T-CREST [32] project. The software tool-chain of the latter includes a LLVM-based compiler and the Platin tool for WCET analysis.

uTVM [11] is an extension of TVM that provides an implementation of TVM for micro-controllers already presented in section 4.3. The adaptation of uTVM with static C runtime [35] has extensively been compared with our results in the section 5.

N2D2 [33] is an end to end framework from the creation of the model to its implementation including the training. On the code generation, the authors explore how approximation techniques can improve the performance and energy efficiency of hardware accelerators in machine learning applications. We will assess these tools as future work.

Proprietary code generator frameworks. New massively parallel hardware adapted to neural networks need specific programming pattern in order to obtain the best possible computation performance. Hardware manufacturers provide tools that enable clients to generate optimized code for their target. We can cite eIQ [26] from NXP, TensorRT [25] from NVIDIA, KaNN [17] from Kalray or OpenVINO [16] from Intel. eIQn, TensorRT and OpenVino rely on a dynamic graph explorer runtime while KaNN proposes a static scheduling and memory allocation. These tools only generate optimized code for specific targets and do not implement a generic approach. Xilinx Vitis AI is a tool that generates application code for Xilinx targets. Such targets are composed of a host CPU (from the x86 or ARM families) and a hardware accelerator that is composed of programmable logic (FPGA). The tool generates C code for the host, and so-called “kernels” that are called by the host (using an API such as OpenCL) and executed by the accelerator. The data transfers between the host and the accelerator are handled using Xilinx runtime.

CoreAVI⁵ claims to develop code generation toolchains for AI models compatible with DO-178C and ISO-26262 requirements. They mainly target GPUs with Intel Tiger Lake or AMD E9171. We were not able to assess their solution and we believe that they are only supporting CUDA or Vulkan code generation (no C generation).

⁵ <https://coreavi.com>

The Matlab Coder toolbox allows the generation of the C code for the inference of an already trained network. The generated code requires no external library, which makes it portable. Ansys⁶ proposes to use the Scade toolchain to generate C code compatible with DO178C requirements. To our knowledge, this targets at this time traditional processors and relies on the conversion of neural networks models into Lustre nodes. Then, they use the qualified C code generator. The converter AI models into Scade will have to guarantee the semantic preservation.

LLVM front-end frameworks. TVM [7] is a tool capable of compiling machine learning models from different popular frameworks and generating specific low-level optimized code for a diverse set of hardware back-ends.

MLIR (Multi-Level Intermediate Representation Overview) [22] is a LLVM intermediate representation which was developed with the idea to use the same IR for all compiler optimizations (hence the “Multi-Level”). It contains particular features that target machine learning applications, in particular it is possible to represent computation graphs in MLIR. MLIR can be instantiated into dialects that allow to put the focus on particular aspects of the code, to specify constraints or apply specific optimizations. An example of MLIR dialect that is particularly relevant to critical embedded applications such as the ones we target is described in [30]: it enables the semantics of synchronous reactive applications inside an MLIR description.

7 Conclusions

Machine learning applications are proven to be useful and are largely used in many domains, however, most of them are not built with avionics constraints in mind. In this work, we presented our approach to automatically reproduce the inference model of feed-forward neural networks in C code, respecting semantic preservation, predictability and aeronautic requirements. We proposed a framework that is modular and straightforward, capable of generating readable and traceable code. We also compared the present work with the state of the art and proved our approach to be competitive in the evaluated criteria.

As future work, we have already identified along the paper many improvements to be made (e.g. compliance with Nef, automatic flow-fact generation). We will continue exploring other frameworks to get the best practices. We also plan to target parallel C code execution. The current versions are suitable for pipelining or parallelizing computations.

References

- 1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: <https://www.tensorflow.org/>.
- 2 Erin Alves, Devesh Bhatt, Brendan Hall, Kevin Driscoll, Anitha Murugesan, and John Rushby. Considerations in assuring safety of increasingly autonomous systems. *NASA*, 2018.
- 3 Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. <https://onnx.ai/>, 2019.
- 4 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis (regular paper). In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

⁶ <https://www.ansys.com/fr-fr/products/embedded-software/>

- 5 Siddhartha Bhattacharyya, Darren Cofer, David Musliner, Joseph Mueller, and E. Engstrom. Certification considerations for adaptive systems. *2015 International Conference on Unmanned Aircraft Systems, ICUAS 2015*, pages 270–279, July 2015. doi:10.1109/ICUAS.2015.7152300.
- 6 Timothy Bourke, L elio Brun, Pierre-  variste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th Conference on Programming Language Design and Implementation (PLDI)*, pages 586–601, 2017.
- 7 Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.
- 8 Sergei Chichin, Dominique Portes, Marc Blunder, and Victor Jegu. Capability to embed deep neural networks: Study on cpu processor in avionics context. In *10th European Congress Embedded Real Time Systems (ERTS 2020)*, 2020.
- 9 Jean-Louis Cola o, Bruno Pagano, C edric Pasteur, and Marc Pouzet. Scade 6: From a kahn semantics to a kahn implementation for multicore. In *2018 Forum on Specification Design Languages (FDL)*, pages 5–16, 2018.
- 10 Rory Conlin, Keith Erickson, Joseph Abbate, and Egemen Kolemen. Keras2c: A library for converting keras neural networks to real-time compatible C. *Eng. Appl. Artif. Intell.*, 100:104182, 2021.
- 11 TVM consortium. microTVM: TVM on bare-metal, 2021. URL: <https://tvm.apache.org/docs/topic/microtvm/index.html>.
- 12 Mathieu Damour, Florence De Grancey, Christophe Gabreau, Adrien Gauffriau, Jean-Brice Ginestet, Alexandre Hervieu, Thomas Huraux, Claire Pagetti, Ludovic Ponsolle, and Arthur Clavi ere. Towards certification of a reduced footprint acas-xu system: A hybrid ml-based solution. In *40th International Conference Computer Safety, Reliability, and Security (SAFE-COMP)*, pages 34–48, 2021.
- 13 Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 248–255, 2009.
- 14 EUROCAE / RTCA. Do-178c, software considerations in airborne systems and equipment certification, 2011.
- 15 Google. Protocol buffers, 2001. URL: <https://developers.google.com/protocol-buffers/>.
- 16 Intel. Open vino documentation, 2018.
- 17 Kalray. Kann platform for high-performance machine learning inference on kalray’s mppa  intelligent processor, 2021.
- 18 Kalray. Mppa  coolidge™ processor - white paper, 2021. URL: <https://www.kalrayinc.com/documentation/>.
- 19 Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *29th International Conference Computer Aided Verification (CAV)*, pages 97–117, 2017.
- 20 Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
- 21 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, L eon Bottou, and Kilian Q. Weinberger, editors, *26th Annual Conference on Neural Information Processing Systems*, pages 1106–1114, 2012.
- 22 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, et al. MLIR: scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *International Symposium on Code Generation and Optimization, (CGO)*, pages 2–14, 2021.

- 23 Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, 1989.
- 24 Y. Liu, C. Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. Enhancing the interoperability between deep learning frameworks by model conversion. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.
- 25 NVIDIA. Tensorrt documentation, 2021.
- 26 NXP. Eiq™ ml software development environment, 2020. URL: <https://www.nxp.com/design/software/development-software/eiq-ml-development-environment:EIQ>.
- 27 Michael P. Owen, Adam Panken, Robert Moss, Luis Alvarez, and Charles Leeper. Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2019.
- 28 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- 29 Hammond Pearce, Xin Yang, Partha S. Roop, Marc Katzef, and Torur Biskopsto Strom. Designing neural networks for real-time systems. *IEEE Embedded Systems Letters*, pages 1–1, 2020.
- 30 Hugo Pompougnac, Ulysse Beaunon, Albert Cohen, and Dumitru Potop-Butucaru. From SSA to Synchronous Concurrency and Back. Research Report RR-9380, INRIA Sophia Antipolis - Méditerranée (France), December 2020. URL: <https://hal.inria.fr/hal-03043623>.
- 31 Partha Pratim Ray. A review on tinymml: State-of-the-art and prospects. *Journal of King Saud University - Computer and Information Sciences*, 34(4):1595–1623, 2022.
- 32 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.
- 33 Olivier Sentieys, Silviu Filip, David Briand, David Novo, Etienne Dupuis, Ian O'Connor, and Alberto Bosio. Adequatedl: Approximating deep learning accelerators. In *24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS 21)*, 2021.
- 34 Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations (ICLR)*, 2015.
- 35 Rafael Stahl. ptvm staticrt codegen, 2021. URL: https://github.com/tum-ei-eda/utvm_staticrt_codegen.
- 36 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, version 8.0 edition, 2004. URL: <http://coq.inria.fr/>.
- 37 Texas Instruments. TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip. Technical Report SPRS893E, Texas Instruments Incorporated, 2013.
- 38 The Khronos NNEF Working Group. Neural Network Exchange Format, 2018.
- 39 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

Using Quantile Regression in Neural Networks for Contention Prediction in Multicore Processors

Axel Brando ✉ 

Barcelona Supercomputing Center (BSC), Spain

Isabel Serra ✉ 

Barcelona Supercomputing Center (BSC), Spain
Centre de Recerca Matemàtica, Barcelona, Spain

Enrico Mezzetti ✉ 

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

Jaume Abella ✉ 

Barcelona Supercomputing Center (BSC), Spain

Francisco J. Cazorla ✉ 

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

Abstract

The development of multicore-based embedded real-time systems is a complex process that encompasses several phases. During the software design and development phases (DDP), and prior to the validation phase, key decisions are taken that cover several aspects of the system under development, from hardware selection and configuration, to the identification and mapping of software functions to the processing nodes. The timing dimension steers a large fraction of those decisions as the correctness of the final system ultimately depends on the implemented functions being able to execute within the allotted time budgets. Early execution time figures already in the DDP are thus needed to prevent flawed design decisions resulting in timing misbehavior being intercepted at the timing analysis step in the advanced development phases, when rolling back to different design decisions is extremely onerous. Multicore timing interference compounds this situation as it has been shown to largely impact execution time of tasks and, therefore, must be factored in when deriving early timing bounds. To effectively prevent misconfigurations while preserving resource efficiency, early timing estimates, typically derived from previous projects or early versions of the software functions, should conservatively and tightly overestimate the timing requirements of the final system configuration including multicore contention. In this work, we show that multi-linear regression (MLR) models and neural network (NN) models can be used to predict the impact of multicore contention on tasks' execution time and hence, derive *contention-aware* early time budgets, as soon as a release (binary) of the application is available. However, those techniques widely used in the mainstream domain minimize the average/mean case and the predicted impact of contention frequently underestimates the impact that can potentially arise at run time. In order to cover this gap, we propose the use of quantile regression neural networks (QRNN), which are specifically designed to predict the desired high quantile. QRNN reduces the number of underestimations compared to MLR and NN models while containing the overestimation by preserving the high quality prediction. For a set of workloads composed by representative kernels running on a NXP T2080 processor, QRNN reduces the number of underestimations to 8.8% compared to 46.8% and 31.3% for MLR and NN models respectively, while keeping the average over estimation in 1%. QRNN exposes a parameter, the target quantile, that allows controlling the behavior of the predictions so it adapts to user's needs.

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

Keywords and phrases Neural Networks, Quantile Prediction, Multicore Contention

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.4



© Axel Brando, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla;
licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 4; pp. 4:1–4:25



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant PID2019-110854RB-I00 / AEI / 10.13039/501100011033 and the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 772773).

1 Introduction

Software applications control an increasing number of complex functionalities in real-time embedded products. For example, in the automotive domain, Advanced Driver Assistance System (ADAS) functionalities, like lane keeping and obstacle detection, in modern cars are implemented in software, which is going to be the central element to reach full (L5) autonomy. This trend towards smarter artificial intelligence based on-board software drives an unprecedented increase in the size of the software component of embedded real-time systems in domains like automotive and avionics. In fact, embedded real-time products already encompass software with millions of lines of code. On the other hand, the use of multicores to provide the required computing performance compounds the complexity to develop and validate multi-million line real-time software products.

During the design and development phases (DDP) engineering process, the integrator selects the configuration of the hardware by choosing values for the control registers (critical configuration settings in CAST-32A [9] jargon). Also, in order to complete the intended final configuration (IFC) [9], the integrator determines the mapping of tasks to each computational node, which in turn determines which tasks will be co-executed in the multicore and hence, compete for its resources. Those decisions are steered by the timing and functional requirements of the software functions the system is meant to support. Based on those requirements and the system schedule, the software providers are assigned a time budget for each of their software function which is meant not to be exceeded at run time. In the DDP, multiple configuration scenarios (e.g, configuration, task mapping, schedules) are assessed in order to converge to the system’s IFC.

However, timing budgets are only consolidated against timing requirements in the late validation phases when timing analysis is typically performed to derive reliable worst-case execution time (WCET) bounds for each task. Capturing a timing misconfiguration so late in the development process will result in costly roll-backs in the design and implementation phases. In particular, building on optimistic timing estimates to derive and allocate time budgets to an application will result in timing violations to arise in the verification and validation stages and will require changes to the application itself and/or to the system schedule, which will cause the system to undergo once again through the V&V process. For this reason, while early figures are not meant to be as accurate as late WCET timing bounds, they are still required to conservatively over-estimate tasks’ timing requirements as much as possible. Moderate over-estimation can lead to slight over-provisioning and will not jeopardize the overall system timing behavior.

Providing early timing estimates for the software functionalities is a challenging task in many ways as estimates are typically derived from previous experience on past projects or from representative early software implementations. The provision of such early estimates is even more challenging when the system is deployed on multicore platforms because tasks affect each other timing behavior causing variable access latencies when simultaneously accessing shared hardware resources. This translates into variability in their execution time, typically referred to as *multicore timing interference* or *contention impact*, which can cause 20x or more performance degradation [29, 55] and ultimately complicates the determination of trustworthy time bounds.

Exploring and assessing a large set of scenarios in the DDP requires assessing also the impact of contention in each considered configuration, where task mapping and schedule play a critical role. A naive approach to derive contention information during DDP consists in exhaustively executing each scenario on the actual board collecting evidence on how tasks affect each other timing. This approach, however, is quickly defeated by the time it takes to carry each experiment, which can limit the design space (i.e the schedules) that can be explored. Instead, a faster approach consists in developing multicore prediction models that can provide accurate estimates of tasks execution time when co-executed in a multicore.

Analysis. Multi-linear regression (MLR) models and neural network (NN) models, which have been originally developed for the mainstream domain [52, 11, 59, 34], can be adapted to the problem at hand to explore a large fraction of the design space in a short amount of time. In particular, such models could be in principle exploited to produce early timing estimates of a system as soon as a binary release of the applications is available. While these approaches produce reasonably accurate estimates of tasks' execution time, they are inherently designed to predict the average behavior of the phenomena they model, since the most accurate prediction is the one closer to the majority of cases and thus closer to average or median patterns. As we observed, it is crucial for early estimates in DDP to be over-approximating the behavior in the final configuration and we must seek for more conservative models to diminish the risk of being misled into optimistic estimates.

Proposal. On these grounds, we propose a prediction model based on quantile-regression neural networks (QRNN) that can conservatively predict the impact of multicore timing interference. QRNN aim at optimizing the quantile regression loss function which, generically, allows approximating any conditional desired quantile. This enables the user to choose the (high) quantile that best adapts to its needs. Overall, QRNN allows fast evaluation of system configuration by providing conservative, yet accurate, predictions of contention impact.

Evaluation. We show the benefits of QRNN over MLR and NN on a set of representative kernels used in artificial intelligence software for autonomous operation on an avionics representative multicore processor, the NXP T2080. Our results show that QRNN reduces the number of workloads for which time budgets under estimate (i.e. are lower than the actual multicore contention time of the task) to 8.8% compared to 46.8% and 31.3% for MLR and NN, respectively.

The rest of this work is organized as follows. Section 2 narrows down the specific multicore contention problem addressed and introduces MLR and NN. Section 3 introduces QRNN. Section 4 presents our evaluation framework. Section 5 reports on the experimental results. Section 6 covers the most relevant related work. Section 7 presents some lines that can be explored as a follow up of this work. Section 8 presents the main conclusions of this work.

2 Multicore Contention Prediction

Multicore contention modeling is a wide problem that spans several domains and stages in the software development process [43, 11, 42, 3]. We start by narrowing down the particular multicore contention problem we address and a set of properties for the resulting techniques to adhere to the specific requirements of the particular application scenario (those properties were summarized already in Section 1). The main acronyms we use are described in Table 1.

We focus on a deployment scenario in which the target multicore platform is fixed and the set of applications to be integrated in the final embedded product is known. That is, the functionality to be provided for the product is frozen and so is the software to implement it. A first release of the applications has been made so there exists an executable of each

■ **Table 1** Main acronyms used in this work.

Acronym	Definition	Acronym	Definition
ADAS	Advanced Driver Assistance System	DDP	Design and Development Phases
DS	Data Set	EMs	Event Monitors
H	Holdout set	IFC	Intended Final Configuration
M	All the model hyper-parameters	MAE	Mean Absolute Error
MCP	Multicore Processor	MLR	Multi-Linear Regression
MSE	Mean Square Error	NN	Neural Network
QR	Quantile Regression	QRNN	Quantile Regression NN
SoC	System on Chip	TOI	Task Order Invariant
TUA	Task Under Analysis	TVE	(T)rain, (V)alidation, t(E)st set
V&V	Validation and verification		

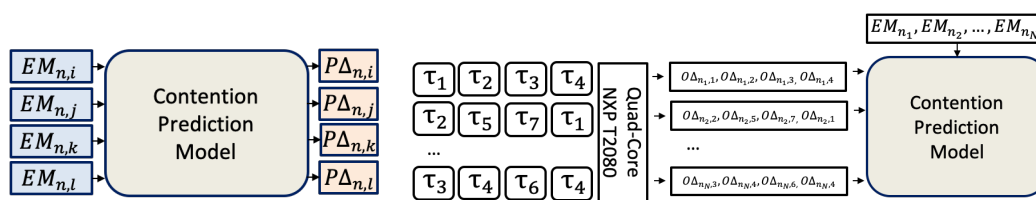
application. Applications can suffer variations as part of the natural development process across different releases, yet preserving the same functionality. These variations include, for instance, optimizations to its performance or functional behavior. We target a *homogeneous multicore processor* in which the performance of each core is identical and the time to access any off-core resource is the same from every core. This is the case of the T2080 when its cache is not partitioned or when the L2 cache is configured so each core receives an even number of ways, as we do in this work. However, it is not the case if, for instance, L2 partitions across cores differ in size. It is nor the case for other architectures, like some models of the Intel Xeon, in which the cores and slices of the L3 cache are connected via a ring interconnect, so the time it takes a core to access a slice depends on their location in the ring.

Our work contrasts with other works that derive early estimates at the model level (e.g. Matlab) [17] and estimates “as the code is written” [18]. Others focus on scenarios where the hardware platform is not even available and compile the source code for different instruction set architectures on generic and parameterizable processor models to obtain early timing estimates on the impact of the architecture setup [19]. The majority of these approaches focus on the analysis of programs in single-core scenarios and do not address the impact of multicore contention. The works addressing multicore interference, instead, necessarily consider more mature setups where consolidated or even final software products are made available [52, 11, 61, 59, 50].

The goal of our contention modeling exercise is not producing a generic model for the target platform (T2080 in our case) that is application independent. Instead, the model considers the applications provided and contributes to speeding up the selection of the IFC.

► **Property 1** (Prediction speed). *DDP multicore contention models for real-time systems must be fast to enable exploring large design spaces.*

Several previous works [29, 55] show that contention may dominate the execution time of tasks running in a multicore with some applications easily suffering an increase above 2x-5x with respect to their solo execution time even for small core counts like 4 cores (corner case programs can suffer much higher slowdown). For DDP, no reference figure has been reported for the accuracy of timing predictions, which is in fact end-user and application dependent. Yet, we regard the pessimism introduced by our QRNN model (1% on average and 1.49% in the worst case) as quite reasonable for DDP. Besides it is key to produce conservative early timing estimates that tend to over-approximate the behavior in the IFC, therefore reducing the risk of producing optimistic estimates.



■ **Figure 1** Contention Models Usage. ■ **Figure 2** Contention Modelling Training.

► **Property 2** (Tendency to Overestimation). *DDP multicore contention models for real-time systems should tend towards overestimation to reduce the risk of experiencing timing violations too late in the development process, requiring excruciatingly onerous rollbacks and re-design.*

Trustworthy execution time bounds for a task τ_i can be derived when the task executes in isolation, ET_i^{solo} . For multicore processors (MCP), software's timing behavior also depends on the contention factor, often considered as a Δ over its execution time in isolation¹, which is expressed as $ET_i^{mcp} = ET_i^{solo} \times \Delta$.

Deriving time budgets for the multicore execution time requires estimating a bound to Δ . Contention bounds can be derived by experimentation, i.e. by running *all* potential workloads on the target board under the IFC so that the timing budget for τ_i can be expressed as $TB_i^{mcp} = ET_i^{solo} \times O\Delta_i^{max}$, where $O\Delta_i^{max}$ is the maximum observed contention impact suffered by τ_i . However, this approach is inherently time consuming and cannot be exploited for exploring non-negligible design spaces.

In terms of the number of workloads, for a heterogeneous multicore it can be computed as the permutation with repetition of all contenders A^C , where A is the number of applications in the data set (DS) that can repeat in several cores and C is the number of cores. The number of workloads reduces to $CR_A^C = \frac{(A+C-1)!}{C!(A-1)!}$ for homogeneous multicore architectures.

In terms of runs, depending on the experimentation environment in each run of a workload we can obtain the slowdown for one of the tasks in the workload or all C tasks. The former, our case, requires C runs per workload to obtain $O\Delta_{n,i}$ for each task, while the latter needs one per workload. However, for homogeneous multicores, fewer runs are required when several copies of the same task are present in the workload, in particular, $A \cdot CR_A^R = A \cdot \frac{(A+R-1)!}{R!(A-1)!}$ where $R = C - 1$ is the number of contenders.

Overall, in the general case exhaustively covering all configurations on the real board is unaffordable, even if each experiment requires just few milliseconds. With the number of cores in the multicore processors evaluated in the real-time domains increasing (e.g., the NXP Layerscape LX2160 already encompasses 16 cores), the number of workloads increases to millions.

2.1 Contention Modeling

Contention models are generally orders of magnitude faster than experimentation in the target board and can be executed in high-performance computing clusters, which allows many more parallel experiments than making executions on few target boards that can be available for experimentation. In this line, standard fully-fledged timing analysis techniques are not fit for deriving early estimates. Measurement based timing analysis requires running each workload on the target board whereas static timing analysis has known scalability issues.

¹ The main terms used in the mathematical formulation is summarized in Table 2. Instead Table 1 shows the main acronyms used in the main text.

■ **Table 2** Main terms used in the formulation (notation) in this work.

Term	Definition	Size
A	Total number of tasks (applications) in the data set	
C	Number of cores (e.g. $C = 4$)	
DS	Data Set	
H, I	Number of output and input values	
h_i^l	i -th input/output value in layer l	
J	Number of EMs per task/core (e.g. $J = 262$)	
K	Number of EMs per workload, $K = J \cdot C$ (e.g. $K = 1048$)	
N	Number of workloads in the DS	
$EM_{n,i}$	All the EM of task τ_i in workload n when it runs in isolation	J
EM_n	All the EMs of the n -th workload (point) in the DS	K
EM_*	All the EMs for all the workloads in the DS	$N \cdot K$
$em_{n,i}^j$	The j -th EM of τ_i of workload n	1
em_n^k	The k -th EM of the n -th workload (point) in the DS	1
ET_i^{mcp}, ET_i^{solo}	Execution time of task τ_i in multicore processors and in isolation	
$O\Delta_{n,i}$	Observed contention for τ_i in workload n	1
$O\Delta_n$	Observed Contention for all tasks in workload n	C
$O\Delta_*$	Observed Contention for all workloads in the DS	$N \cdot C$
$P\Delta_{n,i}$	Predicted Contention for τ_i in workload n	1
$P\Delta_n$	Predicted Contention for all tasks in workload n	C
ϕ	Neural Network (function)	
R	$C - 1$	
TB^{mcp}	Time Budget in multicore	

Contention models produce an estimate to contention in the form of a *predicted* Δ ($P\Delta$), so that $TB_i^{mcp} = ET_i^{solo} \times P\Delta_i^{max}$, where $P\Delta_i^{max}$ is the maximum predicted contention impact. The process of deriving $P\Delta_i^{max}$ builds on several factors that capture the contention a task can suffer from and generate on co-runner tasks.

In real platforms, event monitors (EMs) provide insightful information about how a task uses shared resources, which in turn are the inherent sources of contention. EMs report metrics like access counts to resources, hit/miss accesses to cache memories, and other activities of the task on the underlying hardware.

In this work, we target the NXP T2080 [22], a quad-core MPSoC which is currently considered for certification for avionics products [48]. The T2080 comprises 262 EMs that provide insightful information on the use of resources of the analyzed application at core, shared L2, and memory levels. For a given workload, the EMs collected for each task while running on the T2080 in isolation are fed as input to the contention model.

As shown in Figure 1 for a quad-core processor, to predict the contention impact, the contention models use the EMs collected (in isolation) for all the tasks in the workload, denoted as n , constituting a function named f . The predicted contention impact for task τ_i when running in workload n , together with τ_j, τ_k, τ_l , is denoted as $P\Delta_{n,i} = f(EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l})$. $EM_{n,i} \in \mathbb{R}^J$ are all the EMs (collected in isolation) of a task τ_i where J is the number of EMs read per core ($J = 262$ in the T2080).

For the training of the model, see Figure 2, we build on the results of executing multicore workloads, generated from a set of A tasks that are executed on the available cores C on the target board (one task per core). The observed (real) contention $O\Delta_{n,i}$ for each task τ_i in each workload n is collected and used in order to compute $P\Delta_{m,i}$ in a different (unseen) workload m .

2.2 Formalization

Several techniques have been proposed in the mainstream domain for multicore contention prediction, from which we identify two families: MLR- and NN-based models [52, 11, 59, 34]. A commonality of the different models is that they create an input data set (DS) for training. Such input DS is composed by the EMs collected for several tasks used to compose the workloads and the observed slowdown when executing a subset of workloads on the target board. Reducing the subset of this input DS used for training contributes to Property 1. The input DS is shown in Equation 1:

$$\mathcal{DS} = (EM_*, O\Delta_*) = \{(EM_n, O\Delta_n)\}_{n=1}^N \quad (1)$$

$EM_* \in \mathbb{N}^{N \times K}$ are the EMs of all the N workloads in DS. K is the number of EMs read in total, that for the case of the NXP T2080 is $K = 4 \times 262 = 1048$, since $C = 4$. $EM_n \subset \mathbb{N}^K$ are the EMs all the tasks in the n -th workload when executed in isolation. That is, EM_n is the concatenation of the EMs of each task composing the workload when run in isolation.

$O\Delta_*$ is the observed contention for all the tasks in all workloads in the DS. Likewise, $O\Delta_n = \{O\Delta_{n,1}, \dots, O\Delta_{n,C}\} \subset \mathbb{R}^C$ is the contention for the C executed tasks in workload n .

2.3 Multi-Linear Regression (MLR) Models

For a Tasks Under Analysis (TUA), τ_i , of the n -th workload in the DS, a multi-linear model is a linear transformation from the EM values (EM_n) to the $P\Delta_{n,i} =: \hat{y}_n$, where i is omitted because \hat{y}_n is always referring to the TUA. The MLR can be formulated as follows:

$$\hat{y}_n = W \times EM_n + b \Leftrightarrow \hat{y}_n = w_1 \cdot \underset{\uparrow x_1}{em_n^1} + w_2 \cdot \underset{\uparrow x_2}{em_n^2} + \dots + w_K \cdot \underset{\uparrow x_K}{em_n^K} + b \quad (2)$$

where each $em_n^k \in \mathbb{N}$ is the k -th EM in the n -th workload. As we can see in Eq. 2, we can also use $\{x_k\}_{k=1}^K$ to refer to them. $EM_n \in \mathbb{N}^K$ refers to the EMs input information for workload n . $w_k \in \mathbb{R}$ and $b \in \mathbb{R}$ are the weights to be learnt that define the linear combination between the EMs values and the predicted $\hat{y}_n = P\Delta_{n,i} \in \mathbb{R}^+$.

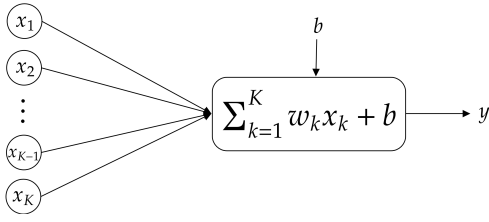
The goal of the MLR is to find the weights $\{W, b\}$ that minimize a certain distance function (known as the loss function) between the predicted output and the real response variable value with respect to the training split set. This minimization process can be typically performed in two different ways.

Given that the MLR is a linear combination of coefficients with the input information, the least-square estimate of W can be computed using the DS where we identify the TUA τ_i , $(EM_*, \{O\Delta_{n,i}\}_{n=1}^N)$:

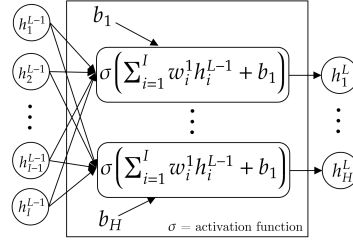
$$\hat{W} = \underbrace{(EM_*^T \cdot EM_*)^{-1}}_{\text{Computationally expensive.}} \cdot EM_*^T \cdot [O\Delta_{1,i}, O\Delta_{2,i}, \dots, O\Delta_{N,i}]^T, \quad (3)$$

where the T superscript denotes the matrix transpose operation, the -1 superscript refers to the inverse of the matrix² and $[O\Delta_{1,i}, O\Delta_{2,i}, \dots, O\Delta_{N,i}]^T$ is the column vector that contains all the contention values for all the N workloads. Importantly, this way of obtaining the optimal weights has a potential drawback in most of the real-world situations, as the inverse of an $N \times N$ matrix must be computed, which has polynomial time complexity.

² To simplify the notation, the \hat{W} matrix implicitly contains the bias b column in that case.



■ **Figure 3** Multi-linear regressor.



■ **Figure 4** Dense neural layer. Given I layer-input values, $\{h_i^{L-1}\}_{i=1}^I$, it provides H layer-output values, $\{h_h^L\}_{h=1}^H$.

As an alternative approach to avoid computing the inverse matrix, we can compute this minimization process by slightly modifying the weights in the gradient direction, i.e. applying a gradient descent method. Nowadays, this differentiation process is implemented in most relevant deep learning libraries, which allows native code to be differentiated automatically [1, 45, 12, 7]. This derivative is computed with respect to a loss function, which can be the mean square error (also known as least-square estimate) that approximates the conditional mean, or an alternative function, as we will see in the next section.

2.4 Neural Network (NN) Models

A NN is also a parametric function ϕ that transforms a vector of EM_n to a predicted contention for a task under analysis τ_i in workload n , $\hat{y}_n = P\Delta_{n,i}$, i.e. it is defined as $\phi: \mathbb{R}^K \rightarrow \mathbb{R}$, transforming $EM_n \mapsto \hat{y}_n$. Instead of a single matrix multiplication such as in MLR, the NN considers several internal non-linear transformations from the input, EM_n , to produce the output value \hat{y}_n . Each of these transformations is known as a “layer” and combines its input values and weights to produce its output, which for the last layer is the output of the model [36, 16]. Roughly speaking, the NN combines a mixture of weights and its input values to minimize a certain distance loss function (as in the MLR case) between the predicted and the real response compared to the DS used to train the model.

Figure 4 represents one NN layer where $\{h_i^{L-1}\}_{i=1}^I$ represent the inputs to the layer and I is the number of neurons in the layer. In the first layer that is $I = K$ and $h_k^0 = em_n^k$ for each $k = 1, \dots, K$ when a n -th workload is fixed. Each transformation, represented as a rectangle in Figure 4, matches Equation 2 with the addition of the non-linear activation function, denoted as σ , which allows the enhance approximation capabilities of the NN by means of the layer stacking process [13]. Each layer will produce a set of outputs, $\{h_h^L\}_{h=1}^H$, where H is the number of neurons in the next layer, then will be either used as inputs to the next layer or as final NN output in case of the last layer.

In probabilistic terms [26, 51], the loss function aims to approximate the conditional probability $p(Y | \mathbf{X}, M)$, where \mathbf{X} represents the theoretically random variable that generate the input values – in our case the EM_n values, Y represents the corresponding random variable that generates the contention values $O\Delta_m$ and, finally, M is the random variable that characterizes all the hyper-parameters in the NN (including the number of layers, the type of layers, the parameters about the learning configuration, etc). Importantly, the conditional probability approximated can be affected by the hyper-parameters selection, which is, therefore, a critical step to consider for the whole process. In this probabilistic context, the common approach [31, 40] is to follow a Maximum Likelihood Estimation (MLE)

or a Maximum A Posteriori (MAP) approach to compute such conditional probability and to assume the sample mean is asymptotically normal, which consequently leads to use the following loss function:

► **Definition 1** (Mean Square Error). *Let $\mathbf{X} \in \mathbb{R}^K$ be a covariate random variable and $Y \in \mathbb{R}$ be a response random variable (i.e. the random variables that generates the input and output values, respectively). Reducing the conditional³ Mean Square Error (MSE) consists in finding a function $\phi: \mathbb{R}^K \rightarrow \mathbb{R}$, characterized by M , which approximates the conditional mean of $p(Y | \mathbf{X}, M)$ by minimizing the loss function defined as*

$$\mathcal{L}_{MSE}(\mathbf{X}, Y) = \mathbb{E} \left[\left(Y - \phi(\mathbf{X}) \right)^2 \right] \approx \frac{1}{N} \sum_{n=1}^N (y_n - \phi(\mathbf{x}_n))^2. \quad (4)$$

For the problem at hand, $\phi(\mathbf{x}_n) = \phi(EM_n)$ is the evaluation of the NN over the EMs (1048 for the T2080) for a certain workload n , producing a forecast $\hat{y}_n = P\Delta_{n,i}$. Similarly, the MLR can be used into this equation as the ϕ function, i.e. $\phi_{MLR}(x_n) = W \cdot EM_n + b$.

The conditional mean is a generically good estimator, and an ideal one in scenarios where the Central Limit Theorem is applicable. However, when the approximated $p(Y | \mathbf{X}, M)$ corresponds to a heavy tailed distribution (or even has some important outliers), computing a conditional mean can lead to unreliable decisions. Then, the median can be a more stable estimator in the presence of certain outlier values. In fact, this is equivalent to repeat the previous MLE reasoning for the normal distribution but using the Laplace distribution. In such case, the conditional loss function is the following:

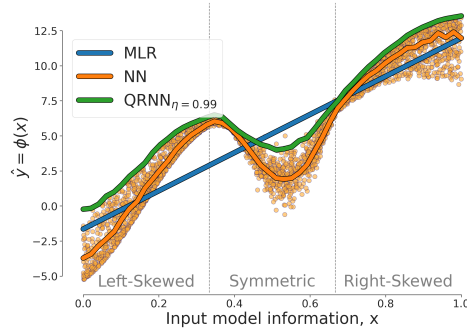
► **Definition 2** (Mean Absolute Error). *Let $X \in \mathbb{R}^K$ be a covariate random variable and $Y \in \mathbb{R}$ be a response random variable. Reducing the conditional⁴ Mean Absolute Error (MAE) consists in finding a function $\phi: \mathbb{R}^K \rightarrow \mathbb{R}$ that approximates the conditional median of $p(Y | \mathbf{X}, M)$ by minimising the loss function defined as*

$$\mathcal{L}_{MAE}(\mathbf{X}, Y) = \mathbb{E} \left[\left| Y - \phi(\mathbf{X}) \right| \right], \quad (5)$$

which provides results that are more robust to outliers and more interpretable than the commonly used MSE [58, 10]. However, a NN optimized with the MSE or the MAE will predict a central conditional value. Therefore, while being appropriate for deriving predictions that are close to the actual values, by definition it will not be able to compute upper-bounds. In other words, a perfect MAE estimation will have a 50% probability of having real values above and below the predicted point. Thus, it should not be used as a proper high-value threshold.

While computing a confidence interval around such central value is technically possible, this brings multiple challenges related to (i) the assumptions on the actual distribution for each value to predict (i.e., whether it can be regarded as Gaussian or not), (ii) computational cost to estimate the confidence interval for each predicted value across the prediction value range, and (iii) variability in the confidence reached (or tightness of the bounds) due to the arbitrary variability in the amount of data that can be available for each predicted value (e.g., for some predicted value ranges we may have very few input observations). Hence, we discard computing confidence intervals for NN based prediction.

³ The term “conditional” is added to highlight that here the information about X should be provided to compute the error of the f with respect to Y . This also makes this definition consistent with the conditional QR definition introduced afterwards.



■ **Figure 5** QRNN versus MLR and NN.

3 Quantile Regression NN

As presented in the previous section, classical NN are usually optimized using the MSE (see Eq 4) or the MAE (see Eq 5), which corresponds to estimate the conditional mean or median, respectively. In this section, we introduce the Quantile Regression (QR) method [32, 8], which allows approximating a desired quantile of the conditional distribution $p(Y | \mathbf{X}, M)$. This is visually represented in Figure 5 that for a given 1-dimension input (the horizontal axis) the goal is to predict the height of the points (the vertical axis). In particular,

- As it can be seen, MLR assumes a linear correlation between the input and output variable, which induces the prediction to be a conditional line.
- NN introduces the possibility to learn the conditional mean (or median) in a non-linear manner but, still, this cannot be used as an upper threshold.
- QRNN allows to approximate a sky-high conditional quantile in a non-linear way, which avoids strong assumptions such as linearity or symmetry between the predicted distribution, i.e. the conditional predicted distribution $p(Y | \mathbf{X}, M)$ can be skewed (such as the initial and final part of Figure 5) and the QRNN obtains a proper response.

This is useful since we can capture confidence intervals without making strong assumptions about the distribution function to approximate. The formal definition of QR depending on \mathbf{X} is as follows:

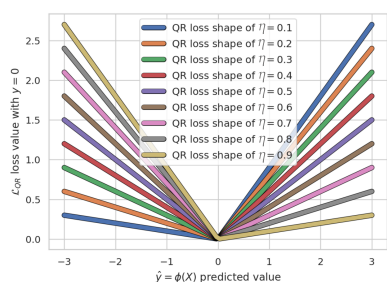
► **Definition 3** (Quantile Regression). *Let $\mathbf{X} \in \mathbb{R}^K$ be a covariate random variable and $Y \in \mathbb{R}$ be a response random variable. Given η in the real interval $[0, 1]$, the conditional quantile regression (QR) consists in finding a function $\phi_\eta: \mathbb{R}^K \rightarrow \mathbb{R}$ which approximates the η -th quantile of $p(Y | \mathbf{X}, M)$ by minimizing the η -th QR loss function defined as*

$$\mathcal{L}_{QR}(\mathbf{X}, Y, \eta) = \mathbb{E} \left[\left(Y - \phi_\eta(\mathbf{X}) \right) \cdot \left(\eta - \mathbb{1}[Y < \phi_\eta(\mathbf{X})] \right) \right], \quad (6)$$

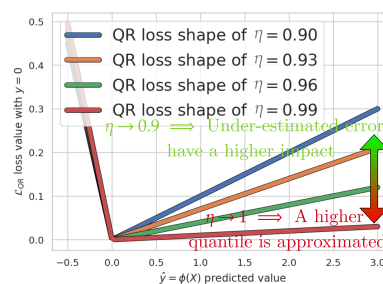
where $\mathbb{1}[c]$ denotes the indicator function that verifies the condition c .

Unlike MSE Eq 4 or MAE Eq 5, the QR expressed in Eq 6 is not always a symmetric function in the sense that when the predictive system over- or under-estimates it sums to the final loss value in the same manner.

This is illustrated in Figure 6 with the representation of a QR loss function shape centered at zero considering different quantile parameters, η s, i.e. $\{\mathcal{L}_{QR}(\mathbf{X}, Y, \eta_t)\}_{t=1}^9$ where $\eta_t = 0.1 \cdot t$, Y is always zero, the $\phi_\eta(\mathbf{X})$ in Eq 6 is the horizontal axis value and the vertical axis corresponds to the loss value in such conditions. As we can see in Figure 6, depending on



■ **Figure 6** QR loss function shape centered at zero. The $\phi_\eta(\mathbf{X})$ in Eq 6 is the horizontal axis and the vertical axis corresponds to the loss value in such conditions.



■ **Figure 7** Behaviour of the QR loss value for high η values. When $\eta \rightarrow 1$ (red case) the under-estimated predictions are multiplied by an almost zero factor, which produces flat shape in $\eta = 0.99$.

the selected quantile η for the QR, its shape will be different. For instance, when the quantile is the 10th percentile ($\eta = 0.1$) the underestimated errors are multiplied by a lower factor than when the forecaster value $\phi(\mathbf{X})$ is overestimating. This is so because the increasing loss value in the positive horizontal axis in such case is clearly higher than the negative part. This effect comes from the second multiplier of the Eq 6, i.e. $(\eta - \mathbb{1}[Y < \phi_\eta(\mathbf{X})])$, which means that when the real value Y is strictly lower than the predicted $\hat{y} = \phi_\eta(\mathbf{X})$, then the indicator function takes the value of 1, otherwise it is 0. In the presented case, as $\eta = 0.1$, it means that when $\phi_\eta(\mathbf{X})$ is underestimated, the difference between the real value and the predicted value will be multiplied by 0.1, which justifies the lower increasing in the negative part of the blue line of Figure 6 as far as the predicted value is from the (here, always zero) real value. Contrastingly, the positive part will be multiplied by -0.9 , which produces a higher increasing as much far as the predicted value is far from the real one but also it ensures that the loss function is always positive.

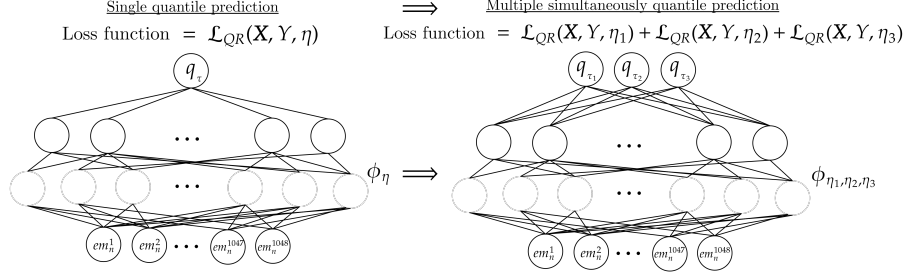
3.1 Predicting Sky-high Quantiles using QR

The problem at hand requires to have a proper sky-high quantile to ensure most of the predictions are below. However, when we use the QR Eq. 6 to predict a quantile $\eta \rightarrow 1$, i.e. when the condition $Y < \phi_\eta(\mathbf{X})$ is satisfied, the whole expression of Eq. 6 tend to be zero due to its second factor $(\eta - \mathbb{1}[Y < \phi_\eta(\mathbf{X})])$ approximates to zero. This is an issue because it implies that any overestimated point by the NN ϕ_η for a certain $\eta \approx 1$ almost does not contribute to the expected error. Hence, higher erroneous values that are overestimated are neglected as lower erroneous values. This has a critical effect in the optimization process because high differences, $Y - \phi_\eta(\mathbf{X})$, will not be taken into account and it will cause the solution to be unavoidably unstable or wrong.

To solve this issue we propose a solution that considers two edges (represented in two colors of the vertical arrow of Figure 7): First, we want to predict the higher quantile possible to reduce the number of under-estimated cases. And second, we want to avoid the neglecting issue that appears when we are predicting $\eta \rightarrow 1$ quantiles. Therefore, (1) the NN model will predict simultaneously several quantiles (including farther and closer quantiles to 1), and (2) all these quantiles will be linearly related with a common previous hidden representation, which means that they will share the last neural network layer.

As we described previously, the closer the quantile value η gets to 1, the worse the effect of avoiding overestimated errors will be. Therefore, (1) considering several (a fixed set of) quantiles that tends to 1 and (2) that preserves a linear relation between a common previous representation (i.e. all the simultaneously predicted quantiles shares the same penultimate

NN layer), we can ensure that lower sky-high quantiles of the set will avoid the neglecting issue when it appears and, at the same time, the higher approximated quantiles will try to push for obtaining a higher extreme upper bound. Combinedly, (1) and (2) allow the model to obtain a balanced solution in both senses. In Figure 8 is described the change we require to perform to predict several quantiles using a single NN model.



■ **Figure 8** Transformation from a single QR NN model to a multiple QR NN model that simultaneously predicts several quantiles.

As shown in Figure 8, the number of inputs and even the internal number of hidden layers and hidden neurons are preserved (as long as it is enough complex to approximate the desired function). The only we need to change is the number of outputs, represented as the $\{q_{\eta_o}\}_{o=1}^3$ last neurons in Figure 8, by changing the number of neurons of the last layer. Each of these neurons will be optimized using a different specific QR-loss function, shown in Eq. 6, for the corresponding quantile η value.

3.2 Task Order Invariance

We set an additional constraint on our multicore contention prediction models that typical ML models do not provide. In particular, the predicted contention for a given task must be the same under any permutation of its contenders.

► **Property 3** (Task Order Invariance (TOI)). *For homogeneous multicores, multicore contention models for a given task τ_i in a given workload n must provide the same estimate ($P\Delta_{n,i}$) regardless of the core where τ_i runs and any permutation of its contenders.*

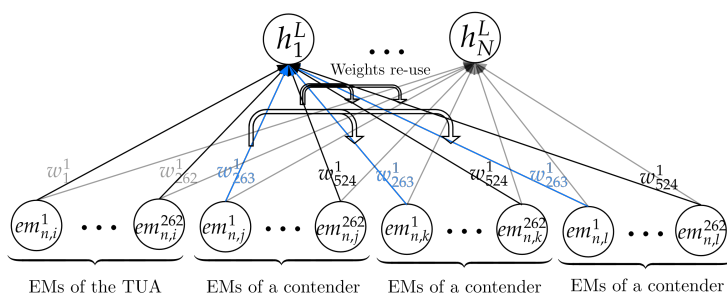
Considering a workload n consisting in tasks τ_i , τ_j , τ_k , and τ_l , the contention suffered by each of these tasks must not be affected by the core in which tasks executes. Therefore, it must not be affected where the task under analysis and the other tasks in the workload are. Specifically, for a given task under analysis (TUA) τ_i in a certain workload n :

$$\begin{aligned}
 P\Delta_{n,i} &= f(EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l}) = f(EM_{n,j}, EM_{n,i}, EM_{n,k}, EM_{n,l}) = \\
 &= \dots = f(EM_{n,l}, EM_{n,k}, EM_{n,j}, EM_{n,i})
 \end{aligned}$$

► **Definition 4** (Task Order Invariant). *Given a four-core multicore contention forecaster ϕ and four sets of EMs, $\{EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l}\}$ where τ_i is the TUA, this forecaster can be considered a Task Order Invariant (TOI) predictor if Equation 7 holds regardless of the order in which EM sets appear in the parameter list, which means disregarding the core mapping of both the TUA and the contenders, as long as they run in parallel.*

$$\phi(EM_{n,i}, EM_{n,j}, EM_{n,k}, EM_{n,l}) = \phi(EM_{n,l}, EM_{n,k}, EM_{n,j}, EM_{n,i}) \quad (7)$$

TUA position can change.
Same contenders with different order.



■ **Figure 9** Example definition of the first layer of the NN or the whole MLR model for the T2080 to ensure they provide the same output regardless the order of the contenders. In short for all the $l = 1, \dots, O$ output neurons of the L -layer, $w_{j+1*262}^l = w_{j+2*262}^l = w_{j+3*262}^l$. Note that $w_j^l, j = 1, \dots, 262$ are the EMs of the TUA in this case τ_0 .

3.2.1 Existing Models

For MLR, in Eq. 2 the set of K weights are divided in C groups/cores of J EM each and the j -th position for each group is the same EM (e.g. data cache misses). Each of these EM has associated a different weight $w_i, w_{i+J}, \dots, w_{i+(C-1) \cdot J}$, which can get a different value as part of the training. It then follows that the order in which the contenders tasks are passed to the model affect its results. Note that the j -th counter of task τ_i in workload n (i.e. $em_{i,n}^j$) is the $j \cdot (i - 1)$ -th counter in the workload ($em_n^{j \cdot (i-1)}$) for $j = 1, \dots, J$ and $i = 1, \dots, C$.

For NN, as it can be in in Figure 4, the same logic applies. In the first layer, $h_k^0 = em_n^k$ and there is a different weight associated to each h_k^0 with each of then potentially taking a different value. As a result, the order of the contender tasks matters.

As an illustrative example, Figure 10a shows the contention estimates obtained with the NN model for different permutations of the 3 contenders (C_1, C_2 , and C_3) for 5 arbitrary workloads. We can see that depending on the of the contenders (shown in the x-axis) the produced estimate varies showing that NN does not fulfill Property 3 (nor does MLR). Also, across permutations the predictions can vary significantly.

3.2.2 Achieving TOI

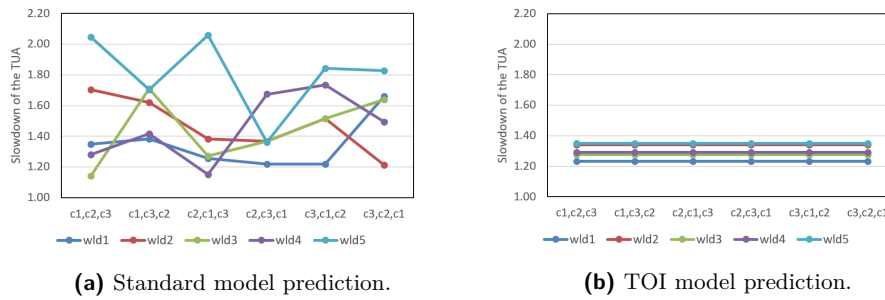
In order to achieve TOI, we propose a method that can be commonly applied to any presented NN model as well as the MLR. The method consists in sharing the weights across the same EM in all cores where contenders run, as it is shown in Figure 9. Particularly, the TOI dense layer will satisfy the following expression (τ_i is the TUA in workload n):

$$h_l^L = \sigma \left(w_1 \cdot em_{n,i}^1 + w_2 \cdot em_{n,i}^2 + \dots + w_J \cdot em_{n,i}^J + \sum_{k=1}^3 (w_{J+1} \cdot em_{n,k}^1 + w_{J+2} \cdot em_{n,k}^2 + \dots + w_{2 \cdot J} \cdot em_{n,k}^J) \right), \quad (8)$$

↑ The new weight-sharing part to be TOI.

where the shaded area is the new weight-sharing part and σ is the non-linear function or activation function introduced in Section 2, such as the REctified Linear Unit (RELU). Importantly, when we want to produce a TOI MLR, this activation function does not appear and, therefore, the Eq. 8 without the σ constitutes the overall model instead of a single layer like in the NN case.

4:14 Quantile NN for Multicore Contention Prediction



■ **Figure 10** Contention predictions of a standard model with and without the TOI for a given workload under different contender permutations.

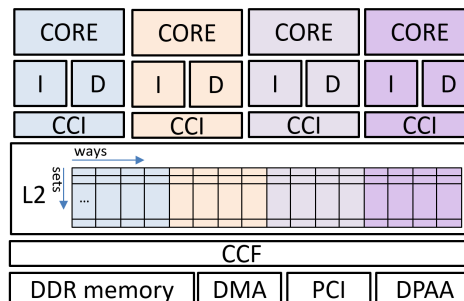
For the same workloads and contender permutations showed in Figure 10a, we produce predictions with a NN in which the weight of the same EM in the different cores is shared. The net result is that contention predictions are invariant to the permutations of the contenders as shown in Figure 10b.

4 Experimental Setup

Our experimental setup includes the target platform and its configuration (Section 4.1), the kernels we use to compose workloads (Section 4.2), the particular experiments we carried out (Section 4.3), and the configuration used for the specific contention models, such as the number and type of layers in NN and QRNN (Section 4.4).

4.1 Hardware Platform

We perform our experiments on a NXP T2080 Reference Design Board [22]. The T2080RDB includes a NXP T2080 System on Chip (SoC) for which an avionics multi-core certification case has been started [48]. The T2080 SoC includes a CPU cluster with 4 e6500 cores [21], see Figure 11. Each core has its own private 32KB 8-way instruction and data caches. In each core a core-cluster interface (CCI) serves as the bridge for data and instruction cache requests from and to the L2. The L2 cache is shared between all the cores. The core cluster, the DDR memory controller, the DMA and other I/O controllers are connected via the CoreNet coherence fabric (CCF). In this work, we focus on the main path from cores to main memory, and do not address the potential contention arising in the I/O. In order to favor time predictability, we configure the T2080 as follows:



■ **Figure 11** Simplified block diagram of the NXP T2080 SoC.

- *Shared L2 cache*: shared caches are one of the main sources of contention in modern SoC. In order to favor predictability and simplify the work of timing analysis tools in the validation phase, shared last level caches are typically partitioned via software (set partitioning) or via hardware support (way partitioning). The T2080 allows each core to be assigned a subset of the L2 cache ways. In our experiments, we assign each core a disjoint set of 4 ways by properly configuring the L2 cache control registers.
- *Hyper-threading support*: the hyper-threading capability in each e6500 core has been deactivated. From a multicore contention perspective, tasks running in hyper-threading mode in a core share not only first level instruction and data caches but also some core resources, potentially affecting each others performance significantly.

4.2 Workloads

We use several kernels (basic operators) that are commonly used in machine learning libraries, which in turn, are used for many operations of autonomous driving and ADAS software, from perception and detection to planning and control. For instance, matrix multiplication is a central element of YOLOv3 machine learning library [53] and radar applications [23, 54], and has been shown to account in some scenarios for 67% of YOLO’s execution time [20]. The kernels we use in this work are:

- Matrix Multiplication is one of the most common kernels for many functionalities like object detection and path planning in autonomous navigation, and covariance matrix computation in radar applications. We experimented with two versions: (1) basic (MMB), and (2) optimized (MMO), which “tiles” input matrices to improve data locality.
- (3) Matrix Transpose is another quite common matrix operator
- (4) Matrix Transpose Multiply combines matrix transpose of the second matrix and multiplication of both of them. It is used, for instance, for certain internal operations in NN [24] and for covariance matrix computation in radar applications;
- (5) Rectifier is an activation function in neural networks taking the positive value of its argument or zero when it is negative;
- (6) Image-to-Columns function is used for transforming raw RGB images into matrices in the format needed by neural networks;
- (7) Vector-multiply-add is a type of linear algebra operator.

We also used a set of basic operators with different data types and precisions. In particular, we use (8-9) vector addition with integer long and with floating-point double precision. For (10-11) vector multiplication and (12-13) vector division we also use integer long and fp double types. We also use (14) quicksort sorting algorithm on a randomly ordered array. All these operators are the building blocks for other basic functionalities in machine learning libraries and radar applications.

Also autonomous driving frameworks like Apollo use deep and recurrent neural networks in several stages like object detection, object tracker, etc [47]. Each of those stages works with different input sizes. In the same vein, radar applications typically operate relatively small matrices in comparison to camera-based and LiDAR-based object detection applications. In order to capture this scenario in which input data may or may not fit in the different cache levels, we have developed 3 variants of our kernels: one fitting in DL1, one fitting in 4-ways of the L2, and one going frequently to memory. Overall, we use $A = 42$ kernel variants.

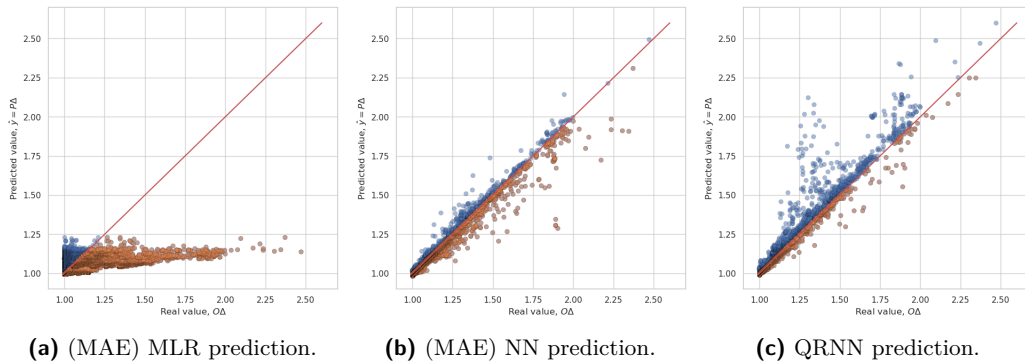
4.3 Experiments

We start by running each kernel in isolation and collect all EMs. EMs are read via performance monitoring counters. While the number of EMs can easily be over hundred, the number of performance monitoring counters is usually below 8 (it is 5 for the T2080). Hence, in order to read all 262 EM, we carried out 53 runs in each of which we read 5 different EMs.

As a second step, we generated a data set with 4-kernel workloads randomly selected from the set of kernels described in the previous section. We run each workload on the target board. For the T2080, it has been reported that the same multicore run is subject to execution time variation [57]. This happens despite exercising a tight control on the experimental setup ensuring that in every run the state of the caches and TLBs is reset. However, other non-resettable resources retain some state that changes across runs. To capture this variability, we repeated each experiment run 50 times and take the high watermark execution time. Note that our experiments show that this variability occurs for multicore executions. For single-core executions, the variability of a repeated measured EM is below its 1% value.

In each modeling experiment, we randomly split the DS into different (sub)sets used to train and validate the models, as commonly done in machine learning literature [15, 44]. The Training set (T) includes the subset of data that will be used to optimize the supervised model. The Validation set (V) includes the subset of data that will be used to decide when to stop optimizing the supervised model. The tEst set (E) includes the subset of data that will be used to verify the quality of the performance or accuracy of the supervised model to generalize.

In all experiments, the percentage of workloads of the overall DS for T and V is 17% and 2%, respectively. The remaining 81% is used as E. Note that only T and V are used to determine the models, while the remaining E of the DS is used in this work to show the accuracy of each model and that hence will not be needed in reality to generate the model. It is also worth noting that when generating the T and V sets, we make sure that the number of times each kernel is used in as TUA is the same. The contenders are generated randomly.



■ **Figure 12** $P\Delta$ vs $O\Delta$ for MLR, NN, and QRNN.

4.4 Model Configuration and Libraries

The current forecasting context is a single value regression. In the presented problem, no time-based input information exists and, therefore, it is not required to encode it into the model by using recurrent NN layers [27, 60]. Similarly, the EMs used as inputs are mostly counters with their own meaning, hence not having *spatial proximity information*.

Consequently, convolutional NN layers [35, 2], which are specially designed for images or text information, are not appropriate in our context. Hence, the considered NN models used are based on fully-connected or dense layers [26], which is a Multi-Layer Perceptron (MLP) model with the ReLU non-linear activation function [33] in the hidden layers. Additionally to the common MLP, all the models presented in this work have the first layer customized by following Section 3.2 to ensure TOI.

Regarding implementation, all the models were developed in TensorFlow [1] using the Keras sub-library [12]. We used a greedy search algorithm [37, 30] to select a proper NN architecture including between 1 and 6 hidden layers considering 100, 200, 300, 400 and 500 neurons. Each model is trained with early stopping, hence requiring different time to train, but none exceeds 10 minutes of training. The final selected architecture includes 3 layers of 300 neurons each (including the TOI-layer) for the standard NN (with a single output), and the same architecture for the QRNN model. Therefore, given that both hidden architectures are analogous for those NN models (except in the case of the last layer), they have similar general function approximation capabilities [26].

5 Experimental Results

We start by comparing the accuracy of the estimates provided by each technique. In Figure 12 we see three charts corresponding to the accuracy results of MLR, NN, and QRNN with a target quantile⁴ $\eta = 0.9$. Each point represents a particular workload n with TUA τ_i . The x-axis shows the slowdown observed for the TUA $O\Delta_{n,i}$ and the y-axis the predicted contention $P\Delta_{n,i}$. The bottom-left top-right diagonal highlighted with a red line shows the ideal scenario in which the predicted value matches the observed one. As we can see MLR (Figure 12a) underestimates (i.e. $P\Delta_{n,i} < O\Delta_{n,i}$) for many workloads, see points below the red line. NN (Figure 12b) produces much tighter estimates, yet many of them underestimated. QRNN (Figure 12c) corrects this situation significantly reducing the number of underestimated cases while maintaining high-accuracy.

This is quantified in Figure 13 where we see that MLR underestimates in 46.8% of the cases, NN 31.3% and QRNN reduces it down to 8.8%. In terms of amount of over- and underestimation, Figure 14 shows x-th largest overestimated and underestimated values (referred to as the x-th LOE and LUE value, respectively) of each model. For instance, the (1st) largest overestimated (LOE) value is largest value of $P\Delta_{n,i}/O\Delta_{n,i}$ when $P\Delta_{n,i} > O\Delta_{n,i}$. Likewise the (1st) largest underestimated (LUE) value is the one with the (1st) largest distance to value 1.0 when $P\Delta_{n,i} < O\Delta_{n,i}$, i.e. the (1st) smallest value.

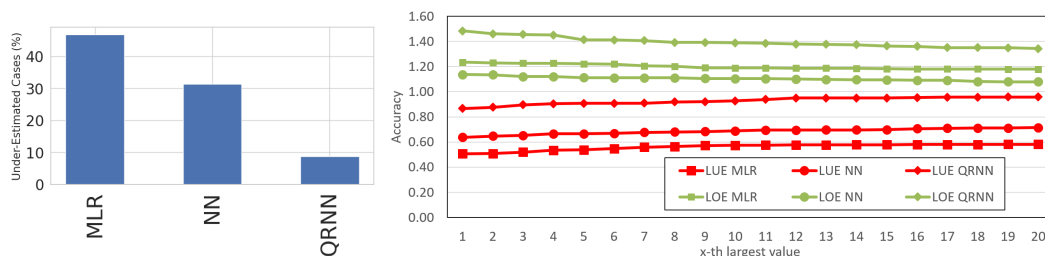


Figure 13 Breakdown.

Figure 14 Largest Over- and Underestimation values.

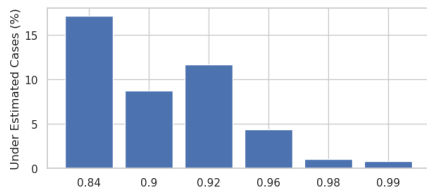
⁴ As it is described in Section 3.1, the proposed QRNN model predicts three different quantiles, $\{0.7, 0.9, 0.99\}$, to support the $\eta = 0.9$ prediction and avoid negligence issues of sky-high quantiles.

In terms of LUE, MLR (red squares in Figure 14) produces the worst results with underestimates below 0.6 even for the 20th largest value. NN (red circles in Figure 14) produces similar underestimates that remain below 0.7 for the 20th LUE. Instead, QRNN (red diamonds in Figure 14) reduces this significantly with underestimation very close to 1.0: at most 0.83 and rapidly going to 0.96 for the 20th LUE.

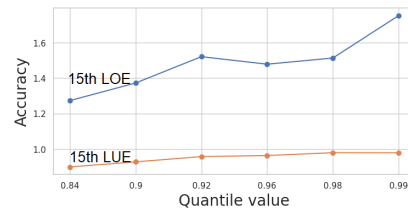
In terms of LOE, QRNN produces worse results than MLR and NN. Yet, the LOE values are moderate going from 1.48 (largest) to 1.34 (20th largest). It follows that, QRNN fulfills Property 2 by tending to overestimation while keeping estimates tight.

5.1 Impact of η

In order to assess the impact of η in the results besides the value used so far $\eta = 0.9$, we evaluate other values of η . In particular, inspired in the analysis of the distributional tails, we select an exponential decay as follows $\{\eta_{it} = 1 - 0.01 * 2^{it}\}$, $it \in [1, 2, 3, 4]$, which takes values from 1 to 0.8, i.e. $\eta \in [0.99, 0.98, 0.96, 0.92, 0.84]$. Figure 15 and Figure 16 evaluate underestimated cases and $x = 15$ -th LUE and LOE for different values of η (similar trends are obtained for other values of x like 10 and 20). We can see that, as η increases, the number of under underestimated cases tends to decrease from over 16% for $\eta = 0.8$ to less than 1% for $\eta = 0.99$. In terms of LUE and LOE, both increase. In the LUE case, this means reducing the underestimation and in the LOE case increasing overestimation. Overall, changing the quantile η , $QRNN_{\eta}$ provides to the end user a mechanism to control over- and underestimation in the way it better adapts to his/her needs.



■ **Figure 15** Underestimated cases.

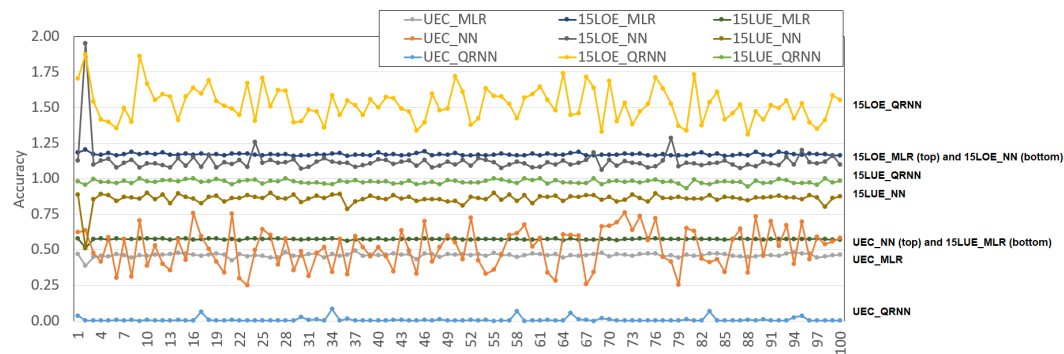


■ **Figure 16** 15th LOE and LUE values.

5.2 Different random partitions of the DS

Results so far have been shown for a particular breakdown of the workload space into TVE sets. For $\eta = 0.99$ Figure 17 shows the results for 100 experiments in terms of the number of underestimated cases (UEC) and the 15th LOE and LUE values (15LOE and 15LUE, respectively). In each of the 100 experiments we randomly selected the workloads in TVE sets as described in Section 4.3 and re-trained all models.

As shown, there is some variability due to the fact that, in the DS across the different workloads, the randomly generated contenders do not properly represent the tasks in the DS. This is illustrated in test number 2, for which we see large variations for NN results in high 15LOE and 15LUE. Despite these variations, more notable in 15LOE for QRNN and UEC for NN, the main conclusions remain the same with QRNN tending to overestimation and almost no underestimation (15LUE is very close to 1 and UEC to 0 for QRNN).



■ **Figure 17** Number of over-estimated cases and the 15th LOE and LUE value for 100 experiments.

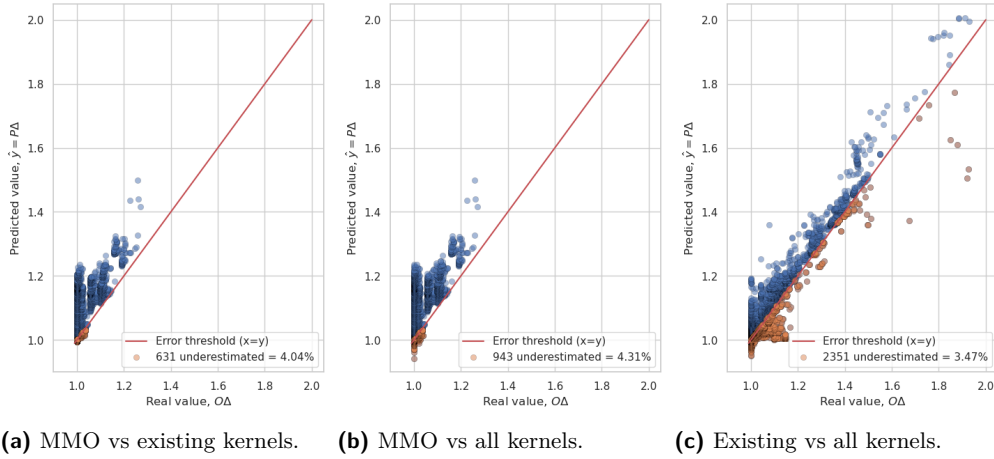
5.3 Changes to the applications

In the application scenario we address, as introduced in Section 2, the applications composing the software component of the embedded real-time product are fixed. However, as part of the usual incremental development process, applications can suffer some updates. This might cause some improvements to the functional behavior of the application, and more importantly to us, it can change the usage of hardware shared resources. Obviously, the more the updated versions differ in their resource usage from previous versions, the more challenging it is to produce tight estimates. In order to capture this situation, we assess the impact of varying some of the applications in the task set. In particular, we consider a scenario in which the QRNN model has been trained with a basic version of matrix multiplication (MMB) that is afterwards optimized resulting in an optimized version of matrix multiplication (MMO). This implies having a holdout set (H) for QRNN, which is an isolated subset of data that will be used to check the capabilities of the model to predict scenarios that could not be observed or are slightly different.

In practical terms, this means that we remove MMO (only use MMB) from TVE so that the weights of the resulting QRNN do not factor in MMO (but MMB). To assess the impact on accuracy, we query QRNN with MMO, so we make contention predictions for MMO, which was not used in TVE. Figure 18a shows the results when we use as contenders of MMO only kernels already used in TVE and in Figure 18b when we also use MMO as contender. In both cases QRNN behaves quite well, keeping both, the number of underestimated cases low and the overall prediction accuracy quite tight. Figure 18c shows the results of the predictions for the kernels already in TVE (that is, all but MMO) when the set of contenders contain at least one copy of MMO. The same trend holds with low number of overestimations (below 3.5%) while accuracy is kept high.

5.4 Execution time requirements

In order to assess the speed of the inference of the models, we perform experiments with TensorFlow software library v2.3.0 on an AMD Ryzen 9 3950X Processor. Our results show that MLR performs 1.25×10^7 predictions per second while NN and QRNN 1.48×10^5 and 1.46×10^5 per second respectively when running the library in a single core. While MLR is faster, we have seen that its accuracy results are rather poor. When we use all 16 cores, performance for NN and QRNN scales perfectly so that we can make more than 2.3×10^6 predictions per second. Overall, a wide design space can be covered with the presented QRNN model, hence achieving the Property 1.



■ **Figure 18** QRNN predictions when MMO is used instead of MMB.

6 Related Works

The importance of capturing timing requirements already in DDP of embedded real-time systems is widely recognized [17, 5, 42, 19, 18, 49]. Existing approaches mainly focus on deriving and exploit early timing bounds to guide architectural exploration and steer design decisions: early WCET figures are obtained either by exploiting simulation of software designs or partial implementations thereof [17, 18, 3], or by using actual observations to abstract away from certain architectural features [42, 19]. In fact, the main objective of these works is providing quick estimates at the expense of loosening accuracy. However, those works do not address the impact of multicore contention, which is the main focus of our work.

ML techniques are widely used in several fields of computer science [46, 41], for the design, optimization, and simulation of computer systems. In the context of time critical systems, approaches building on statistical and machine learning techniques have been proposed to model uncertainties in deriving timing bounds for tasks running in single cores, both in early and late development stages. In [5, 25] statistical approaches are leveraged to model uncertainties in timing estimation rather than predicting WCET figures, but do not apply to multicore systems. Still on single core systems, a hybrid approach using ML to build the timing model within a standard static WCET analysis framework has been proposed in [4].

Preliminary approaches for deriving early WCET estimates based on machine learning techniques are proposed in [6, 28], where relevant code-level constructs such as arithmetic and memory operations, and conditionals, are used to train simple regression models for WCET computation. Our approach builds on hardware events rather than source code, and focuses on predicting multicore contention instead.

The use of machine learning techniques to model the impact of multicore contention has been mainly investigated from the perspective of high-performance systems in the mainstream domain [52, 11, 61, 59, 50]. These works aim at preventing average performance degradation and implement linear regression models to predict the impact of multicore contention. In the scope of real-time systems, non-linear regression with *random forest* has been recently assessed for predicting multicore interference [14]. In contrast with our work, the proposed approaches are mainly oriented towards average performance estimation and in all cases, the underlying models do not prioritize overestimation as a fundamental requirement to avoid timing misconfiguration to arise in the later development stages. Instead, we introduce the use of QRNN for improved and tunable – conservative – prediction accuracy.

Few works address the need for conservative timing estimates in DDP from the perspective of real-time systems. Early modeling of multicore contention in time-critical systems is addressed in [19, 49], where an empirical approach is presented to capture the worst-case impact of contention on a given platform, which is later used to inflate the execution time of a task in isolation, to obtain a bound guaranteed to hold under any workload. While closer in spirit to our method, the inflated execution time estimates can result in up to 20x bigger than programs' execution time in isolation even for 4-core setups, which makes them barely useful for DDP exploration. The work in [34] proposed an NN approach for deriving early WCET bounds using the program features collected at the source code level by applying static WCET analysis methods. We instead build on hardware events to model multicore contention and use QRNN to force accurate but conservative timing estimates.

7 Future Work

In terms of future work, we identify several research opportunities. First, we have used all the EMs available in the underlying platform. However, while EMs provide insightful information about the activities in the processor, a subset of them could suffice to capture the most relevant factors affecting multicore contention. In this line, techniques like principal component analysis could be used for selecting relevant EMs, allowing a dimensionality reduction of the contention models and therefore faster and more accurate models.

It is also the case that, so far, we have used EM collected during the execution of each application in isolation. In fact, contention models could also build on EMs collected during the execution of a subset of the workloads, as this would provide more accurate information about how a given application reacts to contention. The other side of the coin is that experimentation time would increase and training would be more complex. We are interested in exploring trade-offs between accuracy and complexity.

Focusing on the bigger picture, contention models are to be queried by system-level optimization models to explore, for instance, different task schedules based on the expected contention. System-level optimizers require modeling how tasks overlap in time and how events are distributed within each task execution. The latter aspect may call for collecting EMs within tasks phases rather than end to end. Still at system level, contention models can also be extended to cover other devices beyond memory for which activity descriptors – e.g. in the form of EMs or system-level metrics – are available.

Finally, the present article focuses on NN models, which are just one of the state-of-the-art ML models for regression purposes. NN models are not the only ones designed to learn a conditional quantile using QR. For instance, decision trees [39], random forests [38, 14] or Gradient Boosting [56] methods can be used with analogous purposes. As future work, a comparison between extra QR-based models can be performed to assess the functional approximation capabilities of each model in the current forecasting problem.

8 Conclusions

Early contention estimates in multicore setups tightly upper-bounding real contention reduce the risk to detect timing misconfiguration in late phases of the development process that would result in costly changes to the system design and/or implementation. We use quantile-regression neural networks (QRNN) as an alternative to common NN and multi-linear regression (MLR) models to drastically decrease scenarios with contention underestimation while preserving tightness. Moreover, our approach achieves task order

independence to provide identical contention estimates for equivalent task permutations with identical contention in practice. Both, tendency towards – tight – overestimation and task-order independence are, in our view, fundamental properties for the use of contention models in real-time systems, besides prediction speed. Our results show that QRNN consistently reduces the number of underestimated contention bounds with respect to NN and MLR while its η parameter allows the user to find the tradeoff that fits best his/her needs.

References

- 1 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. doi:<http://10.5281/zenodo.4724123>.
- 2 Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017. doi:[10.1109/ICEngTechnol.2017.8308186](https://doi.org/10.1109/ICEngTechnol.2017.8308186).
- 3 Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, 2016. doi:[10.1007/s11241-016-9250-7](https://doi.org/10.1007/s11241-016-9250-7).
- 4 Abderaouf Nassim Amalou, Isabelle Puaut, and Gilles Muller. WE-HML: hybrid WCET estimation using machine learning for architectures with caches. In *RTCSA 2021 - 27th IEEE International Conference on Embedded Real-Time Computing Systems and Applications*, pages 1–10, Online Virtual Conference, France, August 2021. IEEE. URL: <https://hal.inria.fr/hal-03280177>.
- 5 Jakob Axelsson. A method for evaluating uncertainties in the early development phases of embedded real-time systems. In *RTCSA*, 2005. doi:[10.1109/RTCSA.2005.12](https://doi.org/10.1109/RTCSA.2005.12).
- 6 Armelle Bonenfant, Denis Claraz, Marianne de Michiel, and Pascal Sotin. Early WCET Prediction Using Machine Learning. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASISs)*, pages 5:1–5:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:[10.4230/OASISs.WCET.2017.5](https://doi.org/10.4230/OASISs.WCET.2017.5).
- 7 James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: <http://github.com/google/jax>.
- 8 Axel Brando, Joan Gimeno, Jose A Rodríguez-Serrano, and Jordi Vitrià. Deep non-crossing quantiles through the partial derivative. *International Conference on Artificial Intelligence and Statistics*, 2022. URL: <https://proceedings.mlr.press/v151/brando22a.html>.
- 9 Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.
- 10 Tianfeng Chai and Roland R Draxler. Root mean square error (rmse) or mean absolute error (mae)?—arguments against avoiding rmse in the literature. *Geoscientific model development*, 7(3):1247–1250, 2014. doi:[10.5194/gmd-7-1247-2014](https://doi.org/10.5194/gmd-7-1247-2014).
- 11 Yuxia Cheng, Wenzhi Chen, Zonghui Wang, and Yang Xiang. Precise contention-aware performance prediction on virtualized multicore system. *Journal of Systems Architecture*, 72:42–50, 2017. Design Automation for Embedded Ubiquitous Computing Systems. doi:[10.1016/j.sysarc.2016.06.006](https://doi.org/10.1016/j.sysarc.2016.06.006).
- 12 François Chollet. Keras, 2015. URL: <https://github.com/fchollet/keras>.

- 13 Francois Chollet. *Deep Learning with Python*. Manning Publications Co., USA, 1st edition, 2017. doi:doi/10.5555/3203489.
- 14 Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 246–259, 2019. doi:10.1109/RTSS46320.2019.00031.
- 15 Harris Drucker, Christopher J Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. *Advances in neural information processing systems*, 9, 1996. doi:10.5555/2998981.2999003.
- 16 Oliver Duerr, Beate Sick, and Elvis Murina. *Probabilistic Deep Learning: With Python, Keras and TensorFlow Probability*. Manning Publications, 2020. URL: https://tensorchiefs.github.io/dl_book.
- 17 Raimund Kirner et al. Fully automatic worst-case execution time analysis for matlab/simulink models. In *ECRTS*, 2002.
- 18 Trevor Harmon et al. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Trans. Industrial Informatics*, 8(2), 2012.
- 19 C Ferdinand, R Heckmann, D Kästner, K Richter, N Feiertag, and M Jersak. Integration of code-level and system-level timing analysis for early architecture exploration and reliable timing verification. In *ERTS2 2010, Embedded Real Time Software & Systems*, 2010.
- 20 Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176. IEEE, June 2017. doi:10.1109/dsn-w.2017.47.
- 21 Freescale semiconductor. e6500 Core Reference Manual. <https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf>, 2014. E6500RM. Rev 0. 06/2014.
- 22 Freescale semiconductor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.
- 23 Jonah Gamba. *Automotive Radar Applications*, pages 123–142. Springer Singapore, Singapore, 2020. doi:10.1007/978-981-13-9193-4_9.
- 24 Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.
- 25 P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 580–588, 2001. doi:10.1109/DATE.2001.915082.
- 26 Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: <http://www.deeplearningbook.org>.
- 27 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- 28 Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASISs)*, pages 5:1–5:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASISs.WCET.2018.5.
- 29 Javier Jalle, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Bounding resource-contention interference in the next-generation multipurpose processor (ngmp). In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS²)*, 2016.
- 30 Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019. doi:10.48550/arXiv.1806.10282.

- 31 Alex Kendall and Yarín Gal. What uncertainties do we need in bayesian deep learning for computer vision? *Neural Information Processing Systems, 2017*, pages 5580–5590, 2017.
- 32 Roger Koenker and Kevin F Hallock. Quantile regression. *Journal of economic perspectives*, 15(4):143–156, 2001. doi:10.1257/jep.15.4.143.
- 33 Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.
- 34 Vikash Kumar. Deep neural network approach to estimate early worst-case execution time. In *Proceedings of Digital Avionics Systems Conference (DASC)*, 2021.
- 35 Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
- 36 Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015. URL: <https://www.nature.com/articles/nature14539>.
- 37 Guohao Li, Guocheng Qian, Itzel C Delgadillo, Matthias Muller, Ali Thabet, and Bernard Ghanem. Sgas: Sequential greedy architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1620–1630, 2020.
- 38 Nicolai Meinshausen. Quantile regression forests. *Journal of Machine Learning Research*, 7:983–999, December 2006. doi:10.5555/1248547.1248582.
- 39 AV Meshcheryakov, VV Glazkova, SV Gerasimov, and IV Mashechkin. Measuring the probabilistic photometric redshifts of x-ray quasars based on the quantile regression of ensembles of decision trees. *Astronomy Letters*, 44(12):735–753, 2018. doi:10.1134/S1063773718120058.
- 40 Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.
- 41 Atul Negi and K Rajesh. A review of ai and ml applications for computing systems. In *2019 9th International Conference on Emerging Trends in Engineering and Technology - Signal and Information Processing (ICETET-SIP-19)*, pages 1–6, 2019. doi:10.1109/ICETET-SIP-1946815.2019.9092299.
- 42 Stefana Neno and Daniel Kastner. Worst-case timing estimation and architecture exploration in early design phases. In *In Niklas Holsti, editor, 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik*, 2009.
- 43 Jan Nowotzsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS*, 2014.
- 44 Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, David Sculley, Sebastian Nowozin, Joshua Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can you trust your model’s uncertainty? evaluating predictive uncertainty under dataset shift. *Advances in neural information processing systems*, 32, 2019.
- 45 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch, 2017.
- 46 Drew Penney and Lizhong Chen. A survey of machine learning applied to computer architecture design. *ArXiv*, abs/1909.12373, 2019. arXiv:1909.12373.
- 47 Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 48 David Radack, Harold G. Tiedeman, and Paul Parkinson. Civil certification of multi-core processing systems in commercial avionics. Technical report, Rockwell Collins, 2018.
- 49 Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–25, 2012.
- 50 Jitendra Kumar Rai, Atul Negi, and Rajeev Wankar. Machine learning based performance prediction for multi-core simulation. In Chattrakul Sombattheera, Arun Agarwal, Siba K. Udgata, and Kittichai Lavangnananda, editors, *Multi-disciplinary Trends in Artificial Intelligence*, pages 236–247, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

- 51 Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999. doi:10.7551/mitpress/4937.001.0001.
- 52 Shenyuan Ren, Ligang He, Junyu Li, Zhiyan Chen, Peng Jiang, and Chang-Tsun Li. Contention-aware prediction for performance impact of task co-running in multicore computers. *Wireless Networks*, February 2019. doi:10.1007/s11276-018-01902-7.
- 53 Hamid Tabani, Roger Pujol, Jaume Abella, and Francisco J. Cazorla. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 144–145. IEEE, May 2020. doi:10.1109/isorc49007.2020.00030.
- 54 Lee Teschler. The basics of automotive radar, 2019. URL: <https://www.designworldonline.com/the-basics-of-automotive-radar/>.
- 55 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, 2016.
- 56 Jasper Velthoen, Clément Dombry, Juan-Juan Cai, and Sebastian Engelke. Gradient boosting for extreme quantile regression. *arXiv preprint*, 2021. arXiv:2103.00808.
- 57 Sergi Vilardell, Isabel Serra, Roberto Santalla, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. HRM: merging hardware event monitors for improved timing analysis of complex mpsoes. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(11):3662–3673, 2020. doi:10.1109/TCAD.2020.3013051.
- 58 Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005. URL: <https://www.jstor.org/stable/24869236>.
- 59 Felipe Vieira Zacarias, Rajiv Nishtala, and Paul Carpenter. Contention-aware application performance prediction for disaggregated memory systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers, CF '20*, pages 49–59, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387902.3392625.
- 60 Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint*, 2014. arXiv:1409.2329.
- 61 Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1443–1456, May 2016. doi:10.1109/TPDS.2015.2442983.

A Formal Link Between Response Time Analysis and Network Calculus

Pierre Roux ✉ 🏠 

ONERA, Toulouse, France

DTIS – Université de Toulouse, F-31055 Toulouse, France

Sophie Quinton ✉ 🏠 

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France

Marc Boyer ✉ 🏠 

ONERA, Toulouse, France

DTIS – Université de Toulouse, F-31055 Toulouse, France

Abstract

Classical Response Time Analysis (RTA) and Network Calculus (NC) are two major formalisms used for the verification of real-time properties. We offer mathematical links between these two different theories. Based on these links, we then prove the equivalence of various key notions in both frameworks. This enables specialists of both formalisms to get increase confidence on their models, or even, like the authors, to discover errors in theorems by investigating apparent discrepancies between some notions expected to be equivalent. The presented mathematical results are all mechanically checked with the interactive theorem prover Coq, building on existing formalizations of RTA and NC. Establishing such a link between NC and RTA paves the way for improved real-time analyses obtained by combining both theories to enjoy their respective strengths (e.g., multicore analyses for RTA or clock drifts for NC).

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Networks → Formal specifications; Software and its engineering → Formal methods; General and reference → Verification

Keywords and phrases Response Time Analysis, Network Calculus, dense time, discrete time, response time, formal proof, Coq

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.5

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.8.1.3>

Funding This work has been partially supported by the French national research organization ANR (grant ANR-17-CE25-0016) through the RT-PROOFS project.

1 Introduction

Classical Response Time Analysis (RTA) and Network Calculus (NC, together with its variant Real-Time Calculus) are two major formalisms used for the verification of real-time properties. Some of the differences between RTA and NC are well-known: RTA tends to be based on discrete time while NC relies on dense time, there is no notion of task in NC, etc. Still, fully understanding the implications of such differences – enough, for example, to be able to compare the state of the art in both approaches – requires a solid expertise in both formalisms, which very few people have. To the best of our knowledge, no formal link has ever been proposed to relate models and verification techniques from both worlds. This is now made easier by recent work on formalizing each technique separately using the Coq interactive theorem prover: RTA in Prosa [9] and NC in NCCoq [22].



© Pierre Roux, Sophie Quinton, and Marc Boyer;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio; Article No. 5; pp. 5:1–5:22



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



In this paper, we provide the foundations for a formal connection between RTA and NC. Specifically, we show how to translate the behavior of a real-time system as formalized in Prosa into a trace as specified in NCCoq, and conversely. This requires in particular a clean formalization of time and how to switch back and forth between discrete and dense time. We also relate core modeling concepts of NC, namely arrival curves and FIFO scheduling, to their Prosa counterparts. All definitions and proofs are formalized in Coq and available as additional material submitted together with this paper.

Our work is significant in many ways. First, it makes it easier for experts of one of the two approaches to understand the other by formally relating definitions. Second, given that a formal specification in Coq may be incorrect (meaning that it does not correspond to its informal definition), linking RTA and NC definitions is a way to increase confidence in these definitions. This has in fact led us to discover a bug in the definition of static priority in NC, as discussed at the end of the paper. Third, our formal connection provides the necessary foundations to compare existing analyses: proving that they are equivalent, that one is strictly more precise than another, or that they are incomparable. In addition, we hope that this connection can be used to build improved analyses based on a combined use of RTA and NC. Last, but not least, another strong contribution of this paper is its formalization of clocks. Discrete time is not a well suited setting for addressing issues related to clock drifts. The formal connection provided here between discrete and dense time represents a first step towards handling clock drifts in Coq.

This paper is organized as follows. Section 2 discusses related work. Section 3 provides an informal overview of the contribution. Sections 4 and 5 then present in a formal way the relevant state of the art regarding modeling in Coq of RTA and NC, respectively. Section 6 presents our first contribution, which formally links physical time (as in NC) and discrete clock-based time (as in Prosa). Sections 7, 8 and 9 then provide formal links between arrival sequences and cumulative curves, response times and horizontal deviations, request bound functions and arrival curves, respectively. Finally, Section 10 addresses properties at the scheduling level regarding FIFO and fixed-priority scheduling and Section 11 concludes.

All along the paper, definitions and lemmas are tagged with their (**name**) in the companion Coq development. No pen-and-paper proof of any result is provided in the paper: we focus instead on intuitions and explanations. Of course, we can only do that because the provided Coq proofs provide a much higher level of confidence.

2 Related Work

Building an analysis to guarantee that a system satisfies some real-time requirement is often a complex process, requiring long and error-prone proofs. One way to increase confidence in the correctness of such analyses is to use model checking, as e.g., in [5] to verify schedulers. Model checkers are automatic but less versatile than proof assistants such as Coq [10] or Isabelle/HOL [26]. In this paper, we follow a recent trend in the real-time community towards computer-assisted formal specifications and proofs using Coq¹.

Coq [10] is a proof assistant, i.e., a tool offering (1) a language to state theorems and describe their proofs, and (2) software – think of it as a compiler – for verifying these proofs. It can also be used to develop software whose execution is proved to conform to their (formal) specification such as the CompCert C compiler [19] or the CertiKOS operating

¹ See for example the Call to Action at ECRTS 2016, *Real Proofs for Real Time: Let's do better than "almost right"* [1]. Note that a similar momentum was given a decade ago in the programming language community. A number of mechanized formalizations (using either Coq or other tools) now appear each year at POPL, their main conference.

system [13]. When checking a proof, Coq will complain if a lemma is used without providing a proof for one of its hypotheses or if the proved hypotheses do not match the expected ones. Of course, Coq cannot guarantee that the formalization indeed corresponds to what was intended, so readability of the specification is key in Coq.

The main mathematical concepts of RTA have been formally defined and proved, with Coq, in the Prosa library [9]. This work has since seen many extensions. For example, [8] uses Coq as much for formalization purposes as for verification, while [7, 12] both use Coq as a powerful tool for providing abstract proofs which can then be instantiated into a variety of more concrete analyses. CertiCAN [11] represents a third type of application of Coq in relation to Prosa, in that it not only produces Coq proofs of various real-time analyses of the CAN protocol (from which efficient analysis tools can be extracted) but it can also be used to certify the results of non-certified industrial tools such as RTaW-Pegase. In yet another related line of work, [14, 15] have shown that it is possible to use Prosa for the schedulability analysis of a real-time OS kernel, namely CertiKOS.

On the NC side, first results on the formal verification of NC computation were presented in [20]. The aim was to verify that an existing tool was correctly using the NC theory. An Isabelle/HOL library was developed, specifying the main concepts of NC (flows and servers, arrival and service curves) and stating the main theorems but without proving them². The NC tool was then instrumented to provide not only a result, but also a proof on how NC had been used to produce that result. Isabelle/HOL was in charge of checking the correctness of this proof. Much more recently, actual proofs, this time using Coq, of the core mathematical concepts of NC have been provided in [22]. Actual computations of such non trivial manipulations of functions in the min-plus dioids can also be verified using Coq [23].

Regarding the specific contribution of this paper, namely the link between RTA and NC, note that comparing theories that coexist in the real-time community is not a new challenge [21], yet little research has been done on building bridges between them. Recent work has focused on connecting Compositional Performance Analysis (CPA, [24]) and NC into a common formal framework [6, 17], however not using proof assistants.

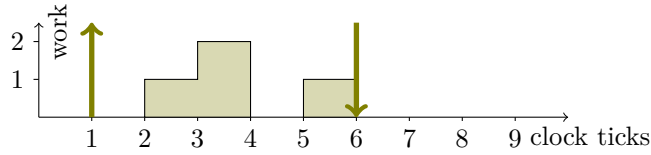
3 Overview of the Contribution

Before going into detail, let us provide here an informal description of the contributions of this paper, based on simple examples of RTA and NC execution traces.

Figure 1 shows a usual representation of a RTA trace, which contains here a single *job* characterized by its *arrival time* $arr(j)$ and its *cost*, both of which are integers. The scheduling of j determines the service it receives, and thus its *completion time* $end(j)$. Note that cost and instants are all integers but instants denote some points in time whereas a cost is a workload. The response time of j is defined as $end(j) - arr(j)$. A more detailed description of RTA will be presented in Section 4.

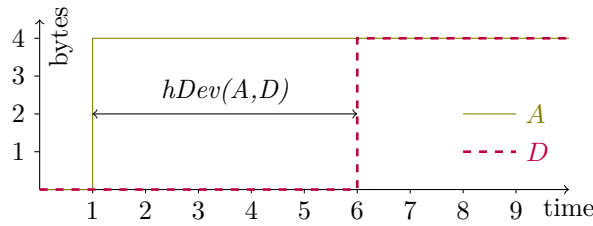
Figure 2 shows a comparable trace in NC. NC models workload using the notion of *cumulative curve*. A cumulative curve A is a non-decreasing function from \mathbb{R}_+ to \mathbb{R}_+ , whose semantics is that $A(t)$ represents a cumulative amount of work (in bits or CPU cycles). Like in RTA, the domain and image of A are the same sets, but the domain represents a (dense) time set whereas the image represents an amount of work. A curve can represent an amount of work demand, or an amount of work received. For instance, the crossing of a server by a packet is represented by two functions, the arrival cumulative curve A , whose value is null up to the packet arrival and is increased by the packet size at its arrival time, and a

² They were assumed to be correct, since they have been established in the literature for long.



■ **Figure 1** RTA: Scheduling of a job j with $arr(j) = 1$, $cost(j) = 4$ and the service received by job equal to 1 at instants 2 and 5, and equal to 2 at instant 3, leading to $end(j) = 6$.

departure cumulative curve D which is also null up to the packet departure and is increased by the packet size at its departure time. The delay is then defined as the horizontal deviation between A and D (the formal definition of which will be given in Section 5).



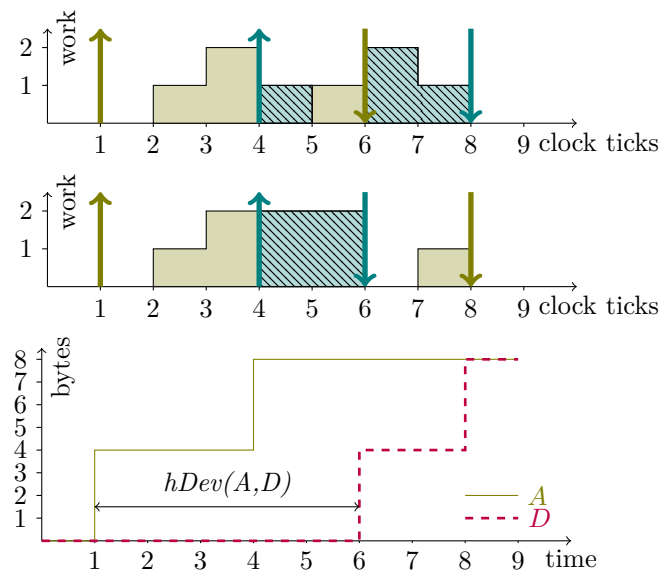
■ **Figure 2** NC: Arrival and departure cumulative curves representing a packet of size 4 entering a server at time 1 and leaving at time 6.

The relation between RTA and NC appears quite simple when comparing Figures 1 and 2, but the similarity between the graphical horizontal lines hides a major difference. Time in NC is the physical time, common to all calculators or switches (neglecting relativity effects) whereas time in RTA is the clock time, given by a hardware element, subject to imprecision and drifts.

Let us now discuss a slightly more complex example. In RTA, a *task* is a (possibly infinite) sequence of jobs, and the response time of the task in a schedule is the maximum of all individual response times. In NC, a *flow* is commonly represented by a single cumulative sequence. Note that the information is not exactly the same in both models. First, the NC model does not precisely represent the instants at which a task is scheduled, and only represents completion times. Since it does not represent the scheduling itself, it cannot support properties on scheduling. Second, the NC model of flow has no notion of individual packets or jobs. For example, when looking at Figure 3, representing the arrival and departure of two packets of size 4, one cannot guess if the packet leaving at time 6 is the first or the second one. In fact, even the number of packets is unknown: the jump of size 4 at time 1 could be created by a single packet of size 4, but also by 2 packets of size 2, and so on. Nevertheless, under the assumption that all jobs of the same task are scheduled with FIFO order, we can prove that the horizontal deviation is equivalent to the response time. Third, the RTA model does not capture clock drifts, making it more difficult to plug systems with different clocks, whereas NC uses the real universal time and can easily handle clock drift between systems.

We can now detail the core contributions reported in this paper, as follows:

- a clock model linking RTA and NC time, that can handle clock drifts;
- a mapping of each job j in a Prosa trace to a pair (A_j, D_j) in NC;
- a proof that the horizontal deviation $hDev(A_j, D_j)$ is equal to the response time of job j in case of perfect clocks, and a valid upper bound in presence of clock drifts;



■ **Figure 3** Graphical representation of two different schedulings of two jobs j, j' and the associated arrival and departure curves (A, D). Note that NC curves cannot distinguish the two schedulings.

- similar results for entire tasks and not just single jobs;
- a translation of a request bound function from RTA into an arrival curve from NC and vice versa;
- an equivalence between RTA and NC definitions of FIFO;
- an error found in a NC theorem during some preliminary works toward an equivalence on static priority.

All these items have been formalized in Coq based on the Prosa and NCCoq frameworks.

As already mentioned, this equivalence is only based on job release time, completion time and cost, ignoring the scheduling itself. Then, the property on FIFO relies on the observable part related to input/output, not on its internal implementation. And the result on static priority that will be presented in Section 10.2 is only a preliminary work. The modeling of the schedule will be discussed in the conclusion.

4 Response Time Analysis

Let us provide a more formal description of how the general concepts used by RTA are specified in Prosa. The Prosa library is structured around three main parts:

- **behavior/** provides a trace-based semantics of real-time systems and is meant to be as generic as possible. This part represents the common ground for all other parts of Prosa.
- **model/** contains a variety of modeling concepts which can be used to specify real-time systems, e.g., regarding task arrivals, preemptions, scheduling policies, platform abstractions etc. This part is meant to be used as a library, where one can pick definitions suitable for a specific system model.
- **analysis/** is where response time or schedulability analysis proofs are located.

In the rest of this section, we present the definitions from the Prosa library that are needed to relate RTA and NC. For readability, we omit trivial well-formedness conditions and a few basic definitions.

4.1 Behavior

The `behavior` part of Prosa is the core of the library, so most of its concepts are used in our work. First, let us recall that Prosa is based on a discrete model of time, so time is defined using natural numbers. This is formalized in Coq in the file `time.v` of Prosa as

```
Definition duration := nat.
Definition instant := nat.
```

To insist on this and avoid confusion with the dense time used by NC, we use in this paper the term *tick* for what is called instant in Prosa, and *number of ticks* for durations.

A *job* in Prosa is an abstract type with decidable equality: given two jobs, one can at least determine whether they are the same job or not. One can specify additional job parameters such as cost or arrival. Note that the cost of a job, which denotes the amount of service it requires to complete, is of type `work` (also represented using natural numbers), which would correspond in a real system to a number of processor cycles.

► **Definition 1.** (`job_cost`, `job_arrival`) *The cost of a job j , of type `work`, is denoted $cost(j)$. The arrival of a job j , of type `tick`, is denoted $arr(j)$.*

This is formalized in Coq in the file `job.v` of Prosa as

```
Definition JobType := eqType.
Definition work := nat.
Class JobCost (Job : JobType) := job_cost : Job -> work.
Class JobArrival (Job : JobType) := job_arrival : Job -> instant.
```

An arrival sequence is then defined as a function mapping any tick to the (finite) sequence of jobs that arrive at that tick.

► **Definition 2.** (`arrival_sequence`) *Given a type of jobs `Job`, an arrival sequence is a function $arrseq : \mathbb{N} \rightarrow 2^{Job}$.*

This is formalized in Coq in the file `arrival_sequence.v` of Prosa as

```
Definition arrival_sequence (Job : JobType) := instant -> seq Job.
```

A schedule essentially specifies which jobs are scheduled at every tick, and how much service they receive. In practice, depending on the specific execution platform, a lot more information may be relevant. This is why Prosa provides an abstract notion of processor state, which provides at least the above information about jobs scheduled and service. A schedule is then defined as a function that maps a tick to a processor state.

► **Definition 3.** (`schedule`) *A schedule is a function $sched : \mathbb{N} \rightarrow PState$. Given an instance of the abstract processor state class, the service received by a job j in a processor state $p \in PState$ is denoted $service_in(j, p)$.*

This is formalized in Coq in the file `schedule.v` of Prosa as

```
Definition schedule (PState : Type) := instant -> PState.
```

with

```
Class ProcessorState (Job : JobType) (PState : Type) :=
{ service_in : Job -> PState -> work; (* ... *) }.
```

Given a schedule and an instance of the abstract notion of processor state, one can derive the service received by a job at a given tick, between two given ticks, or up to a given tick.

► **Definition 4.** (*service*) Given a schedule $sched$, the service received by a job j before a tick $t \in \mathbb{N}$, denoted $service(j, t)$, is defined as $\sum_{0 \leq t' < t} service_in(j, sched(t'))$.

This is formalized in Coq in the file `service.v` as

```
Context {Job : JobType} {PState : Type} '{ProcessorState Job PState}.
Variable sched : schedule PState.

Definition service_at (j : Job) (t : instant) := service_in j (sched t).
Definition service_during (j : Job) (t1 t2 : instant) :=
  \sum_(t1 <= t < t2) service_at j t.
Definition service (j : Job) (t : instant) := service_during j 0 t.
```

Finally, a job completes its execution when it has received at least as much service as its cost³. The definition of a response time bound follows.

► **Definition 5.** (*job_response_time_bound*) A number of ticks r^+ is a response time bound for a job j if $service(j, arr(j) + r^+) \geq cost(j)$.

This is formalized in Coq as

```
Definition completed_by (j : Job) (t : instant) :=
  service j t >= job_cost j.
Definition job_response_time_bound (j : Job) (R : duration) :=
  completed_by j (job_arrival j + R).
```

We will in addition use a related definition from the `analysis` part, which introduces the notion of completion sequence.

► **Definition 6.** (*completion_sequence*) Given an arrival sequence $arrseq$ and a schedule $sched$, the corresponding completion sequence, denoted $endseq(arrseq, sched)$, is the function $\mathbb{N} \rightarrow 2^{Job}$ that maps each tick t to the jobs that complete at t .

This is formalized in Coq using the following definition from `service.v`

```
Definition completes_at (j : Job) (t : instant) :=
  ~~ completed_by j t.-1 && completed_by j t.
```

and then in the file `completion_sequence.v` as

```
Definition completion_sequence : arrival_sequence Job :=
  fun t => [seq j <- arrivals_up_to arr_seq t | completes_at sched j t].
```

We have now all the basic concepts required to describe the behavior of a real-time system for RTA, except the notion of readiness, which is not needed in this paper. In the following, we will sometimes use the term *trace* to refer to a pair $(arrseq, sched)$.

³ Note that we do not impose that a job receives exactly the amount of service corresponding to its cost. It could indeed receive more than needed to complete from the processor in the last tick of its execution.

4.2 Model

Unlike `behavior`, which is intended to be as universal as possible and to which all analyses using Prosa must relate, the `model` part is really meant as a library to be extended and picked from depending on the target system model and analysis. In this paper we will only use basic constructs from this part of Prosa to specify tasks and request bound functions.

Similar to jobs, a *task* in Prosa is nothing more than an abstract type with decidable equality. One usually specifies a task cost, and a function to relate jobs and tasks.

► **Definition 7.** (`job_task`, `task_cost`) *The task of a job j is denoted $\text{task}(j)$. The cost of a task tsk , of type `work`, is denoted $\text{cost}(\text{tsk})$.*

This is formalized in Coq in the file `concept.v` of Prosa as

```
Definition TaskType := eqType.
Class JobTask (Job : JobType) (Task : TaskType) := job_task : Job -> Task.
Class TaskCost (Task : TaskType) := task_cost : Task -> duration.
```

Defining the arrival of a task is less trivial than for a job and there exists a variety of models in Prosa for doing so, including periodic and sporadic arrival models. Prosa also defines arrival curves, which however constrain the number of arrivals rather than the amount of requested workload as in NC. We use here request bound functions, which are closer to the NC arrival curves.

► **Definition 8.** (`request_bound`) *A request bound is a monotonic function $\text{rbf} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{rbf}(0) = 0$. An arrival sequence arrseq satisfies an upper request bound rbf for a given task tsk if for any ticks $t_1, t_2 \in \mathbb{N}$ such that $t_1 \leq t_2$, the cumulative cost of all jobs of tsk that arrive in arrseq between t_1 and t_2 is bounded by $\text{rbf}(t_2 - t_1)$.*

This is formalized in Coq in the file `request_bound_functions.v` of Prosa as

```
Definition valid_request_bound_function (request_bound : duration -> work)
:= request_bound 0 = 0 ∧ monotone leq request_bound.

Definition respects_max_request_bound (tsk : Task) (max_request_bound :
duration -> work) := ∀ (t1 t2 : instant), t1 <= t2 ->
cost_of_task_arrivals arr_seq tsk t1 t2 <= max_request_bound (t2 - t1).
```

Let us now introduce the first definition that is not already part of Prosa. We specify the FIFO property, which guarantees that jobs complete in the order in which they arrive.

► **Definition 9.** (`FIFO_property`) *A trace $(\text{arrseq}, \text{sched})$ is said to respect the FIFO property when for any two jobs j_1, j_2 , if j_1 arrives before j_2 then it also completes before it, that is:*

$$\forall j_1, j_2, \forall t_1, t_2, t'_2, \\ (j_1 \in \text{arrseq}(t_1) \wedge j_2 \in \text{arrseq}(t_2) \wedge t_1 < t_2 \wedge j_2 \in \text{endseq}(\text{arrseq}, \text{sched})(t'_2)) \implies \\ \exists t'_1, t'_1 \leq t'_2 \wedge j_1 \in \text{endseq}(\text{arrseq}, \text{sched})(t'_1).$$

For flexibility, we use a version of the FIFO property that applies to jobs of a specific pair of tasks. If it holds for all pairs of a task set then we end up with the above property. The advantage of this version is that it can be used to express the general FIFO property as well as the specific FIFO order between jobs of the same task (which is related to task sequentiality in Prosa).

► **Definition 10.** (FIFO_per_task_property) *A trace $(arrseq, sched)$ is said to respect the FIFO property with respect to two tasks tsk_1 and tsk_2 when any job j_2 in tsk_2 completes after all jobs j_1 arrived before it in tsk_1 , that is:*

$$\begin{aligned} \forall j_1, j_2, \forall t_1, t_2, t'_2, (task(j_1) = tsk_1 \wedge task(j_2) = tsk_2 \wedge \\ j_1 \in arrseq(t_1) \wedge j_2 \in arrseq(t_2) \wedge t_1 < t_2 \wedge j_2 \in endseq(arrseq, sched)(t'_2)) \implies \\ \exists t'_1, t'_1 \leq t'_2 \wedge j_1 \in endseq(arrseq, sched)(t'_1). \end{aligned}$$

Note that there is a definition of FIFO in the model part of the Prosa library, which we do not use. First because it is quite complex, in the sense that it is derived from a more general notion of job level fixed priority. It is therefore more convenient to use a more straightforward definition for connecting RTA to NC. Formally relating our direct definition of FIFO to the Prosa definition is left for future work. Second, and more importantly, the Prosa FIFO definition is in fact different from ours, as it constrains the scheduling of jobs in a first-come-first-serve manner while the above definition constrains the order in which jobs complete. In particular, the Prosa FIFO scheduling does not necessarily guarantee the FIFO property in multicore systems. The notion of server in NC is very different from the notion of scheduler in RTA, and as a result a formal link between the two can only succeed at the interface, expressed in terms of arrivals and completions.

5 Network Calculus

Similarly to how the Prosa library is formalizing RTA analyses in Coq, the NC theory is formalized in the NCCoq library, which is available at <https://gitlab.mpi-sws.org/proux/nc-coq>. A short overview can be found in [22]. This section introduces the notions from NCCoq that we have used in our work.

5.1 Behavior

NC models dense time using the set of real numbers \mathbb{R} , and more specifically its subset of nonnegative values $\mathbb{R}_+ := \{x \in \mathbb{R} \mid x \geq 0\}$. Positive real numbers $\mathbb{R}_+^* := \{x \in \mathbb{R} \mid x > 0\}$ as well as extended reals $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$ also play an important role. In the NCCoq formalization, these fundamental definitions are taken from the MathComp Analysis library [2] and respectively noted $\mathbb{R} : \text{realType}$, $\{\text{nonneg } \mathbb{R}\}$, $\{\text{posnum } \mathbb{R}\}$ and $\backslash\text{bar } \mathbb{R}$. NC relies on a few classes of functions, defined as follows.

► **Definition 11.** (F) $\mathcal{F} := \mathbb{R}_+ \rightarrow \overline{\mathbb{R}}$ is the set of functions from \mathbb{R}_+ to $\overline{\mathbb{R}}$.

This is formalized in Coq in the file `RminStruct.v` of NCCoq as

```
Definition F := {nonneg R} -> \bar R.
```

► **Definition 12.** (Fplus) $\mathcal{F}_+ := \{f \in \mathcal{F} \mid 0 \leq f\}$ is the subset of functions from \mathcal{F} that are nonnegative.

This is formalized in Coq as

```
Definition F_0 : F := fun => 0%E.
Definition nonNegativeF := [qualify a f | F_0 <= f ]%0.
Record Fplus := { Fplus_val :> F; _ : Fplus_val \is a nonNegativeF }.
```

5:10 A Formal Link Between Response Time Analysis and Network Calculus

where the first line defines $f_0 : \mathcal{F}$ the constant⁴ function equal to $0 \in \overline{\mathbb{R}}$; the second line defines the nonnegative functions as functions that are larger than f_0 . The third line uses a common Coq construction, where the subset of a set X satisfying a property P is defined as records with an element $x \in X$ of this set (here, `Fplus_val` of type⁵ `F`) and an unnamed `_` proof of $P(x)$ (here, a proof that `Fplus_val` is a nonnegative function).

► **Definition 13.** (`Fup`) $\mathcal{F}^\uparrow := \{f \in \mathcal{F}_+ \mid \forall xy, x \leq y \Rightarrow f(x) \leq f(y)\}$ is the subset of nondecreasing functions from \mathcal{F}_+ .

The Coq formalization `Fup` proceeds similarly to `Fplus`.

Now equipped with these basic definitions, we can define the main object of NC: data flow cumulative curves. They play a similar role to arrival sequences in RTA.

► **Definition 14.** (`flow_cc`) A cumulative curve is a function $f \in \mathcal{F}^\uparrow$ satisfying

- $f(0) = 0$
- f is left continuous
- f only takes finite values: $\forall t, f(t) \in \mathbb{R}_+$

We denote \mathcal{C} the set of cumulative curves.

This is formalized in Coq in the file `cumulative_curves.v` of `NCCoq`.

NC defines delays using the notion of horizontal deviation between two cumulative curves.

► **Definition 15.** (`hDev_at`, `hDev`) For $f, g \in \mathcal{F}$ and $t \in \mathbb{R}_+$, the horizontal deviation $hDev(f, g, t) \in \overline{\mathbb{R}}_+$ between f and g at t is defined as

$$hDev(f, g, t) := \inf \{d \in \mathbb{R}_+ \mid f(t) \leq g(t + d)\}$$

and the horizontal deviation $hDev(f, g) \in \overline{\mathbb{R}}_+$ between f and g is defined as

$$hDev(f, g) := \sup \{hDev(f, g, t) \mid t \in \mathbb{R}_+\}.$$

This is formalized in Coq in the file `deviations.v` of `NCCoq`.

Finally, *servers* constitute the last main notion of NC to model concrete behaviors. `NCCoq` includes a few different flavors of servers. This notion would relate to the notion of scheduler in the RTA world. There is however no such thing formalized in the core of *Prosa*, which is based on schedules in the `behavior` and on scheduling policies (that are predicates on traces) in the `model` part. We thus simply omit servers from the current paper, in which we will directly deal with input and output cumulative curves.

5.2 Model

Just like request bound functions are a tool to express contracts on arrivals in RTA, arrival curves are the NC tool to specify inputs of servers.

⁴ $0 \in \overline{\mathbb{R}}$ is denoted `0%E` in Coq, `%E` being the scope annotation for extended real notations in the `MathComp Analysis` library [2].

⁵ The `>` syntax makes `Fplus_val` a Coq coercion, meaning `Fplus_val` will be inserted automatically by Coq to cast a `Fplus` as a `F` wherever needed. In practice, this means that a value of type `Fplus` can be used just like a function of type `F`.

► **Definition 16.** (`is_maximal_arrival`) A function $\alpha \in \mathcal{F}$ is an arrival curve for any cumulative curve A when

$$\forall t, d \in \mathbb{R}_+, A(t + d) - A(t) \leq \alpha(d)$$

This is formalized in Coq in the file `arrival_curves.v` of NCCoq.

This concludes our overview of the already existing definitions that are needed to present our contribution in the remainder of the paper.

6 Dense versus Discrete Time

Hardware clocks are devices whose aim is to provide a measure of time, generally based on a physical oscillator, characterized by its frequency f . Based on [16], we may distinguish ideal clocks, when the difference between two signal occurrences is exactly $1/f$, from constant drifted clocks, when the difference between two signal occurrences is exactly ρ/f , where $\rho - 1$ represents the drift (commonly related to the temperature), or more general constraints [25]. In computers, the time value is computed as a function based on a counter incremented at each signal occurrence.

Thus, Section 6.1 will introduce a universal notion of clock, to make a link between time and its discrete measurement, and Section 6.2 will introduce elements on the clock quality. The Coq definitions and lemmas referenced in this section can be found in the file `clock.v`.

6.1 Clocks

Since RTA relies on a discrete notion of time (ticks are in \mathbb{N}) whereas NC uses real times in \mathbb{R}_+ , we need a link between discrete and dense times in order to relate both theories. A simple solution would be to use the canonical injection of \mathbb{N} in \mathbb{R}_+ , that is, to consider each tick $n \in \mathbb{N}$ as happening at real time $n \in \mathbb{R}_+$. However, doing this would preclude any modeling of behaviors such as clock drifts. We thus need a more generic modeling. To that end, we first introduce the notion of *universal clock*.

► **Definition 17.** (`uclock`) A universal clock is a function $c : \mathbb{N} \rightarrow \mathbb{R}_+$ satisfying

- $c(0) = 0$
- there exists a $\text{min_intertick}_c \in \mathbb{R}_+^*$ such that for all $n \in \mathbb{N}$, we have

$$c(n + 1) \geq c(n) + \text{min_intertick}_c.$$

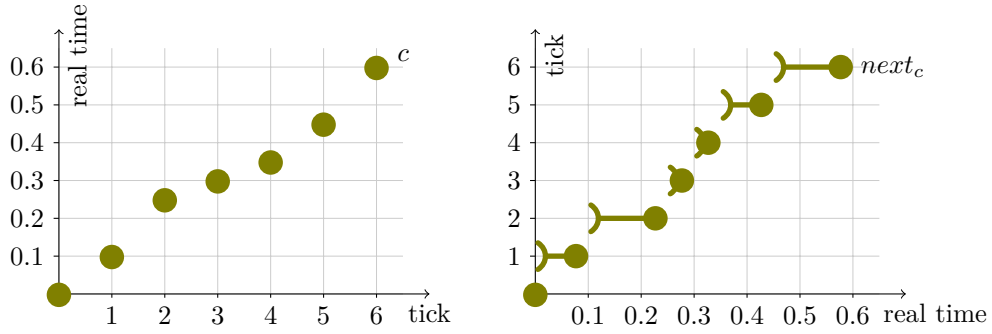
Thus given a clock c , the real value $c(n)$ is the physical time of the n -th clock tick. The condition $c(0) = 0$ means that the clock starts right away while the min_intertick_c is mostly there to exclude functions such as $n \mapsto 1 - \frac{1}{n}$ which could cause all kinds of Zeno phenomena.

While we now have a link from discrete to dense time, it would be useful to get some kind of inverse from dense to discrete time. One can notice that, thanks to the condition on min_intertick_c above, for any clock c and real time r , there is a tick n that happens exactly at time r or is the next one after r .

► **Lemma 18.** (`next_tick_ex`) For any clock c and $r \in \mathbb{R}_+$, there exist $n \in \mathbb{N}$ such that $r \leq c(n)$ and for all $n' < n$, we have $c(n') < r$.

This gives us a function $\text{next}_c : \mathbb{R}_+ \rightarrow \mathbb{N}$.

► **Definition 19.** (`next_tick`) For any clock c , we denote $\text{next}_c : \mathbb{R}_+ \rightarrow \mathbb{N}$ the function mapping each r to the n given by Lemma 18.



■ **Figure 4** Graphical representation of an arbitrary clock c and its inverse function $next_c$.

This is illustrated in Figure 4 and formalized as follows in Coq: we first prove Lemma 18

```

Lemma next_tick_ex (c : uclock) (r : {posnum R}) :
  { n | r%:nngnum <= (c n)%:nngnum
    ∧ ∀ n', (n' < n)%N → (c n')%:nngnum < r%:nngnum }.

```

where $\{n|P(n)\}$ is a Coq notation for “there exists a n such that $P(n)$ ”. Since such a proof is a simply dependent pair $(n, proof\ of\ P(n))$, one can use the first projection `proj1_sig` to extract n out of it.

```

Definition next_tick (c : uclock) (r : {posnum R}) : instant :=
  proj1_sig (next_tick_ex c r).

```

► **Remark 20.** We could as well have chosen n such that $r < c(n)$ rather than $r \leq c(n)$. The current choice appeared more convenient to match the left continuity conditions of NC.

Once clocks are defined, one can use Coq to formally verify a few lemmas about them. In practice, we have proved about a dozen lemmas that came useful in the remaining of our formal development, among which

► **Lemma 21.** (*next_tick_0*) For any clock c , then $next_c(0) = 0$.

► **Lemma 22.** (*uclockK*) For any clock c , any tick $n \in \mathbb{N}$, then $next_c(c(n)) = n$.

6.2 Pseudo Periodic Clocks

Although the condition on $min_intertick_c$ in the definition of clocks was primarily introduced to rule out Zeno phenomena, it can also be used to model clocks whose time between two subsequent ticks is lower bounded. In addition, we will need to model clocks whose intertick time is also upper bounded. Such clocks can be seen as periodic clocks with an inexact period that can vary between some minimal and maximal value. They are thus called *pseudo periodic clocks*.

► **Definition 23.** (*ppuclock*) A *pseudo periodic clock* is a clock c such that there exist $max_intertick$ satisfying $min_intertick_c \leq max_intertick$ and for all n , we have $c(n+1) \leq c(n) + max_intertick$.

► **Example 24.** (*periodic_uclock*) Periodic clocks are a special case of pseudo periodic clocks for which $min_intertick_c = max_intertick$ and the n -th tick happens exactly at time $n \times period$.

```

Program Definition periodic_uclock (period : {posnum R}) : ppucclock :=
  Build_ppuclock
    (Build_uclock (fun n => (n%:R * period%:num)%:nng) _ period _)
    period _ .

```

Thus, a clock c such that: $\forall n, c(n+1) - c(n) = 10^{-9}s$ is a periodic clock representing an ideal 1MHz clock, whereas a clock such that: $\forall n, c(n+1) - c(n) = (10^{-9} + 10^{-12})s$ is also a periodic clock, but representing a 1MHz clock with a constant drift of 0.1%. Finally, any clock such that: $\forall n, c(n+1) - c(n) \in [10^{-9} - 10^{-12}, 10^{-9} + 10^{-12}]$ is a pseudo periodic clock with a non constant drift not greater than 0.1%.

As a tool to link physical time and clock ticks, these definitions of clocks will be pervasive to link RTA and NC in the rest of this paper. Using different clocks for different parts of a system then enables to model drift between different physical clocks.

7 Linking Arrival Sequences and Cumulative Curves

Equipped with this notion of clock, we can now relate arrival sequences from RTA to cumulative curves from NC. We first do so for a single job then for an entire task.

The Coq definitions and lemmas referenced in this section can be found in the file `flow_cc_of_arrival_sequence.v`.

7.1 For a Single Job

► **Definition 25.** (`flow_cc_of_job`) *Given a job j and a clock c , one can build a cumulative curve $A_j \in \mathcal{C}$, as seen in Section 5, defined by*

$$A_j : \mathbb{R}_+ \rightarrow \mathbb{R}_+ \tag{1}$$

$$t \mapsto \begin{cases} 0 & \text{if } t \leq c(\text{arr}(j)) \\ \text{cost}(j) & \text{otherwise.} \end{cases} \tag{2}$$

We can thus establish a relation between jobs and cumulative curves.

► **Definition 26.** (`related_job_flow_cc`) *A job j , with its cost and arrival time, and a cumulative curve A are related when A is exactly the function A_j of Definition 25.*

7.2 For an Entire Task

We can proceed similarly for a task with multiple jobs.

► **Definition 27.** (`flow_cc_of_arrival_sequence`) *Given an arrival sequence arrseq , a clock c and a task tsk , the cumulative curve $A_{\text{tsk}} \in \mathcal{C}$ is defined by*

$$A_{\text{tsk}} : \mathbb{R}_+ \rightarrow \mathbb{R}_+ \tag{3}$$

$$t \mapsto \sum_{\substack{j \in \bigcup \{\text{arrseq}(i) \mid i < \text{next}_c(t)\}, \\ \text{task}(j) = \text{tsk}}} \text{cost}(j). \tag{4}$$

In Coq, we first define the arrival for a given task between two instants t_1 and t_2 .

```
Let arrivals_of_tsk (arr_seq : arrival_sequence Job) tsk t1 t2 :=
  [seq j <- arrivals_between arr_seq t1 t2 | job_task j == tsk].
```

Then, given a clock c , we define the cumulative curve

```
Program Definition flow_cc_of_arrival_sequence
  (s : arrival_sequence Job) (c : uclock) (tsk : Task) :=
  Build_flow_cc (Build_Fup (Build_Fplus
    (fun t => (\sum_(j <- arrivals_of_tsk s tsk 0 (next_tick c t))
      job_cost j)%:R%:E)
    _ ) _ ) _.
```

It is worth noting here that, although it may not be immediately apparent in the pen-and-paper Definition 27, this definition involves some proofs⁶ to prove that A_{tsk} is actually a cumulative curve, i.e., is in \mathcal{C} .

We can thus establish a relation between arrival sequences and cumulative curves.

► **Definition 28.** (`related_arrival_flow_cc`) *Given a clock c and a task tsk , an arrival sequence $arrseq$ and a cumulative curve A are related when A is exactly the function A_{tsk} of Definition 27.*

Finally, we can prove that the cumulative curve for a task is nothing else than the sum of the cumulative curves for each individual job in the task.

► **Lemma 29.** (`flow_cc_of_arrival_sequence_of_job`) *Given a clock c , a task tsk and an arrival sequence $arrseq$, we have for all $t \in \mathbb{R}_+$:*

$$A_{tsk}(t) = \sum_{\substack{j \in \bigcup \{arrseq(i) \mid i < next_c(t)\}, \\ task(j)=tsk}} A_j(t).$$

Note that we may want to write $A_{tsk}(t) = \sum_{j \in \bigcup \{arrseq(i)\}, task(j)=tsk} A_j(t)$ since $A_j(t) = 0$ for jobs arriving after t , but it would lead to infinite sums whose convergence has to be proved to Coq. The given statement is equivalent and more convenient.

We now have a link between the main concrete behavior notions in RTA and NC, namely arrival sequences and cumulative curves. This link constitutes the basis enabling to relate other notions in the rest of the paper: notions of response time and delay, contracts with request bound functions and arrivals curves or various scheduling policies.

8 Linking Response Time and Horizontal Deviation

Equipped with a formal link between RTA's arrival sequences and NC's cumulative curves, we can now relate response times and horizontal deviations. Again, we first do so for a single job then for an entire task with multiple jobs.

The Coq definitions and lemmas referenced in this section can be found in the file `delay.v`.

⁶ By filling the holes `_` of **Program Definition**, the proofs themselves are present in the source file but omitted here for the sake of clarity.

8.1 For a Single Job

► **Definition 30.** (*arrival_sequence_of_job*) Given a set of jobs Job and a job $j \in Job$, we define the arrival sequence $arrseq_j$ of this job as

$$arrseq_j : \mathbb{N} \rightarrow 2^{Job} \tag{5}$$

$$t \mapsto \begin{cases} \{j\} & \text{if } t = arr(j) \\ \emptyset & \text{otherwise.} \end{cases} \tag{6}$$

We then prove that this arrival sequence is consistent with the considered `job_arrival` function, meaning that $j \in arrseq_j(arr(j))$.

► **Lemma 31.** (*arrival_sequence_of_job_consistent*) The arrival sequence $arrseq_j$ of a job j , as defined in Definition 30, is consistent with the arrival time of j .

Thus, from a response time bound on an arrival sequence, one can deduce a horizontal deviation on related arrival curves.

► **Lemma 32.** (*hDev_of_job_response_time*) Given a pseudo periodic clock c , for a given job j , a related cumulative curve A and a cumulative curve D related to the completion sequence of j , if some $r^+ \in \mathbb{N}$ is a response time bound for j , then

$$hDev(A, D) \leq r^+ \times max_intertick_c.$$

Note that here, a pseudo periodic clock, and not just a clock, is required because the result involves the upper bound $max_intertick$ between two consecutive clock ticks. A similar result holds in the reverse direction.

► **Lemma 33.** (*job_response_time_of_hDev*) Given a clock c , for a given job j , a related cumulative curve A and a cumulative curve D related to the completion sequence of j , given a bound on the horizontal deviation between A and D , if the horizontal deviation between A and D is bounded by $r^+ \times min_intertick_c$ for some $r^+ \in \mathbb{N}$

$$hDev(A, D) \leq r^+ \times min_intertick_c$$

then r^+ is a response time bound for j .

Note that this involves the bound $min_intertick_c$ between two consecutive clock ticks.

8.2 For an Entire Task

We can proceed similarly for entire tasks. Thus, from a task response time bound on an arrival sequence, one can deduce an horizontal deviation on related arrival curves.

► **Lemma 34.** (*hDev_of_task_response_time*) Given a pseudo periodic clock c , a task tsk , an arrival sequence $arrseq$, a related arrival curve A , a schedule $sched$ and a cumulative curve D related to its completion sequence $endseq(arrseq, sched)$, if some $r^+ \in \mathbb{N}$ is a response time bound for all jobs of task tsk , then

$$hDev(A, D) \leq r^+ \times max_intertick_c.$$

There is a big caveat for the reverse direction: since NC is unable to distinguish individual jobs, as cumulative curves only register the sum of their costs⁷, one needs to add an additional FIFO hypothesis that the jobs are ordered the same way in the arrival and completion sequences. We end up with the following result to derive a RTA response time bound from a NC horizontal deviation.

► **Lemma 35.** (`task_response_time_of_hDev`) *Given a clock c , a task tsk , an arrival sequence $arrseq$, a related cumulative curve A , a schedule $sched$, and a cumulative curve D related to its completion sequence $endseq(arrseq, sched)$, if the FIFO property of Definition 10 is satisfied with $tsk_1 := tsk$ and $tsk_2 := tsk$ and if the horizontal deviation between A and D is bounded by $r^+ \times \min_intertick_c$ for some $r^+ \in \mathbb{N}$:*

$$hDev(A, D) \leq r^+ \times \min_intertick_c$$

then r^+ is a response time bound for all jobs of task tsk .

We now have a way to interpret RTA analyses results in a NC setting and vice versa.

9 Linking Request Bound Functions and Arrival Curves

The links established between RTA and NC in the previous sections were only pertaining to the *behavior* subsections of Sections 4 and 5, defining respectively RTA and NC. That is, those notions are purely mathematical, no actual computation is made on them. On the contrary, the notions of request bound functions (RBF) and arrival curves, defined in the *model* subsections of Sections 4 and 5, are actually manipulated by analyses. They are “computational” objects and not mere mathematical definitions. Thus, given a RBF, a “constructive” definition of a related arrival curve could enable to actually communicate results from a RTA analysis to a NC one, and vice versa. The current section precisely aims at building such “constructive” links.

The Coq definitions and lemmas referenced in this section can be found in the files `arrival_curve_of_request_bound_function.v` and `request_bound_function_of_arrival_curve.v`.

9.1 From Request Bound Functions to Arrival Curves

Given a RBF and a clock, one can define an arrival curve in \mathcal{F}_+ .

► **Definition 36.** (`arrival_curve_of_request_bound_function`) *Given $rbf : \mathbb{N} \rightarrow \mathbb{N}$ and a clock c , we can define $\alpha_{rbf} : \mathcal{F}_+$ as*

$$\alpha_{rbf} : \mathbb{R}_+ \rightarrow \overline{\mathbb{R}}_+ \tag{7}$$

$$d \mapsto rbf \left(\left\lceil \frac{d}{\min_intertick_c} \right\rceil \right). \tag{8}$$

Moreover, if the RBF is valid, then the arrival curve can be proved to be in \mathcal{F}^\uparrow , i.e., it is nondecreasing.

► **Lemma 37.** (`arrival_curve_of_valid_request_bound_function`) *For any rbf and clock c , if rbf is a valid RBF (according to Definition 8), then $\alpha_{rbf} : \mathcal{F}^\uparrow$.*

⁷ See the example in Section 3 with Figure 3.

One can then prove that our definition indeed translates RBFs into arrival curves.

► **Lemma 38.** (`arrival_curve_of_request_bound_function_is_maximal_arrival`) *Given a clock c , a task tsk , an arrival sequence $arrseq$, a cumulative curve A and a RBF rbf , then if $arrseq$ and A are related for task tsk , if rbf is a valid RBF for tsk in $arrseq$, then α_{rbf} is an arrival curve for A .*

It is worth noting that this correspondence is tight for periodic clocks (i.e., one can prove⁸ the converse of Lemma 38) but can be conservative otherwise.

9.2 From Arrival Curves to Request Bound Functions

Conversely, from an arrival curve in \mathcal{F} , one can define a RBF.

► **Definition 39.** (`request_bound_function_of_arrival_curve`) *Given $\alpha : \mathcal{F}$ and a pseudo periodic clock c , we can define $rbf_\alpha : \mathbb{N} \rightarrow \mathbb{N}$ as*

$$rbf_\alpha : \mathbb{N} \rightarrow \mathbb{N} \tag{9}$$

$$d \mapsto \begin{cases} 0 & \text{if } d \leq 0 \\ \lceil \alpha(d \times max_intertick_c) \rceil & \text{otherwise.} \end{cases} \tag{10}$$

Note that this requires a pseudo periodic clock as it involves `max_intertick`. Also note that $rbf_\alpha(0)$ is explicitly set to 0 because RTA defines valid RBFs as starting at 0 whereas NC has no such requirement⁹ on α . This definition is encoded as follows in Coq.

```

Definition request_bound_function_of_arrival_curve
  (alpha : F) (c : ppuclock) : duration -> nat :=
  fun d =>
    if (d <= 0)%N then 0%N else
      '|ceil (fine (alpha (d%:R * (ppuclock_max_intertick c)%:num)%:nng%R)
        )|%N.

```

There are two things worth noticing about that Coq statement that were not immediately apparent in Definition 39. First, there is an additional absolute value `'| . |` around the ceiling function `ceil`. This is needed for the definition to typecheck because the ceiling function returns a (signed) integer whereas we need a natural number. In practice, since its argument is always nonnegative, it is a no-op. Second, we need to insert `fine` because $\alpha : \mathcal{F}$ returns values in $\overline{\mathbb{R}}$ whereas `ceil` expects an input in \mathbb{R} . `fine` acts as the identity function on \mathbb{R} and maps infinities to 0. This will require an extra finiteness hypothesis in the next two lemmas.

When the arrival curve is in \mathcal{F}^\uparrow , the RBF can be proved valid.

► **Lemma 40.** (`valid_request_bound_function_of_arrival_curve`) *Given $\alpha : \mathcal{F}^\uparrow$ and a pseudo periodic clock c , if α only takes finite values (i.e., for all d , $\alpha(d) \in \mathbb{R}$), then rbf_α is a valid RBF.*

One can then prove that our definition indeed translates arrival curves to RBFs.

⁸ C.f., lemma `arrival_curve_of_request_bound_function_respects_max_request_bound`.

⁹ Even if, in practice, arrival curves with $\alpha(0) \neq 0$ are of no interest hence never used, the set \mathcal{F} enjoys a nice algebraic structure of dioid which $\{\alpha : \mathcal{F} \mid \alpha(0) = 0\}$ doesn't and NC makes extensive use of this algebraic structure.

► **Lemma 41.** (*request_bound_function_of_arrival_curve_respects_max_request_bound*) Given a pseudo periodic clock c , a task tsk , an arrival sequence $arrseq$, a cumulative curve A and $\alpha : \mathcal{F}^\uparrow$ such that for all d , $\alpha(d) \in \mathbb{R}$, then if $arrseq$ and A are related for task tsk , if α is an arrival curve for A , then rbf_α is a RBF for task tsk in s .

Again this is tight only for periodic clocks.

The definitions of α_{rbf} and rbf_α do not appear directly computable as they act on an infinite domain but practical implementations of RTA or NC analyses usually handle some kind of periodic functions¹⁰, in which case α_{rbf} and rbf_α can be actually computed. We believe this is a powerful result, offering an effective way to translate RTA results into NC hypotheses and vice versa. This enables us to combine the two theories and take advantage of their respective strengths to derive real-time analysis results that none of the techniques alone could provide.

10 Linking Scheduling Properties

We have already mentioned in Section 3 that the contribution does not address the scheduling itself. Nevertheless, the First In First Out (FIFO) property as given in Definition 10 is not a scheduling policy, but a property on schedulings. We are thus able to show equivalence of the FIFO property in both formalisms, as presented in Section 10.1. Moreover, while doing preliminary works toward an equivalence of static priority schedulings, an issue was uncovered, as presented in Section 10.2.

10.1 FIFO

While the definition of FIFO in the RTA setting, as provided in Definition 10, appears relatively straightforward, the NC definition may be much more enigmatic to anyone but NC experts. We prove that this NC definition matches the RTA one, thus giving it a higher confidence. The NC definition of the FIFO service policy is as follows.

► **Definition 42.** For $n \in \mathbb{N}$, the cumulative curves A_i and D_i for $i < n$ are said to respect the FIFO service policy when

$$\forall i, j \in \{0, \dots, n-1\}, \forall t, u \in \mathbb{R}_+, A_i(u) < D_i(t) \implies A_j(u) \leq D_j(t).$$

The Coq definitions and lemmas referenced in this section can be found in the file `fifo.v`. We prove in particular the equivalence between the RTA and NC definitions of FIFO.

► **Lemma 43.** (*FIFO_arrival_sequences_to_flow_cc*) Given a clock c , an arrival sequence $arrseq$, a schedule $sched$, two cumulative curves A and D respectively related to $arrseq$ and the completion sequence $endseq(arrseq, sched)$, if $arrseq$ and $sched$ respect the (RTA) FIFO property (for any pair of tasks, according to Definition 10), then A and D respect the (NC) FIFO policy (according to Definition 42).

Note that for the converse to hold, we require the extra hypothesis that each task satisfies the FIFO policy with itself, because NC is blind on the order of jobs within a single task, as illustrated on Figure 3 in Section 3.

¹⁰For the very precise reason that it is possible to perform actual computations on this subclass of functions and that actual real-time behaviors are commonly periodic.

► **Lemma 44.** (`FIFO_flow_cc_to_arrival_sequences`) *Given a clock c , an arrival sequence $arrseq$, a schedule $sched$, two cumulative curves A and D respectively related to $arrseq$ and the completion sequence $endseq(arrseq, sched)$, if A and D respect the (NC) FIFO policy (according to Definition 42) and if for any task tsk , the arrival sequence $arrseq$ and schedule $sched$ satisfy the FIFO policy between task tsk and itself, then $arrseq$ and $sched$ respect the (RTA) FIFO policy (for any pair of tasks, according to Definition 10).*

Thus, we have seen that the definitions of FIFO in RTA and NC do match. This is a worthwhile result as it strengthens our confidence in both definitions. In particular, being more abstract, the NC definition easily looks rather mysterious for a non NC expert.

10.2 Fixed/Static Priority

Although the NC definition of static priority looks more natural than its definition of FIFO, we were eager to prove the same kind of equivalence for it. Here is the definition of fully preemptive static priority as given in [3, Def. 7.8]

► **Definition 45.** *Given $n \in \mathbb{N}$, the cumulative curves A_i and D_i for $i < n$ satisfy the static priority service policy when, for all $i < n$:*

$$\forall s, t \in \mathbb{R}_+, \left(\forall u \in [s, t], \sum_{j < i} A_j(u) > \sum_{j < i} D_j(u) \right) \implies D_i(t) = D_i(s)$$

where $<$ is a total order on $\{0, \dots, n-1\}$.

In this definition, a flow j has a higher priority than i when $j < i$.

Equipped with this definition, the following lemma is proved.

► **Lemma 46.** (`FP_arrival_sequence_to_flow_cc`) *Given a clock c , an arrival sequence $arrseq$ satisfying sequential readiness and a schedule $sched$ on a fully preemptive ideal uniprocessor, a priority $<$, two cumulative curves A and D respectively related to $arrseq$ and the completion sequence $endseq(arrseq, sched)$, if $arrseq$ and $sched$ respect the (RTA) fixed priority policy relative to $<$, then A and D respect the (NC) static priority policy (according to Definition 45).*

The reverse doesn't hold since, as already explained, the departure curve D doesn't represent the scheduling $sched$ but only the related completion sequence.

It is worth noting that the main theorem using Definition 45, given in [3, Thm. 7.6] is wrong¹¹ and was proved in NCCoq by strengthening the above definition, replacing the left-closed interval $[s, t]$ by the left-open one $(s, t]$. This small change was deemed innocuous at the time and did not raise further attention¹². While attempting to prove the equivalence of the strengthened hypothesis with the RTA definition of the fully preemptive fixed priority policy, it became obvious that the hypothesis strengthening was not that innocuous as it broke the equivalence with RTA. Following this discovery, the theorem in NCCoq (`SP_residual_service_curve` in file `static_priority.v` of NCCoq) was rather fixed by strengthening another hypothesis, namely demanding the service curve of the aggregate server to be a cumulative curve¹³ in \mathcal{C} .

¹¹ Counter example: consider two flows 1 and 2 with $1 < 2$, respective arrivals $A_1 := s_2$ and $A_2 := s_1$ and departures $D_1 := s_4$ and $D_2 := s_2$ (with s_d defined as $t \mapsto 0$ when $t \leq d$ and $t \mapsto 1$ otherwise) and an aggregated strict service β defined as $d \mapsto 0$ when $d < 2$ and $d \mapsto 1$ otherwise.

¹² Maybe because it was the most obvious hypothesis strengthening to make the proof given in [3] work.

¹³ This hypothesis strengthening is reasonably innocuous as most service curves already live in \mathcal{C} and could otherwise be easily under-approximated by a service curve in \mathcal{C} .

This experience report is particularly interesting as it shows how formal proofs of equivalence between two theories can unveil errors that were overlooked for a few years.

11 Conclusion

In this paper, we have built bridges between Response Time Analysis (RTA) and Network Calculus (NC) and formalized them with the Coq proof assistant. This shows how the notions of job, task, trace, response time, in RTA are related to the notions of arrival curve, departure curve and delay in NC. To do so, we have formalized a notion of (possibly drifty) clock and proved that what is called FIFO in both formalisms represents the same constraints. We also prove that bounds computed in one framework are valid in the other one, even when considering clock drifts.

The related work presented in Section 2 was already providing increased confidence into RTA and NC by achieving formal proofs in Coq of already known results, sometimes discovering bugs in proofs or in the results themselves. The new formal bridges between these two theories that coexist in the real-time community bring more than just confidence.

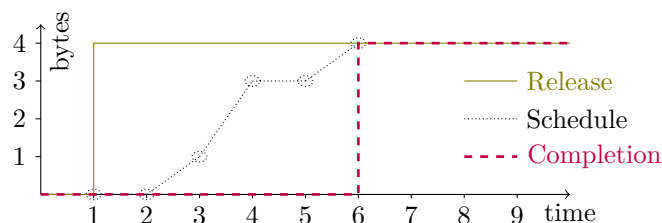
First, the most obvious contribution is a better mutual understanding between both communities. Such a comparison between different points of view may help each community in its reflection on its own models (e.g., NC has a notion of per-flow server, capturing several behaviors and not bound to a scheduling policy, whereas RTA has a notion of global trace, and sets of traces, but no scheduler).

Second, being able to go back and forth between theories allows us to analyze a complete system by using each theory where it is the most convenient and to combine the results to get the best of both theories in each component, like in [18] or more recently, in [17].

Finally, a third result, unexpected when we started this work, concerns the strength of modeling. Formal work makes it possible to check that some theory is correct, in the sense that the model fulfills some properties. But there is no way to check that the model reflects the reality. Building formal bridges between models is thus an effective way to increase our confidence in these models. For example, as reported in Section 10.2, this unveiled a weakness in the definition of static priority in the NC formal model.

These results open opportunities for future research. They provide a strong background for schedulability analyses in presence of clock drifts. They could also be pursued *per-se* to provide more links between RTA and NC, and also provide links with CPA.

Future work will consist in modeling the schedule itself. It may be possible for example to represent in NC the cumulative curves of release, schedule and completion of the job of Figure 1 as illustrated in Figure 5, inspired from [4, §2.3].



■ **Figure 5** NC: Cumulative curves representing the job of Figure 1 and its schedule.

References


- 1 28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016. IEEE Computer Society, 2016. URL: <https://ieeexplore.ieee.org/xpl/conhome/7557819/proceeding>.
- 2 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: a case study in functional analysis. In *IJCAR 2020 - International Joint Conference on Automated Reasoning*, pages 1–19, Paris, France, June 2020. URL: <https://hal.inria.fr/hal-02463336>.
- 3 Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons, Ltd, October 2018. doi: 10.1002/9781119440284.
- 4 Anne Bouillard and Éric Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 18(1):3–49, October 2008. <http://www.springerlink.com/content/876x51r6647r8g68/>. doi:10.1007/s10626-007-0028-x.
- 5 Khaoula Boukir, Jean-Luc Béchenec, and Anne-Marie Déplanche. Requirement specification and model-checking of a real-time scheduler implementation. In Liliana Cucu-Grosjean, Roberto Medina, Sebastian Altmeyer, and Jean-Luc Scharbarg, editors, *28th International Conference on Real Time Networks and Systems, RTNS 2020, Paris, France, June 10, 2020*, pages 89–99. ACM, 2020. doi:10.1145/3394810.3394817.
- 6 Marc Boyer and Pierre Roux. Embedding network calculus and event stream theory in a common model. In *Proc. of the 21st IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2016)*, Berlin, Germany, September 2016. doi:10.1109/ETFA.2016.7733565.
- 7 Sergey Bozhko and Björn B. Brandenburg. Abstract response-time analysis: A formal foundation for the busy-window principle. In Marcus Völpl, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPICs*, pages 22:1–22:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPICs.ECRTS.2020.22.
- 8 Felipe Cerqueira, Geoffrey Nelissen, and Björn B. Brandenburg. On Strong and Weak Sustainability, with an Application to Self-Suspending Real-Time Tasks. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:21, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPICs.ECRTS.2018.26.
- 9 Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284. IEEE, 2016.
- 10 The Coq development team. *The Coq proof assistant reference manual*, 2020. Version 8.13. URL: <https://coq.inria.fr>.
- 11 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. Certican: A tool for the coq certification of CAN analysis results. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 182–191, 2019. doi:10.1109/RTAS.2019.00023.
- 12 Pascal Fradet, Maxime Lesourd, Jean-François Monin, and Sophie Quinton. A generic coq proof of typical worst-case analysis. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 218–229. IEEE Computer Society, 2018. doi:10.1109/RTSS.2018.00039.
- 13 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669. USENIX Association, 2016. URL: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu>.

- 14 Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. Integrating Formal Schedulability Analysis into a Verified OS Kernel. In *Computer Aided Verification*, pages 496–514, New York, United States, July 2019. doi:10.1007/978-3-030-25543-5_28.
- 15 Xiaojie Guo, Lionel Rieg, and Paolo Torrini. A generic approach for the certified schedulability analysis of software systems. In *RTCSA 2021 - 27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Houston (online), United States, August 2021. URL: <https://hal.archives-ouvertes.fr/hal-03540548>.
- 16 ITU-T. Definitions and terminology for synchronization networks. Technical Report Recommendation G.810, International telecommunication union (ITU), 1996.
- 17 Leonie Köhler, Borislav Nikolić, and Marc Boyer. Increasing accuracy of timing models: From cpa to cpa+. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Florence, Italy, March 2019.
- 18 Simon Künzli, Arne Hamann, Rolf Ernst, and Lothar Thiele. Combined approach to system level performance analysis of embedded systems. In Soonhoi Ha, Kiyong Choi, Nikil D. Dutt, and Jürgen Teich, editors, *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2007, Salzburg, Austria, September 30 - October 3, 2007*, pages 63–68. ACM, 2007. doi:10.1145/1289816.1289835.
- 19 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.
- 20 Etienne Mabilie, Marc Boyer, Loïc Fejoz, and Stephan Merz. Towards certifying network calculus. In *Proc. of the 4th Conference on Interactive Theorem Proving (ITP 2013)*, Rennes, France, July 2013.
- 21 Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*, pages 193–202, New York, NY, USA, 2007. ACM. doi:10.1145/1289927.1289959.
- 22 Lucien Rakotomalala, Marc Boyer, and Pierre Roux. Formal Verification of Real-time Networks. In *JRWRTC 2019, Junior Workshop RTNS 2019*, TOULOUSE, France, November 2019. URL: <https://hal.archives-ouvertes.fr/hal-02449140>.
- 23 Lucien Rakotomalala, Pierre Roux, and Marc Boyer. Verifying min-plus computations with coq. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*, volume 12673 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2021. doi:10.1007/978-3-030-76384-8_18.
- 24 Jonas Rox and Rolf Ernst. Compositional performance analysis with improved analysis techniques for obtaining viable end-to-end latencies in distributed embedded systems. *International Journal on Software Tools for Technology Transfer*, 15(3):171–187, 2013.
- 25 Ludovic Thomas and Jean-Yves Le Boudec. On time synchronization issues in time-sensitive networks with regulators and nonideal clocks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, June 2020 Article No.: 27, 4(2), June 2020. doi:10.1145/3392145.
- 26 Makarius Wenzel. *The Isabelle/Isar Reference Manual*, 2021. Version 2021-1. URL: <https://isabelle.in.tum.de>.

Unikernel-Based Real-Time Virtualization Under Deferrable Servers: Analysis and Realization

Kuan-Hsun Chen ✉ 

University of Twente, The Netherlands

Mario Günzel ✉ 

TU Dortmund University, Germany

Boguslaw Jablkowski ✉

EMVICORE GmbH, Dortmund, Germany

Markus Buschhoff ✉

EMVICORE GmbH, Dortmund, Germany

Jian-Jia Chen ✉ 

TU Dortmund University, Germany

Abstract

For cyber-physical systems, real-time virtualization optimizes the hardware utilization by consolidating multiple systems into the same platform, while satisfying the timing constraints of their real-time tasks. This paper considers virtualization based on unikernels, i.e., single address space kernels usually constructed by using library operating systems. Each unikernel is a guest operating system in the virtualization and hosts a single real-time task.

We consider deferrable servers in the virtualization platform to schedule the unikernel-based guest operating systems and analyze the worst-case response time of a sporadic real-time task under such a virtualization architecture. Throughout synthesized tasksets, we empirically show that our analysis outperforms the restated analysis derived from the state-of-the-art, which is based on Real-Time Calculus. Furthermore, we provide insights on implementation-specific issues and offer evidence that the proposed scheduling architecture can be effectively implemented on top of the Xen hypervisor while incurring acceptable overhead.

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

Keywords and phrases Unikernel, Virtualization, Reservation Servers, Deferrable Servers, Cyber-Physical Systems, Real-Time Systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.6

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.8.1.2>

Funding This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (Project No. 398602212). This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).

1 Introduction

Virtualization technology has been widely and successfully used in data centers and cloud environments to consolidate multiple systems as virtual machines (VMs) into the same platform (so-called host). Due to the increasing use of multi-core processors in embedded and cyber-physical systems (CPS's), platform virtualization now is gaining traction also in these domains [15], since it allows for cost reduction, increases efficiency and enhances flexibility.



© Kuan-Hsun Chen, Mario Günzel, Boguslaw Jablkowski, Markus Buschhoff, and Jian-Jia Chen;

licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 6; pp. 6:1–6:22

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



However, virtualization technology was initially not designed to cope with strict timing constraints and those are inherent to CPS's. In such systems, meeting timing requirements (so-called *timeliness*) is as important as the functional correctness. Depending on the applications, a deadline miss may result in lower service quality or even a catastrophic system failure in the worst case. Thus, virtualization must be compatible to real-time software stack and satisfy the time constraints by employing hypervisor-level real-time scheduling policies.

In order to satisfy timing requirements in virtualized environments, e.g., the popular Xen hypervisor [36, 37], periodic server-based approaches have been widely used – especially *deferrable servers* [33]. For instance, the real-time deferrable server (RTDS) scheduler [35, 36] has been officially¹ supported in Xen since 2015, by which each virtual CPU (vCPU) is treated as one deferrable server assigned with an execution budget and a (replenishment) period to serve its corresponding VMs. The budget of the vCPU is consumed only when a task is running on the vCPU. To ensure each real-time task is served sufficiently, corresponding timing analyses [3, 31, 30] should be employed to compute the required capacity for each server budget. Under such setups, the scheduling decision involves two levels, the hypervisor scheduler and the schedulers within the VMs, in a hierarchical manner.

To account for the interplay of servers and tasks, the applicability of such server-based approaches is based on the tightness of corresponding worst-case response time analyses. Note that a task system deployment under over pessimistic analyses may lead to unnecessarily low hardware utilization. For deferrable servers, Saewong et al. developed a sufficient schedulability analysis based on an assumption that server capacity is made available at the very end of the server's period in the worst case [28]. Davis and Burns further developed an exact test to ultimately optimize the schedulability of deferrable servers [8].

However, they also showed that the schedulability of deferrable servers is worse than the other server-based approaches, like periodic servers [29] and sporadic servers [32], due to the well-known phenomenon of *back-to-back* hits [8, 6], i.e., the interference introduced by the suspension of higher priority servers. As the state-of-the-art, Cuijpers and Bril in [7] discussed that the schedulability of deferrable servers is possible to outperform periodic servers and sporadic servers, if one deferrable server only serves one single task. Under such a constraint, the behavior of the deferrable server is no longer influenced by the presence of low-priority tasks. However, a general task model is considered based on real-time calculus [34].

As reported in an empirical study [1], the periodic task activation and the sporadic activation with minimum inter-arrival time are a common industry practice, i.e., 82% and 47% respectively over the investigated systems, and different types of task activation might be involved in the same systems. Thus, it is practically relevant whether the worst-case response time analysis from [7] can be further tightened for a periodic or sporadic task.

Contributions. In this work, we explore *deferrable servers for unikernel-based virtualization*, in which each server serves only one single sporadic task on a virtualized platform. Specifically, we develop the corresponding worst-case response time analysis under fixed-priority scheduling. In practice, we leverage the concept of *unikernel* to motivate and realize the proposed scheduling architecture. In a nutshell, the contributions of this work are as follows:

- We present why unikernel-based virtualization can facilitate the schedulability of deferrable servers. Specifically, we present how to convert the analysis proposed by Cuijpers and Bril in [7] to sporadic tasks served by deferrable servers. Under one practical scenario, our worst-case response time analysis dominates the state-of-the-art (see Section 4).

¹ https://wiki.xenproject.org/wiki/Xen_Project_Schedulers

■ **Table 1** Notation used in this paper.

Symbol	Definition
$\tau_i = (T_i, C_i)$	Sporadic task
T_i	Minimum inter-arrival time
C_i	Worst-case execution time
$DS_i = (P_i, Q_i)$	Deferrable server
P_i	Replenishment period
Q_i	Capacity
$R_i^{-DS}(x)$	Worst-case <i>response</i> time for x time units on DS_i
$R_i^{+DS}(x)$	Worst-case <i>resumed</i> time for x time units on DS_i

- Secondly, we explain how to realize our unikernel-based approach on top of the Xen hypervisor with a few design details. Under the proposed deferrable server model, we implement our own hypervisor scheduler and keep the routines of scheduler bookkeeping and budget replenishment as efficient as possible (see Section 5).
- Finally, we compare our analysis with the state-of-the-art [7] with synthetic periodic task systems. The results show that the applicability of deferrable servers indeed can be greatly improved (see Section 6.1). In addition, we also conduct a case study based on the Xen hypervisor with our deferrable server model and show that our approach of unikernel-based virtualization is feasible in practice (see Section 6.2).

2 Deferrable Server and Task model

We consider that deferrable servers [33] are adopted to preserve the required bandwidth of each virtualized CPS application, so-called virtual machine (VM). Since each VM is realized as a unikernel with one specific vCPU, each VM is treated as one deferrable server DS_i serving only one single sporadic task τ_i . A sporadic task τ_i releases an infinite number of task instances, called jobs, in which the worst-case execution time (WCET) of any of them is at most C_i and the arrival times of any two consecutive jobs of them must be separated by at least the minimum inter-arrival time T_i . The jobs of task τ_i are served based on the first-come first-serve policy within DS_i . Please note that our worst-case response time analysis does not require any limitation on the type of task deadlines, i.e., even arbitrary deadline tasks with deadline $D_i > T_i$ are allowed.

Each deferrable server DS_i is denoted as a tuple (P_i, Q_i) , where P_i is its replenishment period and Q_i is its capacity. If the j -th replenishment time is t , then the next replenishment time is $t + P_i$. The budget of a deferrable server DS_i is set to Q_i initially and is consumed linearly while the corresponding task τ_i is served. When the budget becomes 0, the server DS_i needs to wait until the next replenishment time. At the time instant to replenish the server DS_i , the budget is replenished to Q_i and any unused time budget is lost at the end of each replenishment period. The first job release of every task τ_i is assumed to be after the first budget replenishment of DS_i .

Deferrable servers are scheduled based on preemptive fixed-priority (static-priority) scheduling. For a multiprocessor platform, it is possible to apply partitioned or global scheduling for the deferrable servers. Under a partitioned scheduling paradigm, a deferrable server (vCPU) is dedicated to one physical processor (pCPU). Under a global scheduling paradigm, a deferrable server (vCPU) can migrate from one physical processor to another.

3 Service Condition and Execution Scenarios of Deferrable Servers

In this section we discuss how the service condition of deferrable servers can be sufficiently tested on different systems. By analyzing the possible behaviors of a job served by a deferrable server, two lemmas are derived to provide some useful properties for the next section.

3.1 Service Condition of Deferrable Servers

Under the adopted scheduling paradigm, if the capacity of a deferrable server DS_k within the given replenishment period is feasible, we say that DS_k *fulfills its service condition*. That is, in this case, any request of Q_k amount of computation demand of its served task τ_k at the moment when the budget is fully replenished must be finished within P_k amount of time. This assumption is also required in the related work [7, 8, 28, 20]. Otherwise, one can provide an over-specified capacity Q_k that can never be guaranteed within one period P_k , and the worst-case analysis also has to investigate interplay with the virtualization scheduler.

As a deferrable server DS_i retains its unused budget until the next replenishment period when no job requests it to serve, it can be considered that DS_i voluntarily suspends its execution [6]. As a result, a DS_i may impose back-to-back interference to lower-priority servers (or tasks). Such back-to-back interference can be considered as bursty interference [20, 5], i.e., one additional job should be considered by extending the classical critical instant theorem, or release jitter of DS_i , which can be set to $P_i - Q_i$ [8].

A sufficient service condition test for DS_k is a sufficient test to validate whether DS_k fulfills its service condition or not. Therefore, under uniprocessor preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers, also stated in [8], a sufficient service condition test for the deferrable servers is

$$\forall DS_k, \exists 0 < t \leq P_k, \quad Q_k + \sum_{DS_i \in hp_k} \left\lceil \frac{t + P_i - Q_i}{P_i} \right\rceil Q_i \leq t \quad (1)$$

where hp_k is the set of deferrable servers whose priorities are higher than DS_k . It has been shown that $\sum_{DS_i} \frac{Q_i}{P_i} \leq \ln \frac{3}{2} \approx 0.40546$ ensures that the condition stated in Eq. (1) holds [5, 20].²

For multiprocessor systems, under partitioned scheduling, the condition in Eq. (1) can be directly applied by defining hp_k as the set of higher-priority deferrable servers assigned on the same physical processor as DS_k . Under global preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers on m homogeneous (identical) physical processors, a sufficient service condition test can be written as³

$$\forall DS_k, \exists 0 < t \leq P_k, \quad Q_k + \frac{\sum_{DS_i \in hp_k} \left\lceil \frac{t + P_i - Q_i}{P_i} \right\rceil Q_i}{m} \leq t \quad (2)$$

where hp_k is the set of deferrable servers whose priorities are higher than DS_k under global scheduling and m is the number of homogeneous processors.

Although the sufficient service condition tests in Eqs. (1) and (2) are valid for systems composed of only deferrable servers, they can be extended to consider the co-existence of typical sporadic tasks and other fixed-priority servers by adding corresponding interference terms. More specifically, let $I(t)$ be the worst-case interference of the higher-priority servers and/or tasks for an interval length t . For the rest of this paper, we assume that the sufficient service condition for DS_k is:

$$\exists 0 < t \leq P_k, \quad Q_k + I(t) \leq t \quad (3)$$

² Their proof in [20] ensures a weaker condition: $\forall DS_k, \exists 0 < t \leq P_k, Q_k + \sum_{DS_i \in hp_k} \left\lceil \frac{t + P_i}{P_i} \right\rceil Q_i \leq t$.

³ A sketched proof is provided in Appendix for completeness.

In order to analyze the worst-case response time of a sporadic task τ_k served by DS_k , we further need two additional properties based on finer granularity of the service provided by DS_k . In particular, the worst-case response time $R_k^{-DS}(x)$ for requesting x amount of computation demand, for $0 \leq x \leq Q_k$, can be derived as:

$$R_k^{-DS}(x) = \inf \{t | x + I(t) = t\} \quad (4)$$

Moreover, right after finishing x amount of computation demand, the deferrable server DS_k may be preempted by other higher-priority activities. We further define the *worst-case resumed time* $R_k^{+DS}(x)$, for $0 \leq x < Q_k$, as the longest time that DS_k finishes x amount of computation demand and is scheduled to serve further demands if they exist. That is:

$$\begin{aligned} R_k^{+DS}(x) &= \inf \{t | (x + \epsilon) + I(t) = t\} \text{ for infinitesimal } \epsilon > 0 \\ &= \inf \{t | x + I(t) < t\} \end{aligned} \quad (5)$$

We note that $R_k^{+DS}(Q_k)$ is not defined above, as it is unnecessary in our analysis, and the proper definition involves more complications.

Since $I(t)$ is usually of the form $\sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \rho_i$, where j_i and ρ_i are some real values like $j_i = (P_i - Q_i)$ and $\rho_i = Q_i$ in Equation (1), the exact value of $R_k^{-DS}(x)$ can be computed using fixed-point iterations, where t is increased gradually until $x + I(t) = t$ is reached. The exact value of $R_k^{+DS}(x)$ can be obtained by fixed-point iterations as well, if the standard interference function $I(t)$ is replaced by $\sum_i \left(\left\lceil \frac{t+j_i}{P_i} \right\rceil + 1 \right) \rho_i$.

Please note that the sufficient service condition test, i.e., Eq. (3), is only introduced to give an intuition behind the definition of $R_k^{-DS}(x)$ and $R_k^{+DS}(x)$. However, our analysis is not limited to the scenarios where Eq. (3) holds, but only to all possible scenarios where the service condition holds, i.e., any request of Q_k amount of computation demand must be finished within P_k amount of time when the budget is fully replenished. If this condition is ensured by any means, our timing analysis is applicable.

3.2 Serving one Task by a Deferrable Server

In this section we have a closer look at how a deferrable server DS_k serves a job J released by the task τ_k at time r_J . At the job release of J there may be unfinished backlog $L(r_J)$, i.e., unfinished execution demand from previously released jobs of τ_k . Moreover, we denote by $C(r_J) \leq C_k$ the computation demand of J at its release r_J . Whenever there is available budget $B(t) > 0$ of the server DS_k , the budget can be used to serve first the backlog $L(t)$ and if the backlog reaches $L(t) = 0$ then the budget is used to serve the computation demand $C(t)$ of J . The first time the computation demand reaches 0 is called the finish f_J of J , i.e., we have $C(f_J) = 0$.

If at the release r_J of the job J there is enough budget to complete both the backlog $L(r_J)$ and the computation demand $C(r_J)$ then the job J finishes as soon as additional $L(r_J) + C(r_J)$ budget is consumed.

► **Lemma 1.** *If a job J of task τ_k is released at time r_J and the remaining budget of DS_k is higher than the execution demand $C(r_J)$ of J at time r_J plus the backlog $L(r_J)$ from previous jobs of τ_k at time r_J , then J finishes within $R_k^{-DS}(L(r_J) + C_k)$ time units.*

Proof. If the budget is higher than the execution time plus the backlog at time r_J , then J and all previously released unfinished jobs can execute whenever the server DS_k is not interfered. The job J finishes when $L(r_J) + C(r_J)$ amount of computation demand is finished, which is after at most $R_k^{-DS}(L(r_J) + C_k)$ time units. ◀

In particular whenever there is no backlog and the remaining budget at time r_J is at least C_k time units, then the job J finishes within $R_k^{-\text{DS}}(C_k)$ time units.

However, there are cases in which the budget is not sufficient:

► **Definition 2** (Exhausted budget). *We say the budget B is exhausted by job J if there exists a point in time t such that the following conditions are met:*

- *There is remaining computation demand $C(t) > 0$ that wants to consume the budget.*
- *The budget $B(t) = 0$ has reached 0.*
- *Instead the processor idles or a lower priority server (or task) is served.*

In such a case the jobs have to wait until the next budget replenishment br to be served. If after the budget replenishment the budget $B(br) = Q_k$ is sufficient to finish the backlog and computation demand, i.e., $Q_k \geq L(br) + C(br)$, then the worst-case response time of J can be described by the following lemma.

► **Lemma 3.** *If after a budget replenishment at time br the remaining backlog $L(br)$ and the remaining computation demand $C(br)$ can be fully served, i.e., $Q_k \geq L(br) + C(br)$, then the job has a response time of at most $(br - r_J) + R_k^{-\text{DS}}(L(br) + C(br))$ time units.*

Proof. Similar to Lemma 1, at time br the remaining computation demand of $L(br) + C(br)$ has to be finished. Since the budget is high enough, i.e., $Q_k \geq L(br) + C(br)$, this takes at most $R_k^{-\text{DS}}(L(br) + C(br))$ time units. The response time of J is the result of the time it takes from the release of J until br plus the time to finish the remaining computation demand, i.e., $(br - r_J) + R_k^{-\text{DS}}(L(br) + C(br))$. ◀

4 Worst-Case Response Time Analysis for One Single Task

In this section, we provide a worst-case response time analysis for a sporadic task τ_k served by a deferrable server DS_k , which fulfills its service condition. That is, throughout this section, we implicitly assume that $R_k^{-\text{DS}}(Q_k) \leq P_k$. Furthermore, the utilization of task τ_k is assumed to be no more than the utilization of the deferrable server DS_k , i.e., $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$; otherwise, the worst-case response time of task τ_k is by definition unbounded.

We first explain how to convert the analysis in [7] based on real-time calculus to sporadic tasks served by DS_k . Then, we provide our analysis for a scenario that $T_k \geq P_k$ and $C_k \leq Q_k$, and demonstrate the dominance of our analysis over the analysis in [7] when considering sporadic tasks in Section 4.2. As our analysis requires to evaluate $\sup_{0 \leq x < C_k} R_k^{+\text{DS}}(x) + R_k^{-\text{DS}}(C_k - x)$, we explain how to implement this search in Section 4.3.

4.1 Existing Analysis Converting from Real-Time Calculus

We restate here the analysis from Cuijpers and Bril [7] based on real-time calculus for sporadic tasks. Note that we use a slightly different notation system from that in [7] due to notation discrepancy between real-time calculus and real-time scheduling theory. Suppose that $r_k(t)$ is the accumulated workload in time interval $[0, t)$ for task τ_k .

Let S be some $S > 0$ such that

$$r_k(s + S) - r_k(s) \leq \frac{Q_k}{P_k} \times S, \quad \forall s \geq 0 \quad (6)$$

Under the assumption that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$, setting S to $C_k \times \frac{P_k}{Q_k}$ ensures that

$$S = C_k \times \frac{P_k}{Q_k} \leq C_k \times \frac{T_k}{C_k} = T_k \text{ and } r_k(s + S) - r_k(s) \leq C_k = \frac{C_k}{S} \times S = \frac{Q_k}{P_k} \times S,$$

i.e., Eq. (6) holds.

As for the deferrable server DS_k , if it has a full budget at time t , its service provision from time interval t to $t + R_k^{-DS}(h)$ is at least h . This notation is the inverse representation of the accumulative service in real-time calculus under the same assumption. In terms of real-time calculus, DS_k is guaranteed to provide at least h amount of service within an interval length of $R_k^{-DS}(h)$.

Let h be the minimum value $\leq Q_k$ such that

$$\frac{h}{R_k^{-DS}(h)} \geq \frac{r_k(s+S) - r_k(s)}{S}, \quad \forall s \geq 0. \quad (7)$$

Let H denote $R_k^{-DS}(h)$ for brevity. Cuijpers and Bril [7] showed that the worst-case response time of τ_k is upper bounded by $S + 2H$.

With the above definitions of H and S , we can restate the worst-case response time analysis from Cuijpers and Bril [7] for sporadic tasks.

► **Theorem 4.** *Suppose that the deferrable server DS_k fulfills its service condition and that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$. Then*

- *S can be set to $C_k \times \frac{P_k}{Q_k} \leq T_k$*
- *H is upper bounded by $R_k^{-DS}(Q_k) \leq P_k$*

The worst-case response time of a sporadic task τ_k served by DS_k is upper bounded by

$$C_k \times \frac{P_k}{Q_k} + 2R_k^{-DS}(Q_k) \leq C_k \times \frac{P_k}{Q_k} + 2P_k \leq T_k + 2P_k.$$

Proof. It comes directly from Theorem 1 in [7] and the above analysis of S and H . ◀

4.2 Our Analysis for Sporadic Tasks

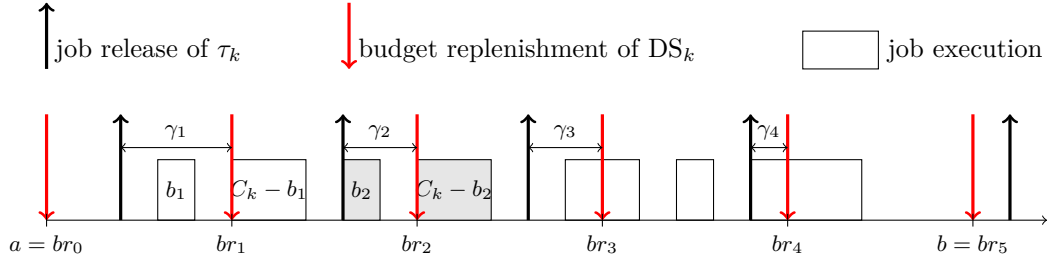
One practical scenario is that the replenishment period P_k is set to its served task period T_k , and the capacity Q_k is set to C_k , i.e., $T_k = P_k$ and $C_k = Q_k$. Such a configuration might be out of intuition by expecting the period alignment is perfect. Assuming the timing behaviors of servers are the same as tasks, the Liu and Layland bound [19] might be applicable at the first glance. However, the response time of a task may still be interfered by a backlog of the previous unfinished job due to any potential misalignment. Hence, a corresponding timing analysis is still needed.

The analysis in Theorem 4 is applicable for sporadic tasks. However, for the scenario with $T_k \geq P_k$ and $C_k \leq Q_k$, in this section, we show that a tighter analysis can be achieved by examining the interplay between τ_k and DS_k more closely.

► **Definition 5** (Consecutive DS service interval). *An interval $G = [a, b) \subseteq \mathbb{R}$ is called a consecutive DS service interval if it is a minimal interval such that the following properties are met:*

- *a and b are time instances where DS_k replenishes its budget.*
- *All jobs of τ_k that are released during G finish their execution during G .*
- *Only jobs of τ_k that are released during G can be executed.*

A consecutive DS service interval can be constructed in the following way: Let a be a time instant such that 1) DS_k replenishes its budget at a , 2) there is no unfinished job of τ_k at time a , and 3) a job of τ_k is released during $[a, a + P_k)$. Then, we set b to be the time of the next replenishment of DS_k such that there is again not unfinished job of τ_k at time b . The interval $[a, b)$ is a consecutive DS service interval. Please note that every job is inside a consecutive DS service interval as the above procedure to construct consecutive DS service intervals can be repeated for the whole time domain.



■ **Figure 1** Analysis scenario for Theorem 7. We analyse the second job J_2 (marked in grey) in the consecutive DS service interval $[a, b)$.

The first job in any consecutive DS service interval receives budget Q_k . Under the assumption that $C_k \leq Q_k$ this job finishes after at most $R_k^{\text{DS}}(C_k)$ time units.

► **Lemma 6.** *Under the assumption that $C_k \leq Q_k$ and $T_k \geq P_k$, the following holds:*

1. *Between any two budget replenishments there is at most one job release.*
2. *Every job finishes until the second replenishment period after the job release.*
3. *There is at most one previous unfinished job of τ_k at any job release of τ_k .*
4. *There are at most two jobs of τ_k executed between two consecutive budget replenishments of DS_k .*

Proof. 1: If there would be two job releases between two budget replenishments, then $T_k < P_k$, which contradicts our assumption.

2: By contradiction: Assume that J is the earliest job such that it does not finish until the second replenishment period after its release. Let br_{i-1} and br_i the two consecutive budget replenishments before and after the release of J . By **1**, the previous job J' is released before br_{i-1} . Moreover, since J is the first job such that **2** does not hold, J' finishes until br_i . At time br_i there is no backlog from J' or from earlier jobs. The computation demand at br_i by J is at most $C_k \leq Q_k$. By Lemma 3 the job J finishes until $br_i + R_k^{\text{DS}}(C_k) \leq br_i + P_k$ by the sufficient service assumption stated in Section 3.1.

3 and 4: Follow directly from **1** and **2**. ◀

► **Theorem 7.** *Suppose that the deferrable server DS_k fulfills its service condition and that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$. If $C_k \leq Q_k$ and $T_k \geq P_k$, then*

$$R_k^r \leq \max \left((P_k - T_k) + \sup_{0 \leq x < C_k} (R_k^{\text{DS}}(x) + R_k^{\text{DS}}(C_k - x)), R_k^{\text{DS}}(C_k) \right) \quad (8)$$

Proof. We prove this theorem over induction of the jobs in each consecutive DS service interval. Let G be some consecutive DS service interval. We show by induction that for all jobs that are released during G , the upper bound on the worst-case response time given by Eq. (8) holds. We denote by J_i the i -th job that is released by τ_k during G . Moreover, we denote by γ_i the time between the release of J_i and the subsequent budget replenishment br_i . Due to the assumption that $T_k \geq P_k$, we have $\gamma_1 \geq \gamma_2 + (T_k - P_k) \geq \gamma_3 + 2(T_k - P_k) \geq \dots$ as demonstrated in Figure 1. By Lemma 6 there is at least one budget replenishment between any two consecutive job releases. Indeed, there is exactly one budget replenishment between two consecutive job releases since otherwise the earlier job would finish until the second budget replenishment after its release and the consecutive DS service interval would end. Therefore J_i is always released during br_{i-1} and br_i . We define b_i as the time that J_i is executed before br_i and $R_{k,i}^r$ be the response time of the job J_i . For the intermediate jobs J_i in the consecutive DS service interval we show that the following holds:

	backlog	no backlog
budget not exhausted between release of J_i and br_i	(not possible) (Case 2)	$R_{k,i}^\tau \leq R_k^{-\text{DS}}(C_k)$ $\gamma_i \leq R_k^{+\text{DS}}(b_i)$ (Case 1)
budget exhausted between release of J_i and br_i	$R_{k,i}^\tau \leq (P_k - T_k) + R_k^{+\text{DS}}(b_i) + R_k^{-\text{DS}}(C_k - b_i)$ $\gamma_i \leq R_k^{+\text{DS}}(b_i)$ (Case 3)	

Moreover, for the last job J_i in consecutive DS service intervals (if a last job exists) we show that the following holds:

	backlog	no backlog
budget not exhausted between release of J_i and br_i	$R_{k,i}^\tau \leq R_{k,i-1}^\tau$ (Case 2)	$R_k^{-\text{DS}}(C_k)$ (Case 1)
budget exhausted between release of J_i and br_i	$R_{k,i}^\tau \leq (P_k - T_k) + R_k^{+\text{DS}}(b_i) + R_k^{-\text{DS}}(C_k - b_i)$ $\gamma_i \leq R_k^{+\text{DS}}(b_i)$ (Case 3)	

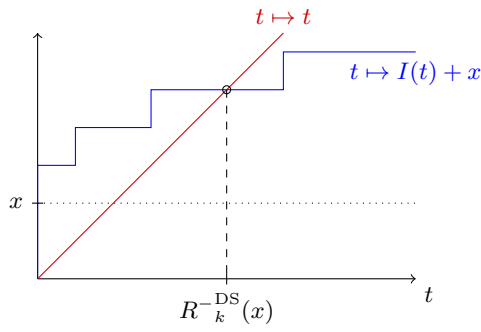
First job J_1 . At the release of J_1 there is $Q_k \geq C_k$ amount of budget and no backlog from previous jobs. Therefore, the budget is not exhausted between the release of J_1 and the next budget replenishment. Hence, the first job is always in Case 1. By Lemma 1 the response time $R_{k,1}^\tau$ of J_1 is upper bounded by $R_k^{-\text{DS}}(C_k)$. If J_1 is an intermediate job, then it does not finish before br_1 . In particular an execution demand of b_1 time units could be served between the release of J_1 and br_1 , although there was enough budget available. We conclude that the worst-case resumed time $R_k^{+\text{DS}}(b_1)$ has to be at least γ_1 .

Induction step $J_{i-1} \rightarrow J_i$. Under the assumption that J_{i-1} is an intermediate job, we show that for J_i the bounds presented in the tables still hold.

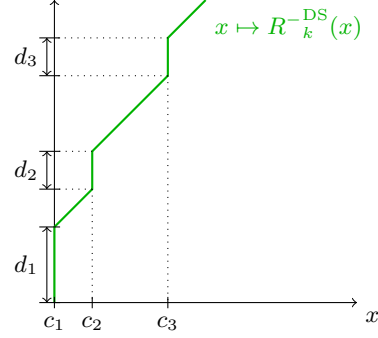
Case 1: The budget is *not* exhausted between the release of J_i and br_i , and there is *no* backlog from J_{i-1} . This job behaves analogously to the first job J_1 . The worst-case response time $R_{k,i}^\tau$ is upper bounded by $R_k^{-\text{DS}}(C_k)$ due to Lemma 1. If J_i is the last job in G , then this is the only bound from the table to be proven. Otherwise, $\gamma_i \leq R_k^{+\text{DS}}(b_i)$ holds since only an execution demand of b_i time units during the interval of length γ_i could be served.

Case 2: The budget is *not* exhausted between the release of J_i and br_i , and there is backlog from J_{i-1} . In this case there is demand at all times during br_{i-1} and the release of J_i . If J_i would not finish before br_i , then the budget would be exhausted no later than at time $br_{i-1} + R_k^{-\text{DS}}(Q_k) \leq br_i$. This is not possible by the assumption of this case. Therefore in this case J_i must finish before br_i and is therefore always the last job in the consecutive DS service interval G . Since the job finishes before br_i we have $R_{k,i}^\tau \leq \gamma_i \leq \gamma_{i-1}$ which is upper bounded by $R_{k,i-1}^\tau$ since J_{i-1} has to finish after br_{i-1} to produce a backlog for J_i .

Case 3: The budget is exhausted between the release of J_i and br_i . For this case, it is irrelevant if J_i is an intermediate or the last job since the same properties have to be proven. Since by Lemma 6 the job finishes until the second budget replenishment after its release, this means at the replenishment time br_i there is no backlog from the previous job J_{i-1} . Moreover, the remaining computation demand from J_i is at most $C_k - b_i \leq Q_k$. By Lemma 3 the response time of J_i is upper bounded by $\gamma_i + R_k^{-\text{DS}}(C_k - b_i)$. Moreover, we know that $\gamma_i \leq (P_k - T_k) + \gamma_{i-1} \leq (P_k - T_k) + R_k^{-\text{DS}}(b_{i-1})$. Since J_{i-1} has to be served by DS_k for $Q_k - b_i$ time units after br_{i-1} such that J_i exhausts the budget of



■ **Figure 2** Computation of $R_k^{-DS}(x)$ by finding the intersection of two functions.



■ **Figure 3** Shape of $R_k^{-DS}(x)$.

DS_k before br_i , we have $b_{i-1} \leq C_k - (Q_k - b_i)$, resulting in $b_{i-1} \leq b_i$. We conclude that $\gamma_i \leq (P_k - T_k) + \gamma_{i-1} \leq (P_k - T_k) + R_k^{-DS}(b_{i-1}) \leq (P_k - T_k) + R_k^{-DS}(b_i) \leq R_k^{-DS}(b_i)$ and the worst-case response time is upper bounded by $(P_k - T_k) + R_k^{+DS}(b_i) + R_k^{-DS}(C_k - b_i)$.

Conclusion. By induction we have proven that the bounds from the above stated tables hold. Since $b_i < C_k$ holds for those jobs with exhausted budget, the response time bound in Eq. (8) holds for all jobs by analyzing all consecutive DS service intervals. ◀

► **Theorem 8** (Dominance discussion). *The worst-case response time bound presented in Theorem 7 dominates the bound from Theorem 4 when $C_k \leq Q_k$ and $T_k \geq P_k$.*

Proof. As $(P_k - T_k) \leq 0$ by assumption, the worst-case response time bound provided in Theorem 7 is upper bounded by $\sup_{0 \leq x < C_k} R_k^{+DS}(x) + R_k^{-DS}(C_k)$. Since $R_k^{+DS}(x) \leq R_k^{-DS}(C_k)$ for all $x < C_k$, the bound from Theorem 7 is also upper bounded by $2R_k^{-DS}(C_k) \leq C_k \cdot \frac{P_k}{Q_k} + 2R_k^{-DS}(Q_k)$ which is the bound from Theorem 4. ◀

4.3 Efficient Computation of Worst-Case Response Time Bound

For the computation of the worst-case response time upper bound presented in Theorem 7 the supremum

$$\sup_{0 \leq x < C_k} R_k^{+DS}(x) + R_k^{-DS}(C_k - x) \quad (9)$$

has to be computed. In this section we discuss a method to do this efficiently without computing the values for R_k^{+DS} and R_k^{-DS} at every point using fixed-point iterations. As presented in Section 3.1, $I(t)$ is of the form $\sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \rho_i$ for some positive real values j_i, P_i and ρ_i . Fixed-point iterations can be used to compute the value of $R_k^{-DS}(x)$, in particular the intersection between the functions $t \mapsto t$ and $t \mapsto I(t) + x$ is computed, as presented in Figure 2.

The efficient presentation and formulation presented in this section is based on the observation that $R_k^{-DS}(x)$ and $R_k^{+DS}(x)$ coincide and grow linearly if the intersection with $I + x$ is on a plateau, i.e., if I is constant during the interval $(R_k^{-DS}(x) - \delta, R_k^{-DS}(x) + \delta)$ then $R_k^{-DS}(y) = R_k^{-DS}(x) + (y - x)$ for all $y \in (x - \delta, x + \delta)$. At those points x where there is a jump of I at $R_k^{-DS}(x)$, the values of $R_k^{+DS}(x)$ and $R_k^{-DS}(x)$ are computed using fixed-point

■ **Algorithm 1** Computation of all values in \mathcal{S} with $c_i \leq C_k$.

Input: $I(t) = \sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \cdot \rho_i$

Output: The set \mathcal{S} with all values (c_i, d_i) where $c_i \leq C_k$.

```

1:  $\mathcal{S} := []$ ;  $x := 0$ 
2: while  $x \leq C_k$  do
3:   Compute  $R_k^{-\text{DS}}(x)$  and  $R_k^{+\text{DS}}(x)$  by fixed-point iterations.
4:    $c := x$ ;  $d := R_k^{+\text{DS}}(x) - R_k^{-\text{DS}}(x)$ 
5:   Add  $(c, d)$  to the set  $\mathcal{S}$ 
6:    $x := x + \min_i (-(x + j_i) \bmod P_i)$  ▷ Time until next jump.
7: return  $\mathcal{S}$ 

```

iterations. The shape of $R_k^{-\text{DS}}$ is presented in Figure 3. Please note that the shape of $R_k^{-\text{DS}}$ and $R_k^{+\text{DS}}$ coincide during the linear parts and only at the jumps (c_1, c_2, \dots) the function $R_k^{-\text{DS}}$ takes the lower value and $R_k^{+\text{DS}}$ takes the higher value.

Based on the shape of the functions $R_k^{-\text{DS}}$ and $R_k^{+\text{DS}}$, there exists a set of tuples $\mathcal{S} = \{(c_1, d_1), (c_2, d_2), (c_3, d_3), \dots\}$ such that

$$R_k^{-\text{DS}}(x) = x + \sum_{(c_i, d_i) \in \mathcal{S}} \chi_{x > c_i} \cdot d_i \quad \text{and} \quad R_k^{+\text{DS}}(x) = x + \sum_{(c_i, d_i) \in \mathcal{S}} \chi_{x \geq c_i} \cdot d_i \quad (10)$$

where $\chi_{x > c_i}$ is 1 if $x > c_i$ and 0 else, and $\chi_{x \geq c_i}$ is 1 if $x \geq c_i$ and 0 else. The computation of the values of \mathcal{S} is presented in Algorithm 1. In each step of the while-loop the size of a jump is computed by the difference between $R_k^{-\text{DS}}$ and $R_k^{+\text{DS}}$, and the corresponding tuple is added to the set \mathcal{S} . Afterwards, the time until the next jump is computed by finding the next point in time where the intersection t with $x + I(t)$ reaches a jump, i.e., after $\min_i (-(x + j_i) \bmod P_i)$ additional time units.

With the representation of $R_k^{-\text{DS}}$ and $R_k^{+\text{DS}}$ achieved in Eq. (10), the supremum in Eq. (9) can be rewritten as

$$\sup_{0 \leq x < C_k} R_k^{+\text{DS}}(x) + R_k^{-\text{DS}}(C_k - x) \quad (11)$$

$$= x + \sum_{(c_i, d_i) \in \mathcal{S}} \chi_{x \geq c_i} \cdot d_i + \sup_{0 \leq x < C_k} x + \sum_{(c_i, d_i) \in \mathcal{S}} \chi_{C_k - x > c_i} \cdot d_i \quad (12)$$

$$= C_k + \sum_{(c_i, d_i) \in \mathcal{S}} (\chi_{x \geq c_i} + \chi_{x < C_k - c_i}) \cdot d_i \quad (13)$$

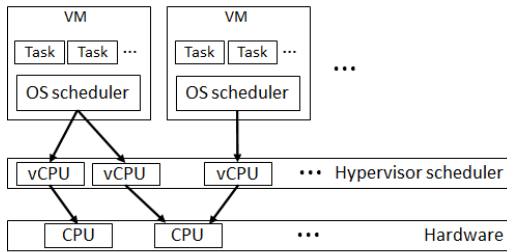
In particular, only finitely many cases have to be checked to find the exact solution of the supremum formulated in Eq. (9).

5 Architecture Model

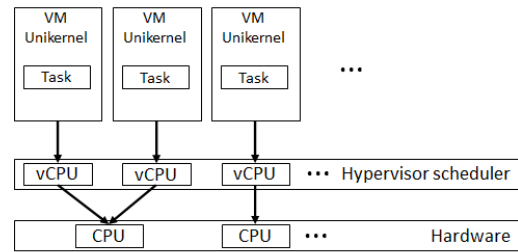
In this section, we present our unikernel-based architecture for hosting virtualized CPS applications. First, we introduce the concept of unikernel-based CPS applications. Next, we shortly describe the Xen architecture as well as its implications for scheduling and compare it with our approach. Finally, we provide some design and implementation details.

5.1 Unikernel-based CPS Applications

In data-centers and cloud environments, each VM is expected to host a general purpose operating system to ease the effort of porting legacy software with a lot of inherent libraries and functionalities. However, most CPS applications are functionally dedicated, single-purposed and thus not dependent on additional functionality. Due to the encapsulation of



■ **Figure 4** depicts the typical scheduling architecture in platform virtualization: The scheduling decision is split into two layers, the OS scheduler of the guest and the hypervisor scheduler.



■ **Figure 5** illustrates our approach where scheduling decisions are reduced to one layer only, due to the adaptation of unikernels.

superfluous libraries and the execution of non-essential processes that are not related to the task of interest, deploying real-time tasks of CPS on a general-purpose OS in fact arises various issues like resource efficiency and timing predictability.

Towards this, several efforts have been made in recent years for the realization of single-purpose appliances [21, 22, 24, 26], so called *unikernels*. These are sealed, single-purpose VM images that can be constructed using the concept of library operating systems (LibOS) [12, 13]. LibOS's allow for the tailoring of an OS code base to the particular needs of a given task, by which only those parts of the OS API are included into the VM image.

This approach has several advantages. Unikernels are characterized by a minimal VM image size which highly increases their security properties, due to the minimal attack surface for malicious code injections. This also translates to a substantial reduction of overall system resource usage. Moreover, unikernels can be instantiated and become fully functional within only a few milliseconds. It has further been shown that unikernels are more efficient and safer than modern container technologies [23]. Please note that while unikernels have indeed a minimal attack surface in comparison with full-fledged operating systems and are fully isolated from other guests, depending on the given scenario, some additional security defence mechanisms would have to be adopted, but considered out of scope in this work.

Considering these benefits of unikernels, as well as the fact that most CPS applications are specialized and functionally dedicated tasks and can be implemented as single-purpose appliances, CPS applications provide an excellent target for unikernels.

5.2 Scheduling Architecture

The case-study results presented in this paper are linked to the Xen hypervisor [2], which we have chosen to realize our unikernel-based approach for hosting virtualized CPS applications. Xen is a type 1 hypervisor and allows for the consolidation of multiple systems on a single platform. Xen runs directly on host's hardware and is the first software layer to execute after the bootloader. The hypervisor is responsible for managing hardware resources, including CPUs and memory. It also handles timers and the scheduling of VMs. Specific to Xen is a privileged VM called Domain 0. It is the first VM to load under Xen, it holds the drivers to the underlying hardware and this is also where the toolstack resides that enables management of further VMs. Typically Domain 0 is deployed on Linux. In order to make use of the existing drivers in Domain 0, Xen uses an approach called paravirtualization, which exposes an API to the guest VM for delegating privileged instructions, including driver calls. This approach is much more efficient than emulation. The backbone of the paravirtualization

driver concept under Xen is the split device driver model. The drivers consist of two parts: the front-end and the back-end. The front-end is situated in the guest VM while the back-end resides in Domain 0. Both parts are isolated and communicate through shared memory.

As shown in Figure 4, the scheduling decision in Xen is split into two tiers. The bottom tier is constituted by the hypervisor scheduler which assigns virtual CPUs (vCPU) to physical CPUs (pCPU). The second tier consists of the guest operating system schedulers within virtual machines, which in turn assign their threads to vCPUs. This split is needed for two reasons. The first being the possibility to abstract physical resources (e.g. pCPUs) into logical resources (e.g. vCPUs) which is a premise for achieving better hardware utilization. Secondly, it allows the hypervisor to enforce timing isolation between the concurrently running VMs. For systems that comprise of task sets without strict timing requirements and that implement a fair share scheduler, this architecture allows for a high resource utilization while at the same time preventing any faulty or compromised VM from hijacking system resources, and by this from negatively influencing the behavior of other VMs in the system. However, when it comes to the need of providing timing guarantees, this scheduling architecture also complicates the corresponding schedulability analysis. As discussed in Section 3.1, the suspension behavior of deferrable servers additionally interferes with the response time of lower priority servers or tasks in the worst case. While improving the applicability of deferrable servers in Section 4.3, our approach, i.e., one sporadic task per server, aligns well with the concept of unikernels, by which CPS applications are deployed as single tasked VMs. As shown in Figure 5, this renders the scheduler instances inside the VMs obsolete. Due to the fact that each of our unikernels hosts a single task, there is no point of assigning more than one vCPU per unikernel.

5.2.1 Real-Time Networking

As in practice, CPS applications commonly assume distributed architectures, the I/O processing of network packets is a matter of particular importance. Xen handles packet processing in Domain 0 where the network driver resides. Each instantiated VM under Xen is connected to a dedicated virtual network interface (VIF) and a corresponding dedicated VIF-thread. This is the context where the actual packet processing takes place. Unfortunately, by default, the VIF-threads in Xen are scheduled independently of the priority of their VMs. This can lead to priority violation, i.e., the order of packet processing mismatches the priorities of vCPUs. In order to solve this issue, we align the priorities of the packet processing threads with the corresponding priorities of the vCPUs in the hypervisor scheduler.

5.3 Design Principles

In our model, each vCPU is implemented as a deferrable server and is described by its *capacity* and *replenishment period*. Due to our adaptation of unikernels, each vCPU has only one task assigned to it. The vCPU is released under the sporadic task activation. The budget denotes the amount of time a vCPU can consume for its execution during the replenishment period. The budget of the vCPU is set to the given capacity at its replenishment periodically. The vCPU can either be runnable or blocked. While running, the vCPU consumes its budget. A vCPU with a depleted budget will not be scheduled. If there are no eligible (runnable and with budget) vCPUs, the hypervisor will schedule an idle vCPU. As vCPUs are implemented as deferrable servers, they can defer their budget to be used at a later time. However, the budget cannot be preserved and transferred into the next period. Budgets that were not consumed during their current periods are lost.

Our implementation relies on partitioned queues, i.e., each pCPU possesses and manages its own run queue of vCPUs. That is, the scheduling model is under (multiprocessor) partitioned scheduling paradigm. The priorities are statically assigned to the vCPUs according to their replenishment periods, following the *rate-monotonic* (RM) policy. As mentioned in Section 3.1, the utilization bound $\ln(3/2) \approx 0.40546$ guarantees the sufficient service condition of the deferrable servers on one physical processor, whereas a tighter analysis can be achieved by adopting Eq. (1). Each time a vCPU is assigned to a run queue, the vCPU is inserted accordingly to its priority. In the case of the RM algorithm, the highest priority is given to the VM with the shortest replenishment period. In this process also the priorities of all lower prioritized vCPUs are updated. Scenarios where a vCPU is assigned to a run queue include the instantiation of a new VM or an existing VM becoming runnable. Blocked vCPUs are removed from the run queue.

5.4 Implementation on Xen

The Xen hypervisor provides an interface to schedulers by exposing an abstract scheduler struct which contains pointers to functions which have to be implemented when adding a new scheduler to Xen. The scheduler policy independent code is situated in the `scheduler.c` file. All of the functions defined in the interface have analogs in this file. It comprises scheduler policy independent code which after execution calls the specialized functions from a specific scheduler implementation. The most important function in `schedule.c` is `schedule()`. As the name suggests, this function is executed when a scheduling decision is needed. In order to choose the next vCPU to run, it deschedules the currently running vCPU and calls the specialized `do_schedule()` function from the custom scheduler file. The default scheduling policy for Xen is implemented in the *Credit Scheduler* which is a fair-share scheduler and therefore not suitable for scheduling tasks with real-time constraints. We have extended Xen with our own scheduler that implements our deferrable server model. In the following, we shortly describe some of its implementation details.

The `do_schedule()` function is critical to the performance of the scheduler, as it is invoked very often. Therefore, we have designed it to be fast and simple. In order to achieve this, we keep our run queue sorted. The sorting process is conducted while inserting or reinserting vCPUs after instantiation or unblocking. As the run queue is already sorted with respect to vCPUs priorities, our `do_schedule()` function has to conduct only two operations. First, it updates the bookkeeping (consumed budget) for the currently running vCPU. Secondly, it chooses the next vCPU to run from the top of the run queue. In the case that there are no eligible vCPUs to run, the algorithm returns the idle vCPU. In our implementation, the scheduling quantum is set to 100 μ s, at which the server budget is updated. The pseudocode for our function is depicted in Algorithm 2.

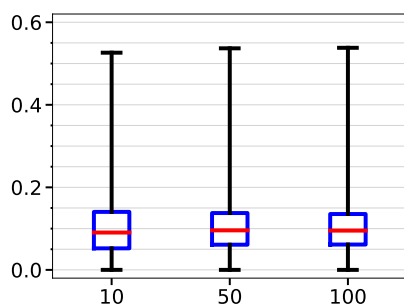
Another aspect worth mentioning is the implementation of the budget replenishment. One way to implement budget replenishment is inside the `do_schedule()` function. However, this would break the efficiency of this function, due to the unnecessarily high amount of budget validations. In most cases, the `do_schedule()` will be invoked more often than budget replenishment is necessary. Therefore, we transferred this functionality to timers. For each of the vCPUs a timer is instantiated with a period that equals the replenishment period of the task. The replenishment itself is happening inside the timer handler. We have extended our toolstack for the scheduler interface in the management domain with the possibility to migrate replenishment timers to other pCPUs in the system. This allows for testing and fine-tuning of the impact of timer interrupts on scheduling.

■ **Algorithm 2** Pseudocode of the `do_schedule()` function from our deferrable server scheduler.

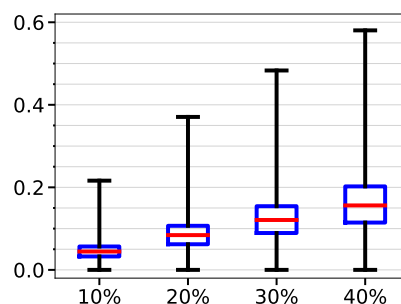
```

1: next = null
2: consumeBudget(getCurrentVCPU)
3: for each vCPU in RunQ do
4:   if vCPU.isRunnable then
5:     if vCPU.hasBudget then
6:       next ← vCPU
7:       break
8: if vCPU == null then
9:   next ← idleVCPU
10: return next

```



■ **Figure 6** $\frac{WCRT_{Ours}}{WCRT_{SOTA}}$ for different number n of servers per test.



■ **Figure 7** $\frac{WCRT_{Ours}}{WCRT_{SOTA}}$ for different total utilization U on one processor.

6 Evaluation and Discussion

Firstly we evaluate the tightness of our response time analysis in comparison to the state of the art [7]. Then, we conduct an empirical case study on a previously synthesized task set. With this case study we aim at showing that our approach of unikernel-based virtualization under deferrable servers is feasible in practice.

6.1 Numerical Simulation

We conducted numerical evaluation to compare our worst-case response time bound (Theorem 7) with the upper bound derived from [7] (Theorem 4). We present two experiments: 1) we distinguish different number of servers, and 2) we distinguish different total utilization of the servers, i.e., $U = \sum_{DS_i} \frac{Q_i}{P_i}$. To synthesize a system with n servers with a total utilization of U we applied 4 steps:

- We generated n utilization values $U_i \in (0, 1)$ that add up to the total utilization U by applying the UUniFast method presented in [4].
- We generated n replenishment periods P_i by drawing them log-uniformly from the interval $[1, 100]$ [ms] as it is suggested in [11] for the generation of task sets.
- We generated n servers DS_1, \dots, DS_n by specifying the replenishment periods P_i from above and by setting the capacity Q_i to $P_i \cdot U_i$. Priorities are set in rate-monotonic order.
- We generated the tasks τ_i by drawing the minimum inter-arrival time T_i uniformly at random from the interval $[1.0P_i, 1.5P_i]$ and by drawing the worst-case execution time C_i uniformly at random from the interval $[0.5Q_i, 1.0Q_i]$.

In the first experiment we generated 1000 sets of servers and their tasks at random using the procedure specified at the beginning of this section for each given number of servers in $\{10, 50, 100\}$. For each system in the first experiment, the utilization was drawn uniformly from the interval $[0.1, 0.4]$ since it has been shown in [5] or Theorem 8 in [20] that $U \leq \ln \frac{3}{2} \approx 0.40546$ guarantees that the service condition stated in Eq. (1) holds.

In the second experiment we generated 1000 sets of servers and their tasks at random using the same procedure specified previously for each given total utilization in $\{0.1, 0.2, 0.3, 0.4\}$. For each system in the second experiment, the number of servers n was drawn uniformly at random from the interval $[10, 100]$.

We applied our worst-case response time analysis ($WCRT_{Our}$) and the converted analysis, i.e., Theorem 4, derived from the real-time calculus based worst-case response time analysis in [7] ($WCRT_{SOTA}$). In Figures 6 and 7 we present the values of $\frac{WCRT_{Our}}{WCRT_{SOTA}}$ for all tasks in the tests as boxplots. In particular the lower the value, the better the performance of our analysis is. The medians are depicted by a horizontal line, the boxes mark a quartile of the data points, and the whiskers present all values.

We observe in Figure 6 that our bound is in median 90.5% smaller than the worst-case response time bound from Theorem 4. Although the number of servers per test does not make a difference in the performance of our method compared to the state of the art, in Figure 7 we can see that the utilization has an impact. The higher the utilization, the closer our bound gets to the bound of the state of the art. However, even with 40% utilization, our bound is in median still 84.4% smaller than the bound derived by the state of the art.

Please note that we limited our experiments for $U \leq \ln \frac{3}{2} \approx 0.40546$ so that the service condition of a deferrable server is always fulfilled. Otherwise, the service condition of a deferrable server cannot be always guaranteed and the focus of the evaluation would be drifted to the service condition tests of deferrable servers. However, our analysis in Theorems 4 and 7 is applicable as long as the deferrable server can fulfill its service condition.

6.2 Case Study

In this subsection, we describe our practical case study. We provide the experiment setup, the measurement methodology and present the results. Similar to the numerical simulation, we synthesize four periodic tasks with a total utilization 40%, and specify the capacity and replenishment period for each deferrable server. Their parameters are shown in Table 2. Note that we stick to periodic tasks to focus on the incurring overhead. They are deployed in form of four unikernels, which are instantiated on the same single processor core. Domain 0 receives one separate core.

The experiments were conducted on a commercial off-the-shelf hardware, a barebone Intel NUC Kit with an Intel i5-8259U 4-core processor running at a constant speed of 2.3 GHz. Turbo Mode, hyper-threading as well as all power management features were disabled. Domain 0 ran on a 64-Bit version of Ubuntu 20.04.1 LTS Server with a para-virtualized Linux kernel 5.4.0-59-generic. The used hypervisor was Xen in version 4.14.1 extended with our deferrable server scheduler and our toolstack.

6.2.1 Benchmark

For the purpose of this case study a User Datagram Protocol (UDP)-based client-server benchmark has been implemented in the programming language C. The client-server model fits the distributed nature of CPS's. The benchmark servers represent our CPS tasks (e.g. protection or control algorithms) deployed as unikernels. The computational workloads of

■ **Table 2** Four periodic tasks and their corresponding deferrable servers for our case study. Note that the time unit is [ms] and the WCRT of each task is derived from our analysis.

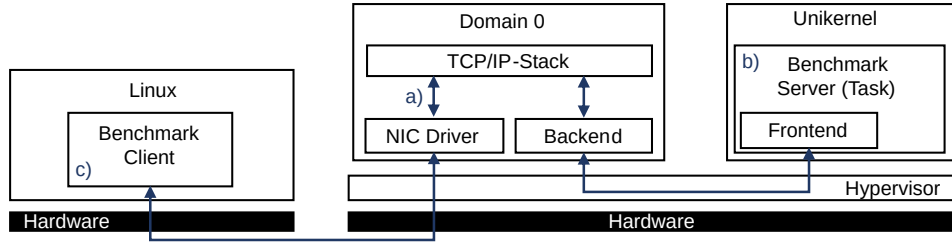
Periodic Task	Deferrable Server	WCRT
$T_1 = 12, C_1 = 1$	$P_1 = 10, Q_1 = 2$	1
$T_2 = 20, C_2 = 4$	$P_2 = 20, Q_2 = 4$	12
$T_3 = 60, C_3 = 8$	$P_3 = 50, Q_3 = 10$	26
$T_4 = 130, C_4 = 9$	$P_4 = 100, Q_4 = 10$	79

the tasks have been configured to fit the parameters of the analyzed task set as Table 2. The benchmark clients are used for triggering the computation inside the unikernels by generating requests (network packets). The requests can be interpreted as, for example, sensor values for a given task. After completing its computation, each task sends a response packet to the corresponding client. The responses can be seen as control commands destined for an actuator in a given CPS. The architecture of our benchmark is depicted in Figure 8.

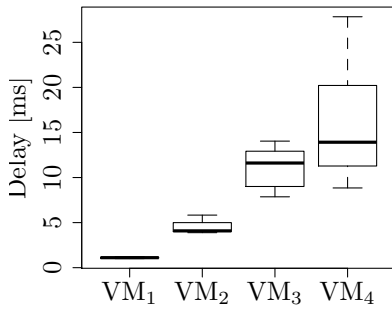
During benchmarking we collect three different types of latencies for every single request/response pair. The most important one is the task response time – a) in Figure 8. We define it as the time interval between the moment when a network packet (request) destined for a given unikernel (task) arrives at the bottom of the Linux TCP/IP network stack in Domain 0 and the time-stamp at which a corresponding departing packet is delegated to the network adapter for a response transmission. In other words, the packet arrival time-stamp and the corresponding packet departure time-stamp correlate with the task release and task finish times from our schedulability analysis. In order to collect these values, we hook into the TCP/IP stack layer-2 kernel functions using `systemtap`⁴ [27] and log the appropriate time-stamps. Another latency type is the task execution time – b) in Figure 8. It allows for quantifying the amount of processor cycles a task needed for completing its workload from the moment it has been scheduled to the point of time where it finishes its computation. We use it to estimate or verify the actual computational workload of our task on the given hardware. To this end, we implemented a clock cycle precise measurement technique as in [25]. This type is not suited for measuring the task response time, as it does not account for the time a given task may remain blocked after its release, e.g. due to its lower priority. On the client, we can also measure the round-trip times for the request/response pairs – c) in Figure 8. We use them to check the plausibility of the other latency types.

Last but not least, before we can present and discuss the results, we have to address the aspect of latencies related to networking. The schedulability analysis from Section 4 currently concentrates on local scheduling and does not account for latencies imposed by networking. However, the response times measured by the means of our benchmark include delays related to the entire software network stack of the host. Therefore, for the purpose of the following case study, we empirically estimated the latencies related to the network stack and adjusted our computed worst-case response time bounds with the worst-case network processing time. In the course of our experiments, the worst-case latency for the network stack did not exceed 250 μ s and amounted on average to 157 μ s.

⁴ <https://sourceware.org/systemtap>



■ **Figure 8** Benchmark architecture and the latencies measurement locations.



■ **Figure 9** Boxplots of the response time values for the different VMs.

■ **Table 3** The minimum value, arithmetic mean, standard deviation, worst-case response time and the worst-case response time bound for the different VMs.

VM	Min.	\bar{x}	σ	WCRT	WCRT(B)
VM ₁	1 ms	1.11 ms	0.047 ms	1.215 ms	1.25 ms
VM ₂	3.88 ms	4.42 ms	0.49 ms	5.83 ms	12.25 ms
VM ₃	7.86 ms	10.99 ms	2.02 ms	14.03 ms	26.25 ms
VM ₄	8.84 ms	15.54 ms	5.14 ms	27.83 ms	79.25 ms

6.2.2 Case Study Results

In the following, we present and discuss the results of our case study. In order to estimate the response times of the evaluated VMs, a total of twenty-two thousand measurements have been conducted. The results are depicted in form of four boxplots in Figure 9 and the corresponding numerical values are presented in Table 3. Please note that the estimated worst case delay, i.e., 250 μ s, as discussed above is added into each WCRT(B) of VMs.

As can be seen in Figure 9, our deferrable server services the four VMs accordingly to their priorities. Further, there are no outliers. This means that during the entire experiment none of the VMs experienced any spikes in their latencies. This in turn translates to a deterministic execution behavior. The standard deviation values from Table 3 support this evidence. In the case of the highest prioritized VM₁, σ amounts to 47 μ s – and this includes the variances caused by the network stack. For comparison, the σ computed from the task execution times collected inside VM₁ (see Figure 8, collection point b) amounts to 12 μ s.

Next, in the case study none of the empirically measured WCRTs exceeds the computed WCRTs bounds, i.e., $WCRT(B)$, from our analysis. This information can be gathered by comparing the WCRTs in Table 3. The comparison also shows that for decreasing VM priorities the computed bounds become more and more conservative.

Our case study leads to the conclusion that our approach is not only feasible but can be efficiently and safely realized in practice even on common-off-the-shelf hardware.

7 Related Work

Various virtualization techniques have been developed as a promising approach for embedded systems and latency-sensitive cloud computing [14]. The scheduling of VMs often implies a hierarchy of schedulers [9]. For hierarchical scheduling, server-based approaches are often considered to represent each virtual machine with its own real-time tasks as a single entity, which is scheduled by the hypervisor. Under fixed-priority scheduling, there are several server-based approaches, e.g., periodic server [29], sporadic server [32], deferrable servers [33]. The concept of deferrable servers was first introduced in [33]. It was originally designated to handle aperiodic activation in hard real-time systems. Afterwards, it has been utilized for resource budgets in [8, 28], where most of timing analyses assume each server may serve multiple tasks. As shown in [8], the lower priority tasks served by deferrable servers, however, may suffer from the back-to-back interference of higher priority tasks, resulting in a lower schedulability than the other periodic server approaches even under an exact schedulability test. Cuijpers and Bril in [7] showed that the response timing analysis proposed in [8] is no longer exact, in the absence of a low-priority task. When a deferrable server only serves a single task, the applicability of deferrable servers could be better. The result was based on real-time calculus [34], which considers a general model of task activation. In this work, we focus on sporadic tasks to derive a tighter worst-case response time analysis.

For using server-based approaches in virtualization, Shin and Lee in [31] developed the compositional scheduling framework (CSF) to compute the capacity of each server, given the replenishment period and the properties of tasks and their scheduler. RT-Xen was introduced in [35], adopting server-based approaches to schedule VMs. The CSF was introduced to RT-Xen for assuring the schedulability of tasks [17]. Both global and partitioned EDF schedulers were implemented for supporting multicore scheduling at the host-level [36]. In this work, we also adopt the deferrable server. However, as RT-Xen was implemented to comply with the compositional scheduling theory and addresses an architecture where scheduling decisions are divided between the hypervisor and the guest OS level, it is not suitable to realize our unikernel-based scheduling model. Therefore, we have extended the Xen hypervisor with our own scheduling infrastructure that implements our deferrable server model.

Lackorzynski et al. were the first to introduce the concept of flattening the hierarchical scheduling [16], where the task information is exported to the hypervisor scheduler. Drescher et al. further proposed to abandon the usage of server-based approaches to achieve ExVM [10]. However, as also noted in [10, 18], such flattening mechanisms might either break the temporal isolation between VMs, or expose task-specific information from a VM, which might not be appealing to the purpose of using virtualization for CPS's. Similarly, RTVirt [37] proposed to enable a cross-layer communication over the schedulers in two different tiers. Although this mechanism can adapt the scheduling decisions according to the dynamic changes, the potential concern is similar to the aforementioned flattening mechanisms. In this work, the main insight is to constrain the number of served real-time tasks of each deferrable server to a single one, without violating any important properties.

8 Conclusions

In this work, we proposed to leverage the scheduling architecture of unikernel-based virtualization to facilitate the schedulability of deferrable servers, in which each server only serves one sporadic task. We presented how to derive the worst-case response time analysis under practical scenarios. The evaluation results show that our analysis outperforms the

restated analysis based on the state-of-the-art [7]. In addition, we demonstrated that the unikernel-based architecture can be effectively implemented on top of the Xen hypervisor [2] and conducted a case study to evaluate the applicability of the proposed approach.

In future work we plan to investigate more scenarios over different relationships between a sporadic task and its deferrable server, e.g., scenarios with $T_k \leq P_k$. Further exploration may unleash the full power of the deferrable servers and eventually benefit more CPS applications under real-time virtualization.

References

- 1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020. doi:10.1109/RTSS49844.2020.00012.
- 2 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. doi:10.1145/1165389.945462.
- 3 Sanjoy Baruah and Nathan Fisher. Component-based design in multiprocessor real-time systems. In *2009 International Conference on Embedded Software and Systems*, pages 209–214, 2009. doi:10.1109/ICSS.2009.71.
- 4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.
- 5 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, pages 107–118, 2015. doi:10.1109/RTSS.2015.18.
- 6 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn B. Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil C. Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real Time Syst.*, 55(1):144–207, 2019. doi:10.1007/s11241-018-9316-9.
- 7 Pieter J. L. Cuijpers and Reinder J. Bril. Towards budgeting in real-time calculus: Deferrable servers. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS’07*, pages 98–113, Berlin, Heidelberg, 2007. Springer-Verlag.
- 8 Robert I. Davis and Alan Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS*, pages 389–398, 2005. doi:10.1109/RTSS.2005.25.
- 9 Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium (RTSS)*, pages 308–319, 1997. doi:10.1109/REAL.1997.641292.
- 10 Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. A flattened hierarchical scheduler for real-time virtualization. In *Proceedings of the 13th International Conference on Embedded Software, EMSOFT ’16*, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2968478.2968501.
- 11 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, pages 6–11, 2010.
- 12 D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995. doi:10.1145/224056.224076.
- 13 Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 38–51, 1997.
- 14 Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014. doi:10.1016/j.sysarc.2014.07.004.

- 15 Boguslaw Jablkowski and Olaf Spinczyk. Cps-xen: A virtual execution environment for cyber-physical applications. In Luís Miguel Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems - ARCS 2015 - 28th International Conference, Porto, Portugal, March 24-27, 2015, Proceedings*, volume 9017 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2015. doi:10.1007/978-3-319-16086-3_9.
- 16 Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2380356.2380376.
- 17 Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh T. X. Phan, Chris Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. Realizing compositional scheduling through virtualization. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 13–22, USA, 2012. IEEE Computer Society. doi:10.1109/RTAS.2012.20.
- 18 Haoran Li, Meng Xu, Chong Li, Chenyang Lu, Christopher Gill, Linh Phan, Insup Lee, and Oleg Sokolsky. Multi-mode virtualization for soft real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 117–128, 2018. doi:10.1109/RTAS.2018.00022.
- 19 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.
- 20 Cong Liu and Jian-Jia Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *2014 IEEE Real-Time Systems Symposium*, pages 173–183, 2014. doi:10.1109/RTSS.2014.10.
- 21 Anil Madhavapeddy, Thomas Leonard, Magnus Skjægstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 559–573, USA, 2015. USENIX Association.
- 22 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472, March 2013. doi:10.1145/2490301.2451167.
- 23 Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132763.
- 24 Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation*, pages 459–473, 2014.
- 25 Gabriele Paoloni. How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures, 2010.
- 26 Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304. ACM, 2011. doi:10.1145/1950365.1950399.
- 27 Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and J Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64, 2005.
- 28 Saowanee Saewong, Ragnathan Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierarchical fixed-priority scheduling. In *ECRTS*, pages 173–181, 2002. URL: <http://csdl.computer.org/comp/proceedings/ecrts/2002/1665/00/16650173abs.htm>.

- 29 Lui Sha, John P. Lehoczky, and Rangunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.
- 30 Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 181–190, 2008. doi:10.1109/ECRTS.2008.28.
- 31 Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, 2004. doi:10.1109/REAL.2004.15.
- 32 Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.
- 33 J.K. Strosnider, J.P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995. doi:10.1109/12.368008.
- 34 Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems, Emerging Technologies for the 21st Century*, pages 101–104, 2000. doi:10.1109/ISCAS.2000.858698.
- 35 Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 39–48, 2011.
- 36 Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the 14th International Conference on Embedded Software, EMSOFT '14*, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2656045.2656066.
- 37 Ming Zhao and Jorge Cabrera. Rtvirt: Enabling time-sensitive computing on virtualized systems through cross-layer cpu scheduling. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys '18*, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3190508.3190527.

A Appendix

► **Lemma 9.** *Under global preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers on m homogeneous (identical) physical processors, a sufficient service condition test can be written as Eq. (2).*

Proof. This can be proved by contrapositive. Suppose that the service condition to provide capacity Q_k for DS_k cannot be fulfilled during a period from a to $a + P_k$ for contrapositive. Under multiprocessor global preemptive fixed-priority scheduling, it implies that whenever DS_k is not served, the m processors are busy serving other higher-priority workload (i.e., servers in this case). The amount of execution time a higher-priority DS_i can be executed from a to $a + t$ is upper-bounded by $\left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i$, by considering the back-to-back hitting.

Therefore, we know that $\exists DS_k, \forall 0 < t \leq P_k, Q_k + \frac{\sum_{DS_i \in hp_k} \left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i}{m} > t$. By taking the negation of the above condition, we reach the conclusion. ◀

A Mathematical Comparison Between Response-Time Analysis and Real-Time Calculus for Fixed-Priority Preemptive Scheduling

Victor Pollex  

INCHRON AG, Erlangen, Germany

Frank Slomka  

Institute of Embedded Systems/Real-Time Systems, Faculty of Engineering and Computer Science, Universität Ulm, Germany

Abstract

Fixed-priority preemptive scheduling is a popular scheduling scheme for real-time systems. This is accompanied by a vast amount of research on how to analyse and check whether these systems satisfy their real-time requirements. Two methods that emerged from this research are the response-time analysis and the real-time calculus. These two methods have been compared empirically on the basis of several abstract systems showing that for some systems one method gives better results than the other and for other systems both methods appear to give the same results. However, empirical analyses inherently contain uncertainty. To get a definitive answer we compare both methods mathematically and we show that both methods give the same results for systems that use fixed-priority preemptive scheduling and independent tasks.

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

Keywords and phrases real-time systems, fixed-priority scheduling, response-time analysis, real-time calculus

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.7

Funding *Victor Pollex:* This work was supported by Federal Ministry of Education and Research under the grant agreement number 01IS21031A.

Frank Slomka: This work was supported by Federal Ministry of Education and Research under the grant agreement number 01IS21031B.

1 Introduction

For real-time systems it is necessary to verify that they meet their real-time requirements. Two of the methods that have emerged to verify these systems are the response-time analysis (RTA) and the real-time calculus (RTC). The response-time analysis originates from a proof [12, Theorem 5] that shows when a real-time system that periodically runs a set of independent tasks will always produce results on time. Whereas the origin of the real-time calculus is a mathematical framework [6, 7] to find a bound for the delay that a data stream is subjected to when flowing through a packet switched network.

Over time large amounts of work was produced regarding the response-time analysis and the real-time calculus that covers, among other things, different scheduling algorithms, different patterns on how tasks recur, and dependencies between tasks as well as empirical comparisons.

In distributed real-time systems where the activation of tasks can follow a complex pattern, the real-time calculus, due to its more expressive model, appears to produce the same or better results than the response-time analysis [14, Benchmark 1]. When the distributed real-



© Victor Pollex and Frank Slomka;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 7; pp. 7:1–7:25



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

time system is a feedback loop, then both methods appear to produce the same results [14, Benchmark 2]. However, when the distributed real-time system has a cyclic dependency or there are data dependencies, then the response-time analysis appears to produce the same or better results than the real-time calculus [14, Benchmark 3 and 4].

All of these comparisons are empirical. Therefore, we only have an indication that one method might always produce the same or better results than the other. However, we do not know for certain. So, to get a definitive answer we compare these methods formally. By means of a mathematical proof we show that both methods produce the same results when the real-time system uses fixed-priority preemptive scheduling and their tasks are independent.

Structure of the Paper

The remainder of this paper has the following structure: First we describe the related work in Section 2. Then we introduce the models and the analyses of the response-time analyses and the real-time calculus in Section 3. Subsequently, in Section 4 we describe the assumptions that we use and formally compare the response-time analysis with the real-time calculus by means of a mathematical proof. We conclude the paper with a summary in Section 5.

2 Related Work

In their seminal paper [12] Liu and Layland introduce a sufficient test for real-time systems that run a set of independent tasks which recur periodically, have an implicit deadline, and are subject to the fixed-priority preemptive scheduling algorithm. For the same type of real-time systems, Joseph and Pandya improve the analysis in [9] by supplying an exact test. Moreover, their test is not only suitable for tasks with implicit deadlines, but also for tasks with restricted deadlines. Lehoczky presents in [11] a further improvement to the previous test by extending the exact test to also include tasks with arbitrary deadlines, which then Tindell et al. further extend in [25] to allow tasks to have a release jitter. Similarly, Audsley et al. improve in [1] the exact test where they assume that the set of tasks have implicit deadlines, but they allow the tasks to block internally and have a release jitter. In [24] Tindell and Clark provides a test that combines all of these improvements. Audsley et al. present in [2] an historic perspective on fixed-priority preemptive scheduling. In [16] Richter presents an abstract representation of the bounds on how often the tasks recur. Based on which, Schliecker et al. introduce in [18] the multiple event busy time as a generalization of the concept of busy period which many response-time analyses use.

The work of Cruz [6, 7] is considered seminal for network calculus which is a mathematical framework to find bounds for the latency that network components cause on bit streams [10]. Fidler presents in [8] a comprehensive survey of the models that the network calculus uses. Based on the network calculus, Thiele et al. introduce in [23] the models for the real-time calculus, how to get these models from a recurring real-time task, and they describe a schedulability test with these models for systems that use a fixed-priority preemptive scheduling algorithm. Chakraborty et al. refine in [5] the real-time calculus to calculate tighter bounds and apply it to scheduling networks. Wandeler improves the real-time calculus further in [26].

In [14] Perathoner et al. use several small abstract systems to empirically benchmark various formal performance analyses with these systems. Among the analyses are the response-time analysis and the real-time calculus. They show that neither of these two analyses always

outperforms the other. But rather it depends on the characteristics of the system under analysis whether one outperforms the other. However, this is an empirical comparison. We compare them mathematically instead.

Naedele et al. present in [13] a schedulability test with the real-time calculus for a system that uses a fixed-priority preemptive scheduling algorithm. They indicate that it is possible to derive the test in [25] from their schedulability test. Similarly, Pollex et al. show in [15] a generalization of the response-time analysis with the help of the real-time calculus. They exemplarily derive the analysis in [25] from the real-time calculus. However, we use the more general schedulability test from [17] for our comparison.

In [17] Schliecker presents the multiple event busy time, how to derive it for fixed-priority preemptive scheduling, and an accompanying analysis as an extension of the work in [25]. They also show how to derive a multiple event busy time from the service curves of the real-time calculus. However, there is no discussion how the multiple event busy times, the one extended from [25] and the other derived from the service curve of the real-time calculus, relate to each other. We show that they are in fact identical.

Boyer and Roux propose in [3, 4] a model which can embed the models that the network calculus and the response-time analysis use, therefore making it possible to analyse a system that uses both models. However, they only look into how to interface between the different models and not how the individual analyses compare. Furthermore, much of the mathematical background that they use assumes real-valued functions that have the extended non-negative real numbers as domain and co-domain. We generalise some of them, where we assume mappings that use partially ordered sets or lattices as domain and co-domain.

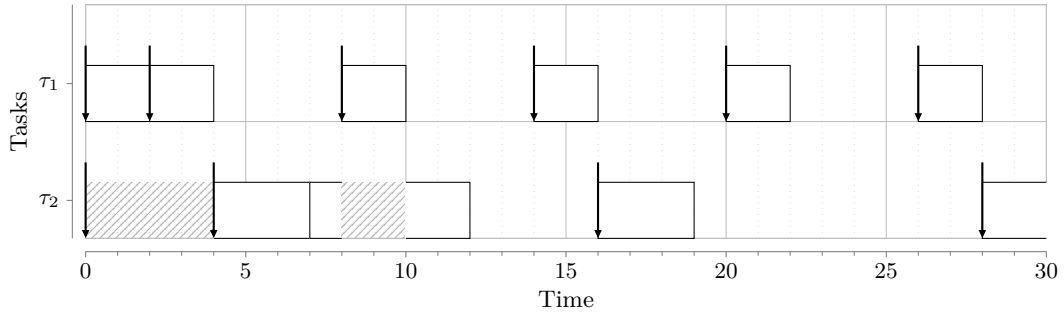
Depending whether the real-time systems use fixed or dynamic priority scheduling, the existing analyses differ because of the different mathematical requirements on the models. In [19, 20] Slomka and Sadeghi introduce a new mathematical framework based on mathematical tools from electric engineering to analyse real-time systems. This new mathematical framework makes it possible to describe an unified analysis for real-time systems that use fixed and/or dynamic priority scheduling. They also describe existing analyses like [25] and analyses for real-time systems with dynamic priority scheduling with the new mathematical framework. Based on this mathematical framework, Slomka and Sadeghi show in [21] preliminary work for investigating the relationship between the response-time analysis and the real-time calculus. They sketch possible similarities, however they do not express the real-time calculus with their new mathematical framework, let alone compare them.

3 Models and Analyses

To analyse a real-time system we need to appropriately model it. Since we compare the response-time analyses with the real-time calculus, we first describe the assumptions and notations that both analyses have in common followed by a running example that we use to illustrate the concepts of both analyses. Second, we present the common mathematical concepts that both analyses use. Third, we restate the notation and the analyses themselves as presented in [17] and [26] for the response-time analysis and the real-time calculus, respectively.

3.1 Common Assumptions and Notation

We assume that the real-time system has a single core processor that uses a fixed-priority scheduler which allows a higher priority task to preempt a lower priority task at any time. The set of tasks Γ that is assigned to the processor has n tasks τ_i , where $i \in \{1, \dots, n\}$. We



■ **Figure 1** Worst-case schedule of the system described in Example 1.

only consider events that cause the system to release a job of a task which the system then puts into the ready queue of the scheduler. The scheduler is work-conserving, i.e. whenever a job of a task is in the ready queue, the scheduler assigns a job to the processor to execute it. Each task τ_i has a unique priority which defines a strict order on the set of tasks Γ . We use the index of τ_i to also represent the priority of the task. A lower numerical value of the index means that the task has a higher priority, i.e. task τ_3 has a higher priority than task τ_8 . The tasks are independent from each other. There are no data dependencies, temporal dependencies, or any other dependencies between them. Also, the jobs of the tasks do not use any shared resources other than the processor.

To illustrate the concepts of both analyses we use the following running example of a real-time system.

► **Example 1.** Let the system consist of two tasks $\Gamma := \{\tau_1, \tau_2\}$, where τ_1 has a higher priority than τ_2 . Task τ_1 releases a job every $p_1 := 6$ time units, has a release jitter of $j_1 := 4$ time units, and the processor needs $c_1^+ := 2$ time units to process each of its jobs. Similarly, task τ_2 releases a job every $p_2 := 12$ time units, has a release jitter of $j_2 := 8$ time units, and the processor needs $c_2^+ := 3$ time units to process each of its jobs.

Figure 1 shows the worst-case schedule for this system. Task τ_1 releases a job at time points 0, 2, and from then on every p_1 time units. Similarly, task τ_2 releases a job at time points 0, 4, and from then on every p_2 time units. The processor completes the first job of task τ_2 at 7 time units and the second job at 12 time units. So, the length of the time interval for the first two jobs from their release to when the processor completes them is $7 - 0 = 7$ and $12 - 4 = 8$ time units, respectively. Because at 12 time units an interval starts where no jobs are pending and therefore the processor is idle, we can conclude that a job of task τ_2 will never need more than 8 time units from the time it was released until the processor completes it.

3.2 Common Mathematical Notation and Definitions

First we introduce common mathematical notation and definitions. Then we present three mathematical definitions which are fundamental to many lemmas on which the theorem of our main contribution bases.

The set of positive integers and non-negative integers is \mathbb{N} and \mathbb{N}_0 , respectively. Furthermore, the set of real numbers, the extended real numbers (includes $-\infty$ and ∞), and the non-negative real numbers is \mathbb{R} , $\overline{\mathbb{R}}$, and \mathbb{R}_0^+ , respectively.

► **Definition 2** (Monotonicity). *Let $f: X \rightarrow Y$ be a mapping from a partially ordered set X to a partially ordered set Y , then f is isotone or antitone if*

$$\forall x_1, x_2 \in X : x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2) \text{ or} \quad (1a)$$

$$\forall x_1, x_2 \in X : x_1 \leq x_2 \Rightarrow f(x_1) \geq f(x_2), \text{ respectively.} \quad (1b)$$

Isotone mappings are also called *order-preserving* or in case of functions *increasing* or *non-decreasing*. Similarly, antitone mappings are also called *order-reversing*, *decreasing*, or *non-increasing*.

► **Definition 3** (Directional Continuity). *Let $f: X \rightarrow \mathbb{R}$ be a function from a subset X of the real numbers, then f is continuous on the left or right at $x \in X$ if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall \xi \in X \cap (x - \delta, x) : |f(\xi) - f(x)| < \epsilon \text{ or} \quad (2a)$$

$$\forall \epsilon > 0 \exists \delta > 0 \forall \xi \in X \cap (x, x + \delta) : |f(\xi) - f(x)| < \epsilon, \text{ respectively.} \quad (2b)$$

If f is continuous on the left or right at every element of X , then f is called continuous on the left or right, respectively. Alternatively, f can be called left-continuous or right-continuous.

The models use several functions like the event load function (Definition 8) or the arrival curves (Definition 11) which in general are increasing, but not strictly increasing. Therefore, their inverse functions do not necessarily exist, but their closely related pseudo-inverse do. We define the pseudo-inverse of a function with the help of its contour set.

► **Definition 4** (Contour Set). *Let $f: X \rightarrow Y$ be a mapping from a set X to a partially ordered set Y , then the lower contour set $X_{f \leq y}$ and the upper contour set $X_{y \leq f}$ of f at $y \in Y$ are*

$$X_{f \leq y} := \{x \in X : f(x) \leq y\} \text{ and} \quad (3a)$$

$$X_{y \leq f} := \{x \in X : y \leq f(x)\}. \quad (3b)$$

► **Definition 5** (Pseudo-Inverse). *Let $f: X \rightarrow Y$ be a mapping from a subset X of a complete lattice L to a partially ordered set Y , then the pseudo-inverse $f^{\leq -1}: Y \rightarrow L$ and $f^{\geq -1}: Y \rightarrow L$ at $y \in Y$ are*

$$f^{\leq -1}(y) := \inf X_{y \leq f} \text{ and} \quad (4a)$$

$$f^{\geq -1}(y) := \sup X_{f \leq y} \quad (4b)$$

with the convention that $\inf \emptyset = \sup X$ and $\sup \emptyset = \inf X$.

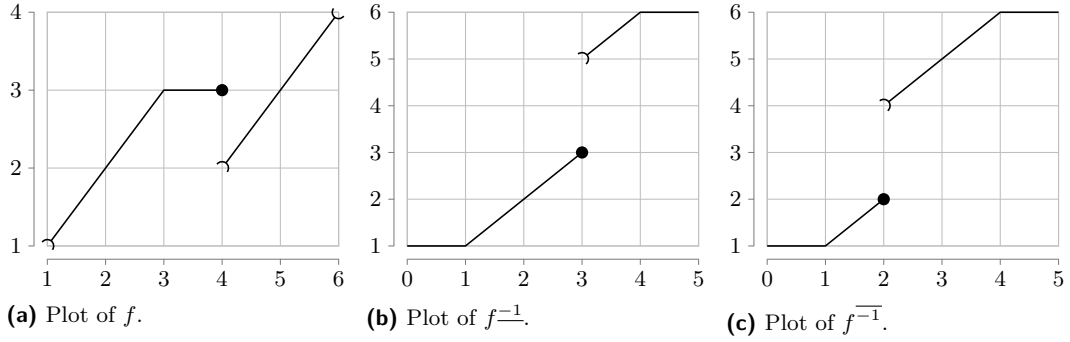
Note that we do not require that f is increasing as in [4, Definition 5]. This makes the new result in Lemma 50 possible. Also, note that the image of the pseudo-inverses $f^{\leq -1}$ and $f^{\geq -1}$ is a subset of the complete lattice L and not a subset of X . The reason is that the pseudo-inverses are the infimum and supremum of subsets of X . These do not necessarily have to be in X , but they are in L . To illustrate this, we use the following example.

► **Example 6.** Let $f: I \rightarrow \mathbb{R}$ be a function from the open interval $I := (1, 6)$ to the real numbers where $f(x) = x$ when $x \in (1, 3]$, $f(x) = 3$ when $x \in (3, 4]$, and $f(x) = x - 2$ when $x \in (4, 6)$. See Figure 2a for a plot of f . Table 1 shows the contour sets and their respective pseudo-inverses for any $y \in \mathbb{R}$. Lastly, Figures 2b and 2c show the plots for the pseudo-inverses $f^{\leq -1}$ and $f^{\geq -1}$. As Table 1 shows, the pseudo-inverses attain the values 1 and 6, which are not in I .

7:6 Response-Time Analysis vs. Real-Time Calculus

■ **Table 1** Resulting contour sets, $I_{y \leq f}$ and $I_{f \leq y}$, and their respective pseudo-inverses, $f^{-1}(y)$ and $f^{\overline{-1}}(y)$, for any $y \in \mathbb{R}$ for function f defined in Example 6.

$y \in$	$I_{y \leq f}$	$f^{-1}(y)$	$y \in$	$I_{f \leq y}$	$f^{\overline{-1}}(y)$
$(-\infty, 1]$	I	1	$(-\infty, 1]$	\emptyset	1
$(1, 2]$	$[y, 6)$	y	$(1, 2]$	$(1, y]$	y
$(2, 3]$	$[y, 4] \cup [y + 2, 6)$	y	$(2, 3]$	$(1, y] \cup (4, y + 2]$	$y + 2$
$(3, 4]$	$[y + 2, 6)$	$y + 2$	$[3, 4]$	$(1, y + 2]$	$y + 2$
$[4, \infty)$	\emptyset	6	$[4, \infty)$	I	6



■ **Figure 2** Plot of the functions f defined in Example 6 and its pseudo-inverses f^{-1} and $f^{\overline{-1}}$.

► **Definition 7** (Deconvolution). Let $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ and $g: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions. The deconvolution in inf-plus \odot and in sup-plus $\overline{\odot}$ are defined as follows [10, Definition 3.1.13 and 3.2.2]:

$$(f \odot g)(x) := \sup_{0 \leq \xi} \{f(x + \xi) - g(\xi)\} \quad (5a)$$

$$(f \overline{\odot} g)(x) := \inf_{0 \leq \xi} \{f(x + \xi) - g(\xi)\} \quad (5b)$$

3.3 Response-Time Analysis

Over time, events occur that release jobs of tasks. To capture the density of these events for a task τ_i , the response-time analysis uses two functions, the event load function η_i^+ and the event distance function δ_i^- .

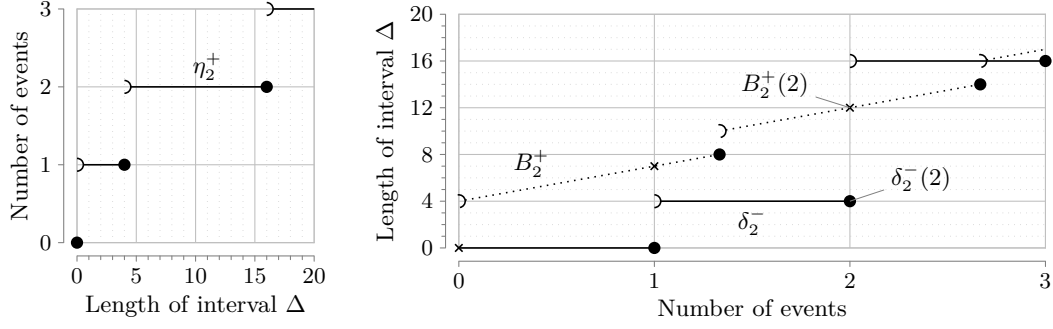
► **Definition 8** (Event Load Function). Confer [17, p. 53, (3.3)]. The upper event load function for task τ_i maps a length of a time interval to an upper bound of the number of events that can occur in any time interval of that length and is denoted by

$$\eta_i^+ : \mathbb{R}_0^+ \rightarrow \mathbb{N}_0. \quad (6)$$

► **Definition 9** (Event Distance Function). Confer [17, p. 53, (3.1)]. The minimum event distance function for task τ_i maps a number of events to a lower bound of the length of a time interval in which at least that amount of events occur and is denoted by

$$\delta_i^- : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+. \quad (7)$$

Given q events, every interval in which at least q events occur has a length of at least $\delta_i^-(q)$. Or, in any interval with a length smaller than $\delta_i^-(q)$ less than q events occur.



(a) The event load function η_2^+ of task τ_2 . (b) The event distance function δ_2^- and the multiple event busy time function B_2^+ of task τ_2 .

■ **Figure 3** The concepts of the response-time analysis applied on the system described in Example 1.

Both the event load function and the event distance function are closely related, such that we can derive one function from the other. The relationship between these functions is that one function is essentially the pseudo-inverse of the other. Commonly, we have the upper event load function and from that we derive the minimum event distance function with (cf. [17, p. 54, (3.7)])

$$\delta_i^- := \eta_i^{+ -1}. \quad (8)$$

Additionally, a task has a worst-case execution-time c_i^+ which describes the maximum amount of processor time without any interference of higher priority task that a job needs for the processor to execute it.

The worst-case response time r_i^+ of task τ_i is bounded by (cf. [17, p. 64, (3.22)])

$$r_i^+ \leq \max_{q \in \mathbb{N}_0} \{B_i^+(q) - \delta_i^-(q)\}. \quad (9)$$

Equation (9) uses the multiple event busy time function $B_i^+ : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$, cf. [17, p. 63, Definition 3.6]. For fixed priority preemptive scheduling the multiple event busy time function is (cf. [17, p. 64, (3.23)])

$$B_i^+(q) = \min_{\Delta \in \mathbb{R}_0^+} \left\{ \Delta : \Delta = q \cdot c_i^+ + \sum_{j=1}^{i-1} (\eta_j^+(\Delta) \cdot c_j^+) \right\}. \quad (10)$$

In (10) we explicitly specify the smallest fix-point to resolve any possible mathematical ambiguity, because that is how [17, p. 64, (3.23)] is intended.

► **Example 10.** Given the system in Example 1 we exemplarily derive the various functions of the response-time analysis for it.

For task τ_1 the worst-case execution time is the same as given in the example system, i.e. $c_1^+ = 2$. Because task τ_1 has the highest priority, the multiple event busy time function is $B_1^+(q) = q \cdot c_1^+$ according to Equation (10). With the release of jobs every $p_1 = 6$ time units and a release jitter of $j_1 = 4$ time units, the event load function is $\eta_1^+(\Delta) = \left\lceil \frac{\Delta + j_1}{p_1} \right\rceil$ for $\Delta > 0$ and $\eta_1^+(\Delta) := 0$ for $\Delta = 0$. Now that we have the event load function we derive the event distance function according to Equation (8), i.e. $\delta_1^-(q) = \max\{0, \lceil q - 1 \rceil \cdot p_1 - j_1\}$.

Similarly, for task τ_2 the worst-case execution time is the same as in the example system, i.e. $c_2^+ = 3$. According to Equation (8) the multiple event busy time function B_2^+ has to consider the interference from the higher priority task τ_1 which results in $B_2^+(q) = q \cdot c_2^+ + \left\lceil \frac{q \cdot c_2^+ + 8}{4} - 1 \right\rceil \cdot c_1^+$ for $q > 0$ and $B_2^+(q) = 0$ for $q = 0$. Task τ_2 releases jobs every $p_2 = 12$ time units and has a release jitter of $j_2 = 8$ time units, therefore its event load functions is $\eta_2^+(\Delta) = \left\lceil \frac{\Delta + j_2}{p_2} \right\rceil$ for $\Delta > 0$ and $\eta_2^+(\Delta) = 0$ for $\Delta = 0$. Deriving the event distance function from the event load function according to Equation (8) results in $\delta_2^-(q) = \max\{0, \lceil q - 1 \rceil \cdot p_2 - j_2\}$.

Now that we have both the multiple event busy time function B_2^+ and the event distance function η_2^+ for task τ_2 we can calculate the upper bound for the worst-case response-time, see Figure 3b for a plot with both of these functions. The worst-case response-time for task τ_2 is not greater than $r_2^+ \leq \max_{q \in \mathbb{N}_0} \{B_2^+(q) - \delta_2^-(q)\} = \max\{0 - 0, 7 - 0, 12 - 4, \dots\} = 8$ time units. Note that we do not necessarily need to compute all the values of the multiple event busy time function B_2^+ . In a schedulable system there will be a point when there are no pending jobs of task τ_2 that could defer the execution of any of its following jobs, cf. [17, p. 72, Theorem 3.9].

3.4 Real-Time Calculus

The real-time calculus models a task with the Greedy Processing Component which has event-based arrival curves $\bar{\alpha}_i$ and resource-based service curves β_i as input.

► **Definition 11** (Event-Based Arrival Curves). Confer [26, p. 16, Def. 1] and [26, p. 73, Def. 3]. Let $R[s, t)$ denote the number of events that occur in the interval $[s, t)$, where $s \in \mathbb{R}_0^+$ is a point in time before $t \in \mathbb{R}_0^+$, i.e. $s \leq t$. Then, the event-based lower arrival curve $\bar{\alpha}^- : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ and the event-based upper arrival curve $\bar{\alpha}^+ : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ satisfy for every point in time $t \in \mathbb{R}$ and every length of interval $\Delta \in \mathbb{R}_0^+$ the property:

$$\bar{\alpha}^-(\Delta) \leq R[t, t + \Delta) \leq \bar{\alpha}^+(\Delta) \quad (11)$$

► **Definition 12** (Resource-Based Service Curves). Confer [26, p. 19, Def. 2] and [26, p. 73, Def. 6]. Let $C[s, t)$ denote the amount of resources that are available in the interval $[s, t)$, where $s \in \mathbb{R}_0^+$ is a point in time before $t \in \mathbb{R}_0^+$, i.e. $s \leq t$. Then, the resource-based lower service curve $\beta^- : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ and the resource-based upper service curve $\beta^+ : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ satisfy for every point in time $t \in \mathbb{R}$ and every length of interval $\Delta \in \mathbb{R}_0^+$ the property:

$$\beta^-(\Delta) \leq C[t, t + \Delta) \leq \beta^+(\Delta) \quad (12)$$

The arrival curves $\bar{\alpha}^-$ and $\bar{\alpha}^+$ map a length of a time interval to a lower, respectively upper, bound of the amount of events that can occur in any interval of that length which causes the system to release jobs. Similarly, the service curves β^- and β^+ map a length of a time interval to a lower, respectively upper, bound of the amount of resources that are available in any interval of that length to execute any pending jobs. However, the arrival curves in Definition 11 are event-based, which map from a length of an interval to a number of events. But, the service curves in Definition 12 are resource-based, which map from a length of an interval to an amount of resources. Number of events and amount of resources are not comparable. So, we need to transform at least one of them into the other. Which we do with workload curves.

► **Definition 13** (Workload Curves). Confer [26, p. 74, Def. 7]. Let $W(u)$ denote the amount of resources that are necessary to process u consecutive events, then the lower workload curve $\gamma^- : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ and the upper workload curve $\gamma^+ : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ satisfy for $u \in \mathbb{R}_0^+$ and $v \in \mathbb{R}_0^+$ consecutive events, where $u \leq v$, the property:

$$\gamma^-(v - u) \leq W(v) - W(u) \leq \gamma^+(v - u) \quad (13)$$

To transform the resource-based lower service curve into its event-based form, we compose (◦) it with the pseudo-inverse of the upper workload curve. Confer [26, p. 74, (4.6)] and [26, p. 75, (4.11)].

$$\bar{\beta}^- = \gamma^{+\overline{-1}} \circ \beta^- \quad (14)$$

Similarly, we transform the event-based upper arrival curve into its resource-based form by composing it with the upper workload curve. Confer [26, p. 75, (4.8)].

$$\alpha^+ = \gamma^+ \circ \bar{\alpha}^+ \quad (15)$$

With the arrival curve and the service curve in either the resource-based form or the event-based form, we can describe an upper bound on the delay of task τ_i . This is the longest time that the processor needs to execute a job of task τ_i . From the time the event occurred which released the job until the job was completely executed. We describe this upper bound with the notion of the largest horizontal distance between functions.

► **Definition 14** (Largest horizontal distance between functions). Confer [10, p. 154, (3.21)]. Let $f, g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions, then the largest horizontal distance \leftrightarrow between f and g is

$$f \leftrightarrow g := \sup_{\lambda \in \mathbb{R}_0^+} \left\{ \inf_{\mu \in \mathbb{R}_0^+} \{ \mu : f(\lambda) \leq g(\lambda + \mu) \} \right\} \quad (16)$$

With the event-based upper arrival curve $\bar{\alpha}_i^+$ and lower service curve $\bar{\beta}_i^-$ of task τ_i , we can express the upper bound of the delay of task τ_i by means of the largest horizontal distance between functions and is (cf. [26, p. 26, (2.11)])

$$\bar{\alpha}_i^+ \leftrightarrow \bar{\beta}_i^- \quad (17)$$

However, we need the event-based lower service curve $\bar{\beta}_i^-$ for task τ_i . Given a task τ_i with its corresponding arrival and service curves, then the remaining lower service curve is (cf. [26, p. 23, (2.10) or p. 201, (A.15)])

$$\beta_j^-(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{ \beta_i^-(\lambda) - \alpha_i^+(\lambda) \} \text{ or } \beta_j^- = (\beta_i^- - \alpha_i^+)^{\nearrow} \quad (18)$$

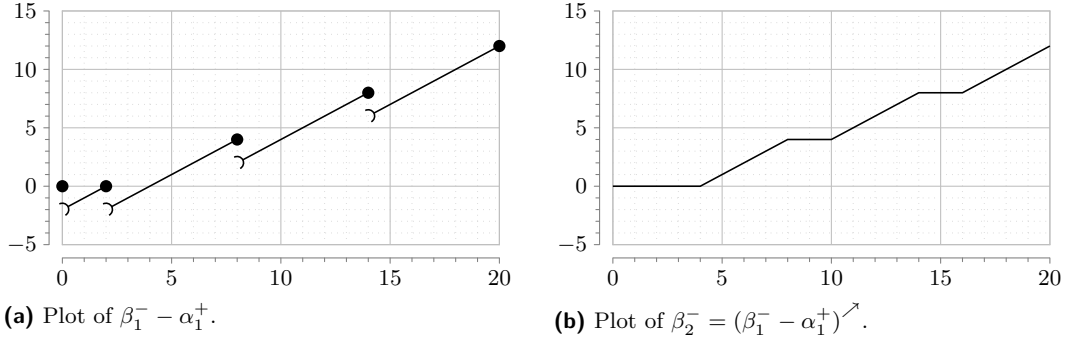
In the latter part of Equation (18) we apply an increasing operator, which is defined as

► **Definition 15** (Increasing operator). Let \mathbb{F} be the set of functions $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}$ and I_x the interval $[0, x]$ for a non-negative real number x , then the increasing operator $\nearrow : \mathbb{F} \rightarrow \mathbb{F}$ is

$$f^{\nearrow}(x) := \sup_{\xi \in I_x} \{ f(\xi) \} \quad (19)$$

The increasing operator transforms a function f into an increasing function and it is a closure operator.

7:10 Response-Time Analysis vs. Real-Time Calculus



■ **Figure 4** The resource-based lower service curve β_2^- of task τ_2 for the system described in Examples 1 and 19 according to Equation (18) by applying the increasing operator.

► **Remark 16.** Let \mathbb{F} be the set of functions $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}$, then

$$f^{\nearrow} \text{ is increasing} \quad (20)$$

Proof. Let I_x the interval $[0, x]$ for a non-negative real number x , $x_1, x_2 \in \mathbb{R}_0^+$ with $x_1 \leq x_2$, then $I_{x_1} \subseteq I_{x_2}$, therefore $f^{\nearrow}(x_1) = \sup_{\xi \in I_{x_1}} \{f(\xi)\} \leq \sup_{\xi \in I_{x_2}} \{f(\xi)\} = f^{\nearrow}(x_2)$. ◀

► **Remark 17 (Increasing closure).** Let \mathbb{F} be the set of functions $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}$, then the increasing operator $\nearrow: \mathbb{F} \rightarrow \mathbb{F}$ is a closure operator on the partially ordered set (\mathbb{F}, \leq) , where \leq is the pointwise order on functions.

Proof. Let $f, g \in \mathbb{F}$ be functions and I_x the interval $[0, x]$ for a non-negative real number x , then

- (a) $f \leq f^{\nearrow}$: Because x is an element in I_x it follows that $f(x) \leq \sup_{\xi \in I_x} \{f(\xi)\} = f^{\nearrow}(x)$.
- (b) $f \leq g \Rightarrow f^{\nearrow} \leq g^{\nearrow}$: Follows directly from Equation (19).
- (c) $f^{\nearrow \nearrow} = f^{\nearrow}$: Follows from Equations (19) and (20), i.e. $f^{\nearrow \nearrow}(x) = \sup_{\xi \in I_x} \{f^{\nearrow}(\xi)\} = f^{\nearrow}(x)$.

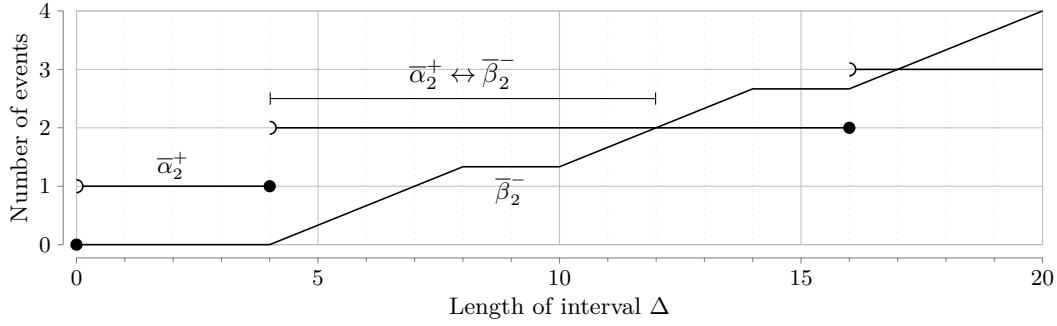
Because the increasing operator \nearrow satisfies (a)–(c), it is a closure operator. ◀

For the case of fixed-priority preemptive scheduling the lower available service curve β_i^- is the lower remaining service curve of the next higher priority task τ_{i-1} . See Figure 4 that shows the available service curve β_2^- of task τ_2 or the lower remaining service curve of task τ_1 for the system described in Examples 1 and 19. When $i = 1$ then the lower available service curve of the highest priority task τ_1 is equal to the lower available service curve to the entire scheduler itself, i.e. $\beta_1^- = \beta^-$.

Similarly to the horizontal distance between functions (Definition 14) we define the vertical distance between functions.

► **Definition 18 (Largest vertical distance of functions).** Confer [10, p. 154, (3.20)]. Let $f, g: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions, then the largest vertical distance \Downarrow between f and g is

$$f \Downarrow g := \sup_{\xi \in \mathbb{R}_0^+} \{f(\xi) - g(\xi)\} = \sup(f - g) \quad (21)$$



■ **Figure 5** The event-based upper arrival curve $\bar{\alpha}_2^+$, the event-based lower service curve $\bar{\beta}_2^-$, and the upper bound of the delay $\bar{\alpha}_2^+ \leftrightarrow \bar{\beta}_2^-$ for τ_2 of the system described in Example 1.

► **Example 19.** Given the system in Example 1 we exemplarily derive the various functions of the real-time calculus for it.

For task τ_1 be event-based upper arrival curve is $\bar{\alpha}_1^+(\Delta) = \left\lceil \frac{\Delta + j_1}{p_1} \right\rceil$ for $\Delta > 0$ and $\bar{\alpha}_1^+(\Delta) = 0$ for $\Delta = 0$. Because task τ_1 has the highest priority, the resource-based lower service curve is $\beta_1^-(\Delta) = \Delta$. Lastly, the upper workload curve is $\gamma_1^+(q) = q \cdot c_1^+$. Similarly, for task τ_2 the event-based upper arrival curve is $\bar{\alpha}_2^+(\Delta) = \left\lceil \frac{\Delta + j_2}{p_2} \right\rceil$ for $\Delta > 0$ and $\bar{\alpha}_2^+(\Delta) = 0$ for $\Delta = 0$. According to Equation (18) the resource-based lower service curve is $\beta_2^-(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{\beta_1^-(\lambda) - \alpha_1^+(\lambda)\}$, see Figure 4b for a plot of β_2^- , and the upper workload curve is $\gamma_2^+(q) = q \cdot c_2^+$.

Figure 5 shows the event-based upper arrival curve $\bar{\alpha}_2^+$ and the event-based lower service curve $\bar{\beta}_2^-$ of task τ_2 . From those we derive the upper bound of the delay $\bar{\alpha}_2^+ \leftrightarrow \bar{\beta}_2^-$ which is $12 - 4 = 8$ time units.

4 Formal Comparison of the RTA with the RTC

In this section we formally compare the upper bound for the worst-case response-time, Equation (9), that the response-time analysis uses with the upper bound for the delay, Equation (17), that the real-time calculus uses. For a fair comparison we must ensure identical initial conditions, therefore we make the following assumptions:

► **Assumption 20.** For mathematical reasons, every curve ($\bar{\alpha}_i^+$, β^- , and γ^+) is increasing and not bounded above, lower curves (β^-) are right-continuous, and upper curves ($\bar{\alpha}_i^+$ and γ^+) are left-continuous.

For the event-based upper arrival curve $\bar{\alpha}_i^+$ this means that an interval of greater length exists where at least as many events occur than in any interval of the same or smaller length and if the system keeps running for all eternity, an infinite amount of events will occur.

► **Assumption 21.** Jobs of tasks do not starve, every job finishes after a finite amount of time. For a set of n tasks, we express this by assuming that the available service for the lowest priority task τ_n is not bounded above, i.e. for all $r \in \mathbb{R}_0^+$ a $\Delta \in \mathbb{R}_0^+$ exists such that $r < \beta_n^-(\Delta)$.

This implies that the resource-based lower service curve β_i^- for every task $i \in \{1, \dots, n\}$ is not bounded above.

► **Assumption 22.** The event load function and the event-based upper arrival curve for a task τ_i are equal, i.e. $\eta_i^+ = \bar{\alpha}_i^+$, because both model exactly the same.

7:12 Response-Time Analysis vs. Real-Time Calculus

Examples for how to define the event load function for common event models are in [16, p. 50] or in [26, p. 16, Ex. 1]. All those definitions satisfy Assumption 20.

► **Assumption 23.** An implicit assumption of the response-time analysis is that one unit of processor time is available per time unit. Therefore, the lower bound of the available resources that a fixed-priority preemptive scheduler has is $\beta^-(\Delta) = \Delta$, cf. [26, p. 20, Ex. 2].

The resource-based lower service curve $\beta^-(\Delta) = \Delta$ is not bounded above and is continuous. Therefore, it is also right-continuous and thus satisfies Assumption 20.

► **Assumption 24.** Furthermore, the response-time analysis assumes that every job of a task τ_i needs at most c_i^+ processor time to execute. So, we have for the upper workload curve $\gamma_i^+(q) = q \cdot c_i^+$.

This also satisfies Assumption 20.

Under Assumptions 20–24 we provide our main contribution, Theorem 31, a proof that the upper bound for the worst-case response-time of the response-time analysis, Equation (9), and the upper bound for the delay of the real-time calculus, Equation (17), are equal for every task in a set of independent tasks as described in Section 3.1.

We divide the proof of Theorem 31 into several steps and begin by revisiting Figures 3b and 5. These show the functions that the response-time analysis and the real-time calculus use to model the system in Example 1. Both analyses calculate the same upper bound for the worst-case response-time or delay. It appears that the functions in Figure 3b are the pseudo-inverse of the functions in Figure 5. So, there seem to exist a relation between the horizontal distance between functions, Equation (16), and the vertical distance between functions, Equation (21). This turns out to be the case, Lemma 25. Next, we need to compare the pseudo-inverse of the event-based upper arrival curve $\bar{\alpha}_i^{+-1}$ and the pseudo-inverse of the event-based lower service curve $\bar{\beta}_i^{-1}$ with the event distance function δ_i^- and the multiple event busy time function B_i^+ . The former is straight forward, Remark 26, however the latter is more challenging. For that we need to determine what the pseudo-inverse of the resource-based lower service curve β_i^{-1} is, Lemma 27. From it, we then have to derive the pseudo-inverse of the event-based lower service curve $\bar{\beta}_i^{-1}$, Lemma 28. With that we can compare the pseudo-inverse of the event-based lower service curve $\bar{\beta}_i^{-1}$ with the multiple event busy time function B_i^+ , Lemma 29. Lastly, we need to verify that the different domains of the pseudo-inverse of the event-based lower service curve $\bar{\beta}_i^{-1}$ and the multiple event busy time function B_i^+ do not affect the comparison, Lemma 30. After all these steps we can finally prove the equality of the upper bound of the worst-case response-time, Equation (9), and the upper bound for the delay, Equation (17), in Theorem 31.

► **Lemma 25.** *Let $f, g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions that are not bounded above, then the largest horizontal distance between f and g is equal to the largest vertical distance between g^{-1} and f^{-1} .*

$$f \leftrightarrow g = g^{-1} \downarrow f^{-1} \tag{22}$$

Proof. f and g are both increasing, so we use the equality of Equation (52). Both functions f and g are also not bounded above, therefore we use Equation (49), Definition 7, and Definition 18.

$$f \leftrightarrow g \stackrel{(52)}{=} (g \overline{\circ} f)^{-1}(0) \stackrel{(49)}{=} (g^{-1} \circ f^{-1})(0) \stackrel{(5a)}{=} \sup_{\xi \in \mathbb{R}_0^+} \{g^{-1}(\xi) - f^{-1}(\xi)\} \stackrel{(21)}{=} g^{-1} \downarrow f^{-1} \quad \blacktriangleleft$$

► **Remark 26.** Let Γ be a set of n independent tasks as described in Section 3.1 and let Assumption 22 hold, then for any task τ_i of Γ the event distance function δ_i^- is equal to the pseudo-inverse of the event-based upper arrival curve $\bar{\alpha}_i^{+-1}$.

$$\delta_i^- = \bar{\alpha}_i^{+-1} \quad (23)$$

Proof. The minimum event distance function is equal to the pseudo-inverse of the upper event load function Equation (8). And the upper event load function is equal to the event-based upper service curve, Assumption 22.

$$\delta_i^- \stackrel{(8)}{=} \eta_i^{+-1} = \bar{\alpha}_i^{+-1} \quad \blacktriangleleft$$

► **Lemma 27.** Let Γ be a set of n independent tasks as described in Section 3.1 and let Assumptions 20 and 21 hold, then the pseudo-inverse of the resource-based lower service curve β_i^{-1} of task τ_i is

$$\beta_i^{-1}(r) = \min_{\Delta \in \mathbb{R}_0^+} \left\{ \Delta : \Delta = \beta^{-1} \left(r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \right) \right\} \quad (24)$$

Proof. Let $f_i(\lambda) := \beta^-(\lambda) - \sum_{j=1}^{i-1} \alpha_j^+(\lambda)$ and $g_{i,r}(\Delta) := \beta^{-1} \left(r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \right)$.

- (a) β^- is upper semi-continuous: Follows from Assumption 20 and Lemma 36.
- (b) α_i^+ is lower semi-continuous: Follows from Assumption 20 and Lemma 36.
- (c) $-\alpha_i^+$ is upper semi-continuous: Follows from (b) and Lemma 34.
- (d) f_i is upper semi-continuous: Follows from (a) and (c) and Lemma 35.
- (e) $\beta_i^-(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{f_i(\lambda)\}$, the resource-based lower service curve of task τ_i is the increasing closure of f_i : Follows from Equation (50).
- (f) f_i is not bounded above: Let x be a non-negative real number and I_x be the interval $[0, x]$. Because f_i is upper semi-continuous, (d), and I_Δ is a compact set, then f_i achieves its maximum in I_Δ . Therefore, for every $\Delta \in \mathbb{R}_0^+$ a $\lambda \in I_\Delta$ exists such that $\beta_i^-(\Delta) = f_i(\lambda)$. From Assumption 21 we have that β_i^- is not bounded above, so for every $r \in \mathbb{R}_0^+$ there exist a $\Delta \in \mathbb{R}_0^+$ and subsequently a $\lambda \in \mathbb{R}_0^+$ such that $r < \beta_i^-(\Delta) = f_i(\lambda)$. Therefore, f_i is not bounded above.
- (g) $g_{i,r}$ is increasing: Follows from Assumption 20 and Equation (42a) and that increasing functions are closed under addition and composition.
- (h) $r \leq f_i(\Delta) \Leftrightarrow \Delta \geq g_{i,r}(\Delta)$: Follows from Assumption 20 and Equation (41)

$$\begin{aligned} r \leq f_i(\Delta) &\Leftrightarrow r \leq \beta^-(\Delta) - \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \Leftrightarrow \beta^-(\Delta) \geq r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \\ &\stackrel{(41)}{\Leftrightarrow} \Delta \geq \beta^{-1} \left(r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \right) \Leftrightarrow \Delta \geq g_{i,r}(\Delta) \end{aligned}$$

- (i) $g_{i,r}$ has a smallest fix-point: f_i is not bounded above, (f), so we have that for any $r \in \mathbb{R}_0^+$ a $\Delta \in \mathbb{R}_0^+$ exists such that

$$f_i(\Delta) > r \Leftrightarrow \beta^-(\Delta) - \sum_{j=1}^{i-1} \alpha_j^+(\Delta) > r \Leftrightarrow \beta^-(\Delta) > r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)$$

7:14 Response-Time Analysis vs. Real-Time Calculus

holds. This implies that $\beta^-(\Delta) \geq r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)$. Because of Assumption 20, we apply Equation (41), and we get $\Delta \geq \beta^{-\overline{-1}}\left(r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)\right) = g_{i,r}(\Delta)$. Let I be the closed interval $[0, \Delta]$ of real numbers, then I is a complete lattice. Because of (g) and $g_{i,r}(\Delta) \leq \Delta$, the restriction of $g_{i,r}$ to I maps to itself, i.e. $g_r(I) \subseteq I$. Therefore, according to Lemma 56, g_r has a smallest fix-point in I .

Equation (24) follows from Equation (47), because of (d) and (e), (h), and (i).

$$\begin{aligned} \beta_i^{-\overline{-1}}(r) &\stackrel{(47)}{=} f_i^{-1}(r) \stackrel{(4a)}{\stackrel{(3b)}{\Delta \in \mathbb{R}_0^+}} \inf \{\Delta : r \leq f_i(\Delta)\} \stackrel{(h)}{=} \inf_{\Delta \in \mathbb{R}_0^+} \{\Delta : \Delta \geq g_{i,r}(\Delta)\} \\ &\stackrel{(i)}{=} \min_{\Delta \in \mathbb{R}_0^+} \{\Delta : \Delta = g_{i,r}(\Delta)\} = \min_{\Delta \in \mathbb{R}_0^+} \left\{ \Delta : \Delta = \beta^{-\overline{-1}}\left(r + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)\right) \right\} \quad \blacktriangleleft \end{aligned}$$

► **Lemma 28.** *Let Γ be a set of n independent tasks as described in Section 3.1 and let Assumptions 20 and 21 hold, then the pseudo-inverse of the event-based lower service curve $\overline{\beta}_i^{-\overline{-1}}$ of task τ_i is*

$$\overline{\beta}_i^{-\overline{-1}}(q) = \min_{\Delta \in \mathbb{R}_0^+} \left\{ \Delta : \Delta = \beta^{-\overline{-1}}\left(\gamma_i^+(q) + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)\right) \right\} \quad (25)$$

Proof. We expand the event-based lower service curve into the composition of the pseudo-inverse of the upper workload curve and the resource-based lower service curve. The pseudo-inverse of the upper workload curve is increasing, Equation (42b), and right-continuous, Equation (43b). So, we expand the pseudo-inverse of the composition according to Equation (45b). The upper workload curve is increasing and left-continuous according to Assumption 20, therefore the two pseudo-inverse operations cancel each other out, Equation (46a).

$$\overline{\beta}_i^{-\overline{-1}} \stackrel{(14)}{=} \left(\gamma_i^{+\overline{-1}} \circ \beta_i^-\right)^{-1} \stackrel{(45b)}{=} \beta_i^{-\overline{-1}} \circ \gamma_i^{+\overline{-1}\overline{-1}} \stackrel{(46a)}{=} \beta_i^{-\overline{-1}} \circ \gamma_i^+$$

We satisfy the antecedents of Lemma 27, and so we get

$$\left(\beta_i^{-\overline{-1}} \circ \gamma_i^+\right)(q) \stackrel{(24)}{=} \min_{\Delta \in \mathbb{R}_0^+} \left\{ \Delta : \Delta = \beta^{-\overline{-1}}\left(\gamma_i^+(q) + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)\right) \right\} \quad \blacktriangleleft$$

► **Lemma 29.** *Let Γ be a set of n independent tasks as described in Section 3.1 and let Assumptions 20–24 hold, then for any task τ_i of Γ the multiple event busy time function B_i^+ is equal of the pseudo-inverse of the event-based lower service curve $\overline{\beta}_i^{-\overline{-1}}$*

$$B_i^+ = \overline{\beta}_i^{-\overline{-1}} \quad (26)$$

Proof. First we abbreviate some expressions. Let $f_i(q, \Delta) := \beta^{-\overline{-1}}\left(\gamma_i^+(q) + \sum_{j=1}^{i-1} \alpha_j^+(\Delta)\right)$ and $g_i(q, \Delta) := q \cdot c_i^+ + \sum_{j=1}^{i-1} (\eta_j^+(\Delta) \cdot c_j^+)$.

(a) $\overline{\beta}_i^{-\overline{-1}}(q) = \min_{\Delta \in \mathbb{R}_0^+} \{\Delta : \Delta = f_i(q, \Delta)\}$: Follows from Lemma 28.

(b) $\beta^{-\overline{-1}}(r) = r$: Follows from Assumption 23 and Definition 5.

(c) $f_i(q, \Delta) = g_i(q, \Delta)$: Follows from (b), Equation (15), Assumption 24, and Assumption 22

$$\begin{aligned} f_i(q, \Delta) &= \beta^{-1} \left(\gamma_i^+(q) + \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \right) \stackrel{(b)}{=} \gamma_i^+(q) + \sum_{j=1}^{i-1} \alpha_j^+(\Delta) \\ &\stackrel{(15)}{=} \gamma_i^+(q) + \sum_{j=1}^{i-1} \gamma_j^+(\bar{\alpha}_j^+(\Delta)) = q \cdot c_i^+ + \sum_{j=1}^{i-1} (\bar{\alpha}_j^+(\Delta) \cdot c_j^+) \\ &= q \cdot c_i^+ + \sum_{j=1}^{i-1} (\eta_j^+(\Delta) \cdot c_j^+) = g_i(q, \Delta) \end{aligned}$$

(d) $B_i^+(q) = \min_{\Delta \in \mathbb{R}_0^+} \{\Delta : \Delta = g_i(q, \Delta)\}$: Follows from Equation (10).

Equation (26) follows directly from (a), (c), and (d)

$$\bar{\beta}_i^{-1}(q) \stackrel{(a)}{=} \min_{\Delta \in \mathbb{R}_0^+} \{\Delta : \Delta = f_i(q, \Delta)\} \stackrel{(c)}{=} \min_{\Delta \in \mathbb{R}_0^+} \{\Delta : \Delta = g_i(q, \Delta)\} \stackrel{(d)}{=} B_i^+(q) \quad \blacktriangleleft$$

► **Lemma 30.** Let $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ and $g: \mathbb{R}_0^+ \rightarrow \mathbb{N}_0$, then

$$\max_{n \in \mathbb{N}_0} \{f^{-1}(n) - g^{-1}(n)\} = \sup_{r \in \mathbb{R}_0^+} \{f^{-1}(r) - g^{-1}(r)\} \quad (27)$$

Proof. Let $h := f^{-1} - g^{-1}, b_X := \sup_{x \in X} \{h(x)\}$, r a non-negative real number and $n := \lceil r \rceil$ a non-negative integer.

(a) $b_{\mathbb{N}_0} \leq b_{\mathbb{R}_0^+}$: Follows from \mathbb{N}_0 being a subset of \mathbb{R}_0^+ .

(b) $f^{-1}(r) \leq f^{-1}(n)$: Follows from $r \leq n$ and Equation (42a).

(c) $g^{-1}(r) = g^{-1}(n)$: Follows from Definitions 4 and 5 and because the co-domain of g are the non-negative integers

$$g^{-1}(r) = \inf_{x \in \mathbb{R}_0^+} \{x : r \leq g(x)\} = \inf_{x \in \mathbb{R}_0^+} \{x : \lceil r \rceil \leq g(x)\} = g^{-1}(\lceil r \rceil) = g^{-1}(n)$$

(d) $h(r) \leq h(n)$: Follows from (b) and (c)

(e) $b_{\mathbb{R}_0^+} \leq b_{\mathbb{N}_0}$: Follows because for every $r \in \mathbb{R}_0^+$ an $n \in \mathbb{N}_0$ exists such that $h(r) \leq h(n)$.

This Lemma follows from (a) and (e). \blacktriangleleft

► **Theorem 31.** Let Γ be a set of n independent tasks as described in Section 3.1 and let Assumptions 20–24 hold, then for any task τ_i of Γ the upper bound for the worst-case response time from the response-time analysis is equal to the upper bound for the delay from the real-time calculus.

$$\max_{q \in \mathbb{N}_0} \{B_i^+(q) - \delta_i^-(q)\} = \bar{\alpha}_i^+ \leftrightarrow \bar{\beta}_i^- \quad (28)$$

Proof. First we substitute the multiple event busy time function B_i^+ with the pseudo-inverse of the event-based lower service curve $\bar{\beta}_i^{-1}$, Lemma 29 and the event distance function δ_i^- with the pseudo-inverse of the event-based upper arrival curve $\bar{\alpha}_i^{+1}$, Remark 26.

$$\max_{q \in \mathbb{N}_0} \{B_i^+(q) - \delta_i^-(q)\} \stackrel{(26)}{=} \max_{q \in \mathbb{N}_0} \left\{ \bar{\beta}_i^{-1}(q) - \delta_i^-(q) \right\} \stackrel{(23)}{=} \max_{q \in \mathbb{N}_0} \left\{ \bar{\beta}_i^{-1}(q) - \bar{\alpha}_i^{+1}(q) \right\}$$

Next we interchange the maximum \max with the supremum \sup and change the set from the non-negative integers \mathbb{N}_0 to the non-negative real numbers \mathbb{R}_0^+ (Lemma 30). This results in the vertical distance between the pseudo-inverse of the event-based lower service curve $\overline{\beta}_i^{-1}$ and the pseudo-inverse of the event-based upper arrival curve $\overline{\alpha}_i^{+1}$ by Definition 18:

$$\max_{q \in \mathbb{N}_0} \left\{ \overline{\beta}_i^{-1}(q) - \overline{\alpha}_i^{+1}(q) \right\} \stackrel{(27)}{=} \sup_{\lambda \in \mathbb{R}_0^+} \left\{ \overline{\beta}_i^{-1}(\lambda) - \overline{\alpha}_i^{+1}(\lambda) \right\} \stackrel{(21)}{=} \overline{\beta}_i^{-1} \downarrow \overline{\alpha}_i^{+1}$$

Finally, we apply the equality between the vertical distance of the pseudo-inverse functions and the horizontal distance between the functions, Lemma 25, so that we ultimately get Equation (17), the upper bound for the delay.

$$\overline{\beta}_i^{-1} \downarrow \overline{\alpha}_i^{+1} \stackrel{(22)}{=} \overline{\alpha}_i^+ \leftrightarrow \overline{\beta}_i^- \quad \blacktriangleleft$$

5 Summary

We looked into the existing analyses for real-time systems with a single processor that uses the fixed-priority preemptive scheduling algorithm to process a set of independent tasks that do not share any resources other than the processor. One is the response-time analysis that Schliecker presents in [17] and the other is the real-time calculus that Wandeler describes in [26]. Both use abstract event models and produce upper bounds on the amount of time that the processor needs to complete the tasks. The existing empirical comparisons could only give us indications as how these two analyses compare.

However, we can now give a definite answer. We gave a mathematical proof that both analyses produce for the investigated type of systems identical upper bounds. So, from a mathematical point of view both analyses are equivalent and regarding the results it does not matter which analysis is used. However, a different criteria, like run-time complexity, can favour one over the other.

References

- 1 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993. doi:10.1049/sej.1993.0034.
- 2 Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real Time Systems*, 8(2-3):173–198, March 1995. doi:10.1007/BF01094342.
- 3 Marc Boyer and Pierre Roux. A common framework embedding network calculus and event stream theory. Technical report, ONERA - The french aerospace lab, May 2016. Preprint. URL: <https://hal.archives-ouvertes.fr/hal-01311502>.
- 4 Marc Boyer and Pierre Roux. Embedding network calculus and event stream theory in a common model. In *21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, September 6-9, 2016*, pages 1–8. IEEE, 2016. doi:10.1109/ETFA.2016.7733565.
- 5 Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10190–10195. IEEE Computer Society, 2003. doi:10.1109/DATE.2003.10083.
- 6 Rene L. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991. doi:10.1109/18.61109.

- 7 Rene L. Cruz. A Calculus for Network Delay, Part II: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991. doi:10.1109/18.61110.
- 8 Markus Fidler. A Survey of Deterministic and Stochastic Service Curve Models in the Network Calculus. *IEEE Communications Surveys & Tutorials*, 12(1):59–86, 2010. doi:10.1109/SURV.2010.020110.00019.
- 9 Mathai Joseph and Paritosh Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, January 1986. doi:10.1093/comjnl/29.5.390.
- 10 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001. doi:10.1007/3-540-45318-0.
- 11 John P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 201–209. IEEE Computer Society, December 1990. doi:10.1109/REAL.1990.128748.
- 12 C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973. doi:10.1145/321738.321743.
- 13 Martin Naedele, Lothar Thiele, and Michael Eisenring. Characterizing Variable Task Releases and Processor Capacities. Technical report, ETH Zürich, 1999. doi:10.3929/ethz-a-004289034.
- 14 Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007, September 30 - October 3, 2007, Salzburg, Austria*, pages 193–202. ACM, 2007. doi:10.1145/1289927.1289959.
- 15 Victor Pollex, Steffen Kollmann, and Frank Slomka. Generalizing Response-Time Analysis. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2010, Macau, SAR, China, 23-25 August 2010*, pages 203–211. IEEE Computer Society, 2010. doi:10.1109/RTCSA.2010.36.
- 16 Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University Carolo-Wilhemina of Braunschweig, 2005. doi:10.24355/dbbs.084-200511080100-362.
- 17 Simon Schliecker. *Performance Analysis of Multiprocessor Real-Time Systems with Shared Resources*. PhD thesis, Technical University Carolo-Wilhemina of Braunschweig, 2011. doi:10.24355/dbbs.084-201111210932-0.
- 18 Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In Catherine H. Gebotys and Grant Martin, editors, *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 185–190. ACM, 2008. doi:10.1145/1450135.1450177.
- 19 Frank Slomka and Mohammadreza Sadeghi. HeRTA: Heaviside Real-Time Analysis, 2020. Preprint. doi:10.48550/arXiv.2007.12112.
- 20 Frank Slomka and Mohammadreza Sadeghi. Beyond the limitations of real-time scheduling theory: a unified scheduling theory for the analysis of real-time systems. *SICS Software-Intensive Cyber Physical Systems*, 35(3-4):201–236, 2021. doi:10.1007/s00450-021-00429-1.
- 21 Frank Slomka and Mohammadreza Sadeghi. Work-in-Progress Abstract: On the relationship between scheduling theory and real-time calculus. In *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2021, Houston, TX, USA, August 18-20, 2021*, pages 195–197. IEEE, 2021. doi:10.1109/RTCSA52859.2021.00030.
- 22 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. doi:10.2140/pjm.1955.5.285.

- 23 Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems, ISCAS 2000, Emerging Technologies for the 21st Century, Geneva, Switzerland, 28-31 May 2000, Proceedings*, pages 101–104. IEEE, 2000. doi:10.1109/ISCAS.2000.858698.
- 24 Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, April 1994. doi:10.1016/0165-6074(94)90080-9.
- 25 Ken W. Tindell, Alan Burns, and Andy J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, March 1994. doi:10.1007/BF01088593.
- 26 Ernesto Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2006. doi:10.3929/ethz-a-005328667.

A Properties of Semi-Continuous Functions

► **Definition 32** (Lower and Upper Limit). *Let $f: X \rightarrow \mathbb{R}$ be a function from a subset X of the real numbers, then the lower respectively upper limit of f at an accumulation point x for X is*

$$\liminf_{\xi \rightarrow x} f(\xi) := \sup_{r>0} \inf_{\xi \in X} \{f(\xi) : 0 < |\xi - x| < r\} \text{ respectively} \tag{29a}$$

$$\limsup_{\xi \rightarrow x} f(\xi) := \inf_{r>0} \sup_{\xi \in X} \{f(\xi) : 0 < |\xi - x| < r\}. \tag{29b}$$

Note that in Definition 32 the domain X of f has no further restrictions. In particular X does not have to be dense.

► **Definition 33** (Semi-Continuity). *Let $f: X \rightarrow \overline{\mathbb{R}}$ be a function from a subset X of the real numbers to the extended real numbers, then f is lower respectively upper semi-continuous at $x \in X$ if*

$$f(x) \leq \liminf_{\xi \rightarrow x} f(\xi) \text{ respectively} \tag{30a}$$

$$f(x) \geq \limsup_{\xi \rightarrow x} f(\xi). \tag{30b}$$

If f is lower respectively upper semi-continuous at every element $x \in X$, then we call f a lower respectively upper semi-continuous function.

► **Lemma 34** (Duality of Semi-Continuity). *Let $f: X \rightarrow \overline{\mathbb{R}}$ be a function from a subset X of the real numbers to the extended real numbers, then*

$$f \text{ is lower semi-continuous} \Leftrightarrow -f \text{ is upper semi-continuous} \tag{31}$$

Proof. Let x be an element of X , then it follows from Definition 33, i.e.

$$f(x) \leq \liminf_{\xi \rightarrow x} f(\xi) \Leftrightarrow -f(x) \geq -\liminf_{\xi \rightarrow x} f(\xi) \Leftrightarrow -f(x) \geq \limsup_{\xi \rightarrow x} -f(\xi) \quad \blacktriangleleft$$

► **Lemma 35** (Semi-Continuous Functions are Closed under Addition). *Let $f, g: X \rightarrow \overline{\mathbb{R}}$ be lower respectively upper semi-continuous functions from a subset X of the real numbers to the extended real numbers, then $f + g$ is lower respectively upper semi-continuous, if no $x \in X$ exists such that $f(x) + g(x)$ is of the type $-\infty + \infty$.*

Proof. Let x be an element of X , then it follows from Definition 33, i.e.

$$f(x) + g(x) \leq \liminf_{\xi \rightarrow x} f(x) + \liminf_{\xi \rightarrow x} g(x) \leq \liminf_{\xi \rightarrow x} f(x) + g(x) \text{ respectively}$$

$$f(x) + g(x) \geq \limsup_{\xi \rightarrow x} f(x) + \limsup_{\xi \rightarrow x} g(x) \geq \limsup_{\xi \rightarrow x} f(x) + g(x). \quad \blacktriangleleft$$

► **Lemma 36.** Let $f: I \rightarrow \mathbb{R}$ be an increasing function from an interval I of the real numbers, then

$$f \text{ is lower semi-continuous} \quad \text{if } f \text{ is left-continuous} \quad (32a)$$

$$f \text{ is upper semi-continuous} \quad \text{if } f \text{ is right-continuous} \quad (32b)$$

Proof. Let $x \in I$ and f be left-continuous respectively right-continuous, then

$$\liminf_{\xi \rightarrow x} f(\xi) = \sup_{r>0} \inf_{\xi \in I} \{f(\xi) : 0 < |\xi - x| < r\} \geq \sup_{r>0} f(x - r) = f(x)$$

$$\limsup_{\xi \rightarrow x} f(\xi) = \inf_{r>0} \sup_{\xi \in I} \{f(\xi) : 0 < |\xi - x| < r\} \leq \inf_{r>0} f(x + r) = f(x) \quad \blacktriangleleft$$

B Properties of Pseudo-Inverses

► **Lemma 37.** Confer [3, Proposition 6, (6) and (7)]. Let $f: X \rightarrow Y$ be a mapping from a subset X of a complete lattice L to a partially ordered set Y , then for an element $x \in X$ and an element $y \in Y$

$$y \leq f(x) \Rightarrow f^{-1}(y) \leq x \text{ and} \quad (33a)$$

$$f(x) \leq y \Rightarrow x \leq f^{-1}(y). \quad (33b)$$

Proof of (33a). Let $y \leq f(x)$, then $x \in X_{y \leq f}$, therefore $f^{-1}(y) = \inf X_{y \leq f} \leq x$. \blacktriangleleft

Proof of (33b). Let $f(x) \leq y$, then $x \in X_{f \leq y}$, therefore $x \leq \sup X_{f \leq y} = f^{-1}(y)$. \blacktriangleleft

► **Lemma 38.** Confer [3, Proposition 6, (14) and (12)]. Let $f: X \rightarrow Y$ be a mapping from a subset X of a complete lattice L to a totally ordered set Y , then for an element $x \in X$ and an element $y \in Y$

$$x < f^{-1}(y) \Rightarrow f(x) < y \text{ and} \quad (34a)$$

$$f^{-1}(y) < x \Rightarrow y < f(x). \quad (34b)$$

Proof of (34a). Let $x < f^{-1}(y)$, then the partial order of X implies that $\neg(f^{-1}(y) \leq x)$. We can then apply Equation (33a) whose contraposition implies $\neg(y \leq f(x))$. The total order of Y then implies $f(x) < y$. \blacktriangleleft

Proof of (34b). Let $f^{-1}(y) < x$, then the partial order of X implies that $\neg(x \leq f^{-1}(y))$. We can then apply Equation (33b) whose contraposition implies $\neg(f(x) \leq y)$. The total order of Y then implies $y < f(x)$. \blacktriangleleft

► **Lemma 39.** Confer [3, Proposition 6, (8), (9), and (10)]. Let $f: X \rightarrow Y$ be an isotone mapping from a totally ordered subset X of a complete lattice L to a partially ordered set Y , then for an element $x \in X$ and an element $y \in Y$

$$f(x) < y \Rightarrow x \leq f^{-1}(y) \text{ and} \quad (35a)$$

$$y < f(x) \Rightarrow f^{-1}(y) \leq x. \quad (35b)$$

$$f^{-1}(y) < x \Rightarrow y \leq f(x) \quad (36)$$

Proof of (35a). Let $f(x) < y$ and let ξ be an element of $X_{y \leq f}$, then we have $f(x) < y \leq f(\xi)$. This implies $x < \xi$, because f is isotone and X is totally ordered. Therefore, x is a lower bound of $X_{y \leq f}$, so $x \leq \inf X_{y \leq f} = f^{-1}(y)$. ◀

Proof of (35b). Let $y < f(x)$ and let ξ be an element of $X_{f \leq y}$, then we have $f(\xi) \leq y < f(x)$. This implies $\xi < x$, because f is isotone and X is totally ordered. Therefore, x is an upper bound of $X_{f \leq y}$, so $f^{-1}(y) = \sup X_{f \leq y} \leq x$. ◀

Proof of (36). Let $f^{-1}(y) < x$, the infimum of $X_{y \leq f}$ is less than x . Then a $\xi \in X_{y \leq f}$ must exist such that $f^{-1}(y) \leq \xi < x$, due to how the infimum is defined and because X is a totally ordered set. On the one hand from $\xi \in X_{y \leq f}$ follows that $y \leq f(\xi)$, on the other hand from $\xi < x$ follows $f(\xi) \leq f(x)$, because f is isotone, subsequently $y \leq f(x)$ holds. ◀

► **Lemma 40.** Let $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be a function that is not bounded above, then the image of f^{-1} is a subset of the non-negative real numbers.

$$f^{-1}(\mathbb{R}_0^+) \subseteq \mathbb{R}_0^+, \quad (37)$$

Proof. Let $\mathbb{R}_0^+ \cup \{\infty\}$ be the complete lattice of which \mathbb{R}_0^+ is a subset of and let y be in \mathbb{R}_0^+ . Since f is not bounded above, a non-negative real number x exists ($x \in \mathbb{R}_0^+$) that satisfies $y < f(x)$.

On the one hand $y < f(x)$ implies $f^{-1}(y) \leq x$ according to (33a). On the other hand the relation $\mathbb{R}_0^+ \supseteq X_{y \leq f}$ implies $0 = \inf \mathbb{R}_0^+ \leq \inf X_{y \leq f} = f^{-1}(y)$. Therefore $f^{-1}(y)$ is a non-negative real number. ◀

► **Lemma 41.** Confer [3, Proposition 6, (17), and (18)]. Let $f: X \rightarrow \mathbb{R}$ be an increasing and left-continuous function from a subset X of the real numbers \mathbb{R} , then for an element $x \in X$ and an element $y \in \mathbb{R}$

$$y < f(x) \Rightarrow f^{-1}(y) < x \quad (38a)$$

$$x \leq f^{-1}(y) \Rightarrow f(x) \leq y \quad (38b)$$

Proof. Let $y < f(x)$. Because f is left-continuous a $\delta > 0$ exists for $\varepsilon = f(x) - y > 0$ such that $|f(x) - f(\xi_1)| < \varepsilon$ holds for any $\xi_1 \in X$ where $x - \delta < \xi_1 < x$. From $|f(x) - f(\xi_1)| < f(x) - y$ it follows that $y < f(\xi_1)$. Let ξ_2 be an element of $X_{f \leq y}$, hence $f(\xi_2) \leq y < f(\xi_1)$ holds. Since f is increasing and X is totally ordered it follows that $\xi_2 < \xi_1$, therefore ξ_1 is an upper bound of $X_{f \leq y}$. So $\sup X_{f \leq y} = f^{-1}(y) \leq \xi_1 < x$. The contraposition $x \leq f^{-1}(y) \Rightarrow f(x) \leq y$ follows directly. ◀

► **Lemma 42.** Confer [3, Proposition 6, (15), and (16)]. Let $f: X \rightarrow \mathbb{R}$ be an increasing and right-continuous function from a subset X of the real numbers \mathbb{R} , then for an element $x \in X$ and an element $y \in \mathbb{R}$

$$f(x) < y \Rightarrow x < f^{-1}(y) \quad (39a)$$

$$f^{-1}(y) \leq x \Rightarrow y \leq f(x) \quad (39b)$$

Proof. Let $f(x) < y$. Because f is right-continuous a $\delta > 0$ exists for $\varepsilon = y - f(x) > 0$ such that $|f(\xi_1) - f(x)| < \varepsilon$ holds for any $\xi_1 \in X$ where $x < \xi_1 < x + \delta$. From $|f(\xi_1) - f(x)| < y - f(x)$ it follows that $f(\xi_1) < y$. Let ξ_2 be an element of $X_{y \leq f}$, hence $f(\xi_1) < y \leq f(\xi_2)$ holds. Since f is increasing and X is totally ordered it follows that $\xi_1 < \xi_2$, therefore ξ_1 is a lower bound of $X_{y \leq f}$. So $x < \xi_1 \leq f^{-1}(y) = \inf X_{y \leq f}$. The contraposition $f^{-1}(y) \leq x \Rightarrow y \leq f(x)$ follows directly. ◀

► **Lemma 43.** *Let $f: X \rightarrow \mathbb{R}$ be an increasing and left-continuous functions from a subset X of the real numbers \mathbb{R} , then for an element $x \in X$ and an element $y \in \mathbb{R}$*

$$f(x) \leq y \Leftrightarrow x \leq f^{-1}(y) \quad (40)$$

Proof. Follows directly from Equations (33b) and (38b).

$$f(x) \leq y \stackrel{(33b)}{\Rightarrow} x \leq f^{-1}(y) \stackrel{(38b)}{\Rightarrow} f(x) \leq y \quad \blacktriangleleft$$

► **Lemma 44.** *Let $f: X \rightarrow \mathbb{R}$ be an increasing and right-continuous function from a subset X of the real numbers \mathbb{R} , then for an element $x \in X$ and an element $y \in \mathbb{R}$*

$$y \leq f(x) \Leftrightarrow f^{-1}(y) \leq x \quad (41)$$

Proof. Follows directly from Equations (33a) and (39b).

$$y \leq f(x) \stackrel{(33a)}{\Rightarrow} f^{-1}(y) \leq x \stackrel{(39b)}{\Rightarrow} y \leq f(x) \quad \blacktriangleleft$$

► **Lemma 45 (Isotone pseudo-inverse).** *Confer [3, Proposition 3]. Let $f: X \rightarrow Y$ be a mapping from a subset X of a complete lattice L to a partially ordered set Y , then*

$$f^{-1} \text{ is isotone} \quad (42a)$$

$$f^{-1} \text{ is isotone} \quad (42b)$$

Proof of (42a). Confer [10, p. 131, Theorem 3.1.2]. Let y_1 and y_2 be elements of Y with $y_1 \leq y_2$ and let x be an element of $X_{y_2 \leq f}$. Then, we have $f(x) \geq y_2 \geq y_1$, so x is also an element of $X_{y_1 \leq f}$. Therefore $X_{y_1 \leq f}$ is a superset of $X_{y_2 \leq f}$ and subsequently $f^{-1}(y_1) = \inf X_{y_1 \leq f} \leq \inf X_{y_2 \leq f} = f^{-1}(y_2)$. ◀

Proof of (42b). Let y_1 and y_2 be elements of Y with $y_1 \leq y_2$ and let x be an element of $X_{f \leq y_1}$. Then, we have $f(x) \leq y_1 \leq y_2$, so x is also an element of $X_{f \leq y_2}$. Therefore $X_{f \leq y_1}$ is a subset of $X_{f \leq y_2}$ and subsequently $f^{-1}(y_1) = \sup X_{f \leq y_1} \leq \sup X_{f \leq y_2} = f^{-1}(y_2)$. ◀

► **Lemma 46 (Directional continuity of pseudo-inverse).** *Confer [3, Proposition 5]. Let $f: I \rightarrow \mathbb{R}$ be an increasing function from an interval I of the real numbers \mathbb{R} , then*

$$f^{-1} \text{ is left-continuous} \quad (43a)$$

$$f^{-1} \text{ is right-continuous} \quad (43b)$$

Proof of (43a). Let y be a real number and $\varepsilon > 0$.

Case $f^{-1}(y) - \varepsilon < \inf I$. For any $\delta > 0$ and for any $v \in (y - \delta, y)$ we have $f^{-1}(y) - \varepsilon < \inf I \leq f^{-1}(v)$. Therefore, we get $f^{-1}(y) - f^{-1}(v) < \varepsilon$.

Case $\inf I \leq f^{-1}(y) - \varepsilon$. Note that $f^{-1}(y) - \varepsilon < f^{-1}(y) \leq \sup I$. Because I is an interval of the real numbers, a real number $\xi \in I$ exists such that $f^{-1}(y) - \varepsilon < \xi < f^{-1}(y)$. By applying Equation (34a) $\xi < f^{-1}(y)$ implies $f(\xi) < y$. Let $\delta := y - f(\xi)$, then for all $v \in \mathbb{R} \cap (y - \delta, y)$ we have $y - \delta = f(\xi) < v$. Through Equation (35a) this implies $\xi \leq f^{-1}(v)$. So, together with $f^{-1}(y) - \varepsilon < \xi$ we get $f^{-1}(y) - f^{-1}(v) < \varepsilon$. ◀

7:22 Response-Time Analysis vs. Real-Time Calculus

Proof of (43b). Let y be a real number and $\varepsilon > 0$.

Case $\sup I < f^{-1}(y) + \varepsilon$. For any $\delta > 0$ and for any $v \in (y, y + \delta)$ we have $f^{-1}(v) \leq \sup I < f^{-1}(y) + \varepsilon$. Therefore, we get $f^{-1}(v) - f^{-1}(y) < \varepsilon$.

Case $f^{-1}(y) + \varepsilon \leq \sup I$. Note that $\inf I \leq f^{-1}(y) < f^{-1}(y) + \varepsilon$. Because I is an interval of the real numbers, a real number $\xi \in I$ exists such that $f^{-1}(y) < \xi < f^{-1}(y) + \varepsilon$. By applying Equation (34b) $f^{-1}(y) < \xi$ implies $y < f(\xi)$. Let $\delta := f(\xi) - y$, then for all $v \in \mathbb{R} \cap (y, y + \delta)$ we have $v < f(\xi) = y + \delta$. Through Equation (35b) this implies $f^{-1}(v) \leq \xi$. So, together with $\xi < f^{-1}(y) + \varepsilon$ we get $f^{-1}(v) - f^{-1}(y) < \varepsilon$. ◀

► **Lemma 47** (Pseudo-inverse operator is antitone). *Let $f: X \rightarrow Y$ and $g: X \rightarrow Y$ be mappings from a subset X of a complete lattice L to a partially order set Y , then*

$$f \leq g \Rightarrow f^{-1} \geq g^{-1} \quad (44a)$$

$$f \leq g \Rightarrow f^{-1} \geq g^{-1} \quad (44b)$$

Proof of (44a). Let $f \leq g$, y an element of Y , and x an element of $X_{y \leq f}$, then $y \leq f(x) \leq g(x)$, so $X_{y \leq f} \subseteq X_{y \leq g}$, and subsequently $f^{-1}(y) = \inf X_{y \leq f} \geq \inf X_{y \leq g} = g^{-1}(y)$. ◀

Proof of (44b). Let $f \leq g$, y and element of Y , and x an element of $X_{g \leq y}$, then $y \geq g(x) \geq f(x)$, so $X_{g \leq y} \subseteq X_{f \leq y}$, and subsequently $f^{-1}(y) = \sup X_{f \leq y} \geq \sup X_{g \leq y} = g^{-1}(y)$. ◀

► **Lemma 48** (Pseudo-inverse of a composition). *Confer [3, Proposition 6, (25) and (24)]. Let $f: X \rightarrow Y$ be a function from a subset X of a complete lattice L to a subset Y of the real numbers and let $g: Y \rightarrow \mathbb{R}$ be an increasing function, then*

$$(g \circ f)^{-1} = (f^{-1} \circ g^{-1}) \quad \text{if } g \text{ is left-continuous} \quad (45a)$$

$$(g \circ f)^{-1} = (f^{-1} \circ g^{-1}) \quad \text{if } g \text{ is right-continuous} \quad (45b)$$

Proof of (45a). Let z be a real number and let g be left-continuous, then we can apply Equation (40) and we get

$$(g \circ f)^{-1}(z) \stackrel{(4b)}{=} \sup_{x \in X} \{x : g(f(x)) \leq z\} \stackrel{(40)}{=} \sup_{x \in X} \{x : f(x) \leq g^{-1}(z)\} \stackrel{(4b)}{=} (f^{-1} \circ g^{-1})(z) \quad \blacktriangleleft$$

Proof of (45b). Let z be a real number and let g be right-continuous, then we can apply Equation (41) and we get

$$(g \circ f)^{-1}(z) \stackrel{(4a)}{=} \inf_{x \in X} \{x : z \leq g(f(x))\} \stackrel{(41)}{=} \inf_{x \in X} \{x : g^{-1}(z) \leq f(x)\} \stackrel{(4a)}{=} (f^{-1} \circ g^{-1})(z) \quad \blacktriangleleft$$

► **Lemma 49** (The Pseudo-Inverse Operators are inverse to each other). *Let $f: X \rightarrow \mathbb{R}$ be an increasing function from a subset X of \mathbb{R} , then*

$$f^{-1-1} = f \quad \text{if } f \text{ is left-continuous} \quad (46a)$$

$$f^{-1-1} = f \quad \text{if } f \text{ is right-continuous} \quad (46b)$$

Proof of (46a). Let x be an element of X and let f be left-continuous, then we can apply Equation (40) and we get

$$f^{-1-1}(x) \stackrel{(4a)}{=} \inf X_{x \leq f^{-1}} \stackrel{(3b)}{=} \inf_{y \in \mathbb{R}} \{y : x \leq f^{-1}(y)\} \stackrel{(40)}{=} \inf_{y \in \mathbb{R}} \{y : f(x) \leq y\} = f(x) \quad \blacktriangleleft$$

Proof of (46b). Let x be an element of X and let f be right-continuous, then we can apply Equation (41) and we get

$$f^{-1\overline{-1}}(x) \stackrel{(4b)}{=} \sup X_{f^{-1} \leq x} \stackrel{(3a)}{=} \sup_{y \in \mathbb{R}} \{y : f^{-1}(y) \leq x\} \stackrel{(41)}{=} \sup_{y \in \mathbb{R}} \{y : y \leq f(x)\} = f(x) \quad \blacktriangleleft$$

► **Lemma 50.** Let $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be an upper semi-continuous function and $g := f^\nearrow$ be the increasing closure of f , then

$$f^{-1} = g^{-1} \quad (47)$$

Proof. Let x be a non-negative real number and let I_x be the interval $[0, x]$, then

(a) $f \leq g$: Follows directly from g being the increasing closure of f , Remark 17.

(b) $f^{-1} \leq g^{-1}$: f is upper semi-continuous, I_x is a compact set, therefore f achieves its maximum in I_x and an element $x_0 \in I_x$ with $x_0 \leq x$ exists where $f(x_0) = g(x)$.

Let y be a non-negative real number. Then, for every element $x \in X_{y \leq g}$ there exists a $x_0 \leq x$ where $y \leq g(x) = f(x_0)$. Therefore, x_0 is an element of $X_{y \leq f}$ and subsequently $f^{-1}(y) = \inf X_{y \leq f} \leq \inf X_{y \leq g} = g^{-1}(y)$.

(c) $f^{-1} \geq g^{-1}$: Follows from (a) and (44a).

Equation (47) follows from (b) and (c). ◀

C Properties of Deconvolution

► **Lemma 51 (Monotonicity of Deconvolution).** Let $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be an increasing function and $g: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ a function, then

$$(f \overline{\circ} g) \text{ is increasing} \quad (48)$$

Proof. Let x_1 and x_2 be non-negative real numbers ($x_1, x_2 \in \mathbb{R}_0^+$) such that $x_1 \leq x_2$.

From $x_1 \leq x_2$ follows that $x_1 + \xi \leq x_2 + \xi$ holds for every $\xi \in \mathbb{R}_0^+$ and subsequently $f(x_1 + \xi) \leq f(x_2 + \xi)$, since f is increasing. Furthermore $f(x_1 + \xi) - g(\xi) \leq f(x_2 + \xi) - g(\xi)$ holds for every $\xi \in \mathbb{R}_0^+$, therefore

$$(f \overline{\circ} g)(x_1) = \inf_{\xi \in \mathbb{R}_0^+} \{f(x_1 + \xi) - g(\xi)\} \leq \inf_{\xi \in \mathbb{R}_0^+} \{f(x_2 + \xi) - g(\xi)\} = (f \overline{\circ} g)(x_2) \quad \blacktriangleleft$$

► **Theorem 52.** Let $f, g: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions that are not bounded above, then

$$(f \overline{\circ} g)^{-1} = (f^{-1} \circ g^{-1}) \quad (49)$$

Proof. Let $y \in \mathbb{R}_0^+$ be a non-negative real number.

(a) $f^{-1}(\mathbb{R}_0^+) \subseteq \mathbb{R}_0^+$: Follows from f being not bounded above and Equation (37).

(b) $g^{-1}(\mathbb{R}_0^+) \subseteq \mathbb{R}_0^+$: Follows from g being not bounded above and Equation (37).

i.e. the images of f^{-1} and g^{-1} are subsets of the non-negative real numbers, therefore $(f^{-1} \circ g^{-1})$ is well-defined.

Part 1 shows that $(f \overline{\circ} g)^{-1} \leq (f^{-1} \circ g^{-1})$:

Let x be a non-negative real number such that $x < (f \overline{\circ} g)^{-1}(y)$. According to (34a) this implies $(f \overline{\circ} g)(x) < y$ and due to (5b) a non-negative real number ξ exists that satisfies $f(x + \xi) - g(\xi) < y$. Furthermore a non-negative real number v exists such that $f(x + \xi) - y < v \leq g(\xi)$, because $g(\xi)$ is a non-negative real number.

On the one hand f is increasing, therefore $f(x + \xi) < y + v$ implies $x + \xi \leq f^{-1}(y + v)$ according to (35a), so $x \leq f^{-1}(y + v) - \xi$. On the other hand $v \leq g(\xi)$ implies $g^{-1}(v) \leq \xi$ according to (33a). This results altogether in $x + g^{-1}(v) \leq f^{-1}(y + v) - \xi + g^{-1}(v) \leq f^{-1}(y + v)$ and due to (a) and (b) ultimately in $x \leq f^{-1}(y + v) - g^{-1}(v) \leq \sup_{\lambda \in \mathbb{R}_0^+} \{f^{-1}(y + \lambda) - g^{-1}(\lambda)\} = (f^{-1} \circ g^{-1})(y)$.

In conclusion, the set $X_{l,y} := \{x \in \mathbb{R}_0^+ : x < (f \overline{\circ} g)^{-1}(y)\}$ is a subset of the set $X_{r,y} := \{x \in \mathbb{R}_0^+ : x \leq (f^{-1} \circ g^{-1})(y)\}$. Since $(f \overline{\circ} g)^{-1}(y) = \sup X_{l,y}$ and $\sup X_{r,y} = (f^{-1} \circ g^{-1})(y)$ this implies $(f \overline{\circ} g)^{-1}(y) \leq (f^{-1} \circ g^{-1})(y)$. Furthermore y is chosen arbitrarily, hence $(f \overline{\circ} g)^{-1} \leq (f^{-1} \circ g^{-1})$.

Part 2 shows that $(f \overline{\circ} g)^{-1} \geq (f^{-1} \circ g^{-1})$:

Let x be a non-negative real number that satisfies $x < (f^{-1} \circ g^{-1})(y)$. Due to (5a) a non-negative real number v exists such that $x < f^{-1}(y + v) - g^{-1}(v)$. Furthermore a non-negative real number ξ exists that satisfies $g^{-1}(v) < \xi < f^{-1}(y + v) - x$ due to (a) and (b).

On the one hand $x + \xi < f^{-1}(y + v)$ implies $f(x + \xi) < y + v$ according to (34a). On the other hand g is increasing, therefore $g^{-1}(v) < \xi$ implies $v \leq g(\xi)$ according to (36). This results altogether in $f(x + \xi) < y + g(\xi)$ and thus $(f \overline{\circ} g)(x) \leq f(x + \xi) - g(\xi) < y$. Because f is increasing, so is $(f \overline{\circ} g)$ according to (48), therefore $(f \overline{\circ} g)(x) < y$ implies $x \leq (f \overline{\circ} g)^{-1}(y)$ according to (35a).

In conclusion the set $X_{r,y} := \{x \in \mathbb{R}_0^+ : x < (f^{-1} \circ g^{-1})(y)\}$ is a subset of the set $X_{l,y} := \{x \in \mathbb{R}_0^+ : x \leq (f \overline{\circ} g)^{-1}(y)\}$. Since $(f \overline{\circ} g)^{-1}(y) = \sup X_{l,y}$ and $\sup X_{r,y} = (f^{-1} \circ g^{-1})(y)$ this implies $(f \overline{\circ} g)^{-1}(y) \geq (f^{-1} \circ g^{-1})(y)$. Furthermore y is chosen arbitrarily, hence $(f \overline{\circ} g)^{-1} \geq (f^{-1} \circ g^{-1})$.

Combining both parts yields the desired equality $(f \overline{\circ} g)^{-1} = (f^{-1} \circ g^{-1})$. ◀

D Other Properties

► **Lemma 53.** Confer [15, Lemma 1]. Let Γ be a set of n independent tasks as described in Section 3.1 and let Assumption 20 hold, then the resource-based lower service curve β_i^- of task τ_i is

$$\beta_i^-(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \left\{ \beta^-(\lambda) - \sum_{j=1}^{i-1} \alpha_j^+(\lambda) \right\} \quad (50)$$

Proof. See [15, Lemma 1]. ◀

► **Lemma 54.** Confer [10, p. 154]¹. Let $f, g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions, then

$$f \leftrightarrow g = \inf_{\mu \in \mathbb{R}_0^+} \{ \mu : \forall \lambda \in \mathbb{R}_0^+, f(\lambda) \leq g(\lambda + \mu) \} \quad (51)$$

Proof. Let $A_\lambda := \{ \mu \in \mathbb{R}_0^+ : f(\lambda) \leq g(\lambda + \mu) \}$, $d(\lambda) := \inf A_\lambda$, and $B := \bigcap_{\lambda \in \mathbb{R}_0^+} A_\lambda$, then $f \leftrightarrow g = \sup_{\lambda \in \mathbb{R}_0^+} \{d(\lambda)\}$ and $\inf_{\mu \in \mathbb{R}_0^+} \{ \mu : \forall \lambda \in \mathbb{R}_0^+, f(\lambda) \leq g(\lambda + \mu) \} = \inf B$.

¹ Le Boudec and Thiran textually state Equation (51) in [10, p. 154] without proof and mistakenly refer to the vertical deviation [10, p. 154, (3.20)] and not the horizontal deviation [10, p. 154, (3.21)].

Case 1. The set A_λ is empty for some $\lambda \in \mathbb{R}_0^+$.

(a) $f \leftrightarrow g = \infty$: For that $\lambda \in \mathbb{R}_0^+$ we have $d(\lambda) = \inf A_\lambda = \inf \emptyset = \infty$, therefore $f \leftrightarrow g = \sup_{\lambda \in \mathbb{R}_0^+} \{d(\lambda)\} = \infty$.

(b) $\inf_{\mu \in \mathbb{R}_0^+} \{\mu : \forall \lambda \in \mathbb{R}_0^+, f(\lambda) \leq g(\lambda + \mu)\} = \infty$: Because there is a $\lambda \in \mathbb{R}_0^+$ where A_λ is the empty set, the set $B = \bigcap_{\lambda \in \mathbb{R}_0^+} A_\lambda = \emptyset$ is also empty. Therefore,

$$\inf_{\mu \in \mathbb{R}_0^+} \{\mu : \forall \lambda \in \mathbb{R}_0^+, f(\lambda) \leq g(\lambda + \mu)\} = \inf B = \inf \emptyset = \infty.$$

Case 2. The set A_λ is not empty for every $\lambda \in \mathbb{R}_0^+$.

(c) A_λ is an interval with $\sup A_\lambda = \infty$: Follows directly from g being increasing, i.e. let $\lambda \in \mathbb{R}_0^+$ and $\mu \in A_\lambda$, then for any $\xi \in \mathbb{R}_0^+$ such that $\xi \geq \mu$ we have $g(\lambda + \xi) \geq g(\lambda + \mu)$, therefore $\xi \in A_\lambda$ and $\sup A_\lambda = \infty$.

(d) B is an interval: Follows directly from (c), an intersection of intervals is an interval.

(e) $\mu \in B \Rightarrow f \leftrightarrow g \leq \mu$: $\mu \in B \Rightarrow \forall \lambda \in \mathbb{R}_0^+ : \mu \in A_\lambda \Rightarrow \forall \lambda \in \mathbb{R}_0^+ : d(\lambda) \leq \mu \Rightarrow \sup_{\lambda \in \mathbb{R}_0^+} \{d(\lambda)\} \leq \mu \Rightarrow f \leftrightarrow g \leq \mu$

(f) $\mu \notin B \Rightarrow \mu \leq f \leftrightarrow g$: $\mu \notin B \Rightarrow \exists \lambda \in \mathbb{R}_0^+ : \mu \notin A_\lambda \Rightarrow \exists \lambda \in \mathbb{R}_0^+ : \mu \leq d(\lambda) \leq \sup_{\lambda \in \mathbb{R}_0^+} \{d(\lambda)\} = f \leftrightarrow g$

Equation (51) follows from (d)–(f), i.e.

$$f \leftrightarrow g = \inf B = \inf_{\mu \in \mathbb{R}_0^+} \{\mu : \forall \lambda \in \mathbb{R}_0^+, f(\lambda) \leq g(\lambda + \mu)\} \quad \blacktriangleleft$$

► **Lemma 55.** Let $f, g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ be increasing functions, then the horizontal distance between them is

$$f \leftrightarrow g = (g \overline{\circ} f)^{-1}(0) \quad (52)$$

Proof. f and g are increasing, so we can use the equality of Equation (51). After some rearranging we apply the definition for the deconvolution in inf-plus, Equation (5b), and for the pseudo-inverse, Equation (4a).

$$\begin{aligned} f \leftrightarrow g &\stackrel{(51)}{=} \inf_{\mu \in \mathbb{R}_0^+} \{\mu : \forall \lambda \in \mathbb{R}_0^+, f(\lambda) \leq g(\lambda + \mu)\} \\ &= \inf_{\mu \in \mathbb{R}_0^+} \{\mu : \forall \lambda \in \mathbb{R}_0^+, 0 \leq g(\lambda + \mu) - f(\lambda)\} \\ &= \inf_{\mu \in \mathbb{R}_0^+} \left\{ \mu : 0 \leq \inf_{\lambda \in \mathbb{R}_0^+} \{g(\lambda + \mu) - f(\lambda)\} \right\} \\ &\stackrel{(5b)}{=} \inf_{\mu \in \mathbb{R}_0^+} \{\mu : 0 \leq (g \overline{\circ} f)(\mu)\} \stackrel{(4a)}{=} (g \overline{\circ} f)^{-1}(0) \quad \blacktriangleleft \end{aligned}$$

► **Lemma 56.** Confer [22, p. 286, Lemma 1]. Let $f : A \rightarrow A$ be an increasing function from and into a complete lattice A and let P be the set of fix-points of f , then P is not empty, P is a complete lattice and

$$\sup P = \sup_{x \in A} \{x : f(x) \geq x\} \in P \quad (53a)$$

$$\inf P = \inf_{x \in A} \{x : f(x) \leq x\} \in P \quad (53b)$$

Proof. See [22, p. 286, Lemma 1] ◀

General Framework for Routing, Scheduling and Formal Timing Analysis in Deterministic Time-Aware Networks

Anais Finzi 

TTTech Computertechnik AG, Wien, Austria

Ramon Serna Oliver 

TTTech Computertechnik AG, Wien, Austria

Abstract

In deterministic time-aware networks, such as TTEthernet (TTE) and Time Sensitive Networking (TSN), time-triggered (TT) communication are often routed and scheduled without taking into account other critical traffic such as Rate-Constrained (RC) traffic. Consequently, the impact of a static transmission schedule for TT traffic can prevent RC traffic from fulfilling their timing constraints.

In this paper, we present a general framework for routing, scheduling and formal timing analysis (FTA) in deterministic time-aware networks (e.g. TSN, TTE). The general framework drives an iterative execution of different modules (i.e. routing, scheduling and FTA) searching for a solution that fulfills an arbitrary number of defined constraints (e.g. maximum end-to-end RC and TT latency) and optimization goals (e.g. minimize reception jitter). The result is an iteratively improved solution including the routing configuration for TT and RC flows, the static TT schedule, a formal analysis for the RC traffic, as well as any additional outputs satisfying user constraints (e.g. maximum RC jitter). We then do a performance evaluation of the general framework, with a proposed implementation of the necessary modules for TTEthernet networks with mixed time-triggered and rate-constrained traffic. The evaluation of our studied realistic use case shows that, using the general framework, the end-to-end latency for RC traffic can be reduced up to 28.3%, and the number of flows not fulfilling their deadlines divided by up to 3 compared to existing methods.

2012 ACM Subject Classification Networks → Network performance evaluation

Keywords and phrases TSN, TTEthernet, AFDX, AVB, Modeling, Routing, Scheduling, Formal timing analysis, Worst-case analysis, Performance evaluation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.8

Funding This paper is part of the ADACORSA project that has received funding within the ECSEL JU in collaboration with the EU H2020 Programme and National Authorities, under grant agreement 876019.

1 Introduction

For many application domains, the temporal behavior of critical communication flows needs to be formally validated. For example, in aerospace the authorities require the proof of correctness as part of the certification process, as it also occurs in emerging industrial automation systems, with respect to critical traffic fulfilling end-to-end latency, jitter and available memory requirements. These proofs have been provided through analysis methods like Network Calculus [10, 8, 4] or the more recent Compositional performance evaluation [24], for technologies like Avionics Full DupleX (AFDX) [1], TTEthernet (TTE) [12, 20] or Time Sensitive Networking (TSN) [11].

Deterministic time-aware networks such as TSN and TTE enhance the event-triggered Rate Constrained traffic class (RC) with a fully synchronous time-triggered (TT) communication paradigm offering stringent guarantees, deterministic real-time temporal behavior, and



© Anais Finzi and Ramon Serna Oliver;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 8; pp. 8:1–8:23



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

composability. For the TT traffic class, determinism is ensured via an offline communication schedule enforcing the contention-free and timely delivery of critical frames across switched multi-hop networks within defined latency and jitter bounds.

The schedule synthesis for the TT traffic class is typically done either through heuristic-based approaches [21, 5] or optimal algorithms based on MiP or SMT solvers [19, 7, 3, 18].

The classical method considering routing, scheduling and FTA is to perform each of these steps sequentially. However, this means that manual intervention to correct non-optimal routing is often necessary, which can be very difficult. In particular, when routing is based on methods such as load balancing, where modifying one route can have repercussions on multiple flows. For this reason, approaches in which the steps computing the schedules and routes are coupled have been developed, using methods such as ILP [17, 6]. However, these approaches do not incorporate a formal timing analysis, so no alternative solutions can be easily explored if the RC constraints are not fulfilled.

To our knowledge, two previous works integrate the formal timing analysis of RC traffic when computing routes for both RC and TT traffic flows along with the schedule for TT traffic [9, 27]. However, both methods consider the end-to-end latency as the single user constraint. Additionally, we discuss in Section 2 several flaws leading to long computation times and inefficiencies that we address with our proposed method.

In this paper we provide a general framework for deterministic time-aware networks with mixed time-triggered and rate-constrained traffic classes, computing the static routing and scheduling configuration such as the constraints of both traffic classes are fulfilled. In our method, the routing, scheduling, and formal timing analysis modules can be implemented and customized by the user according to their own requirements. Moreover, additional RC or TT constraints (e.g. end-to-end latency, frame-memory limitations, etc...) can be incorporated to drive the search towards better solutions.

The main contributions of this paper are twofold, we propose: i) a general search for routing and scheduling which considers formal timing analysis of RC in Section 3; and ii) we conduct a performance evaluation with an application of the general framework for TTEthernet networks, including the time-triggered and rate-constrained traffic classes, in a realistic use case in Section 4 for which we developed a set of modules combining heuristic routing and SMT solver based scheduling with a formal RC analysis based on Network Calculus. To complement the analysis we compare our results with those of the current state-of-the-art in Section 4.4.

2 Related Work

Increasing the performance of time-triggered and event-triggered traffic has been pursued in previous works using methods to either improve the routes or the schedule instants of frames, by including event-triggered constraints into the scheduling problem formulation, or by applying a combination of these. To enhance TT traffic, the routing and scheduling can be done jointly using heuristics [25], or Integer Linear Programming (ILP) formulations [17, 6]. However, when using ILP, the complex worst-case timing analysis necessary to assess the RC constraints cannot be integrated into the ILP formulation as they are not linear.

Other approaches develop enhanced heuristics incorporating RC constraints guiding the search for a schedule. In [22, 7], the routing of flows is fixed and the computation of TT offsets is done considering the RC constraints of TTE. Results show that the schedulability of RC traffic can be significantly improved (e.g. doubled in [7]) with a heuristic search.

Finally, at the time of writing this section and to the best of our knowledge, there are two published approaches [9, 27] on integrating the formal timing analysis of RC traffic with the routing and scheduling.

In [9], the proposed method is developed specifically for a TSN network with scheduled 802.1Qbv (i.e. TT traffic) and AVB flows (i.e. RC traffic). However, we have identified a few limitations impacting the efficiency of the approach: the routing strategy is to re-route only TT flows, without trying to re-route RC flows. Besides limiting the solution space, this re-routing strategy is computationally expensive since a TT flow also requires re-scheduling. Additionally, the scheduling strategy is to re-schedule all TT flows at every new step, which again is a very time consuming operation.

In [27], a method to optimize the routing is proposed for TTE networks, including the TT and RC traffic classes. In their approach the authors use the RC end-to-end latencies when computing the routes for RC flows. However, the computation of the TT routes is done using load balancing independently from RC traffic properties. Moreover, when searching for a solution for the RC constraints, the TT routes are already fixed, which limits the solution space.

Additionally, we have noted that in both [9, 27], the RC constraints are limited to the end-to-end latency constraint, which does not map to typical industry requirements, usually including jitter and frame-memory restrictions (e.g. backlog).

In this paper, we introduce a generalized search framework that can be used for any deterministic time-aware network. We propose to explore the solution space by re-routing and re-scheduling one flow at a time, to avoid the expensive cost of re-scheduling all the flows after the initial schedule is computed. For our performance evaluation in Section 4, we present an implementation of all three functions for TTEthernet. In particular, our method can re-route any RC or TT flow, as well as re-schedule individual TT-flows, although it reduces as much as possible the re-scheduling operation, as it is the most expensive step. The search includes a formal analysis step including extensive RC constraints for our test-case, namely end-to-end latency, jitter, and memory occupancy.

3 General Framework description

The goal of our proposed general framework is to compute, within a configurable time interval, the best possible network configuration including routing, scheduling, and formal timing analysis (FTA), as well as optional parametrization and user-defined constraints.

The general framework implements a search algorithm leveraging the work of predefined functions for routing, scheduling, and FTA. Each of these three functions is encapsulated in a module, and can be adapted to implement existing or future solutions found in literature (e.g. for scheduling, heuristic solver [21], SMT solver [19, 7]). The interfaces of each module (i.e. set of non-optional input and outputs) are defined in Section 3.3.

We begin by describing the general search algorithm in Section 3.2. Then, the different modules are detailed in Sections 3.3 to 3.8.

3.1 Network and System Models

We define a general model wherein a network \mathcal{N} comprises a set \mathcal{V} of nodes and \mathcal{E} of links, and a set of \mathcal{F} of communication streams, or flows, with one sender (talker) and one or multiple receivers (listeners), and wherein $\mathcal{F}^{TT} \subset \mathcal{F}$ represents the subset of TT flows and $\mathcal{F}^{RC} \subset \mathcal{F}$ represents the subset of RC flows. The set \mathcal{C} represents the set of communication constraints,

like maximum end-to-end latency, jitter, or any other optional routing, scheduling, or shaping constraint. For convenience, we also define \mathcal{P} as the complete set of possible network routes between any of the nodes in \mathcal{N} and all of the other.

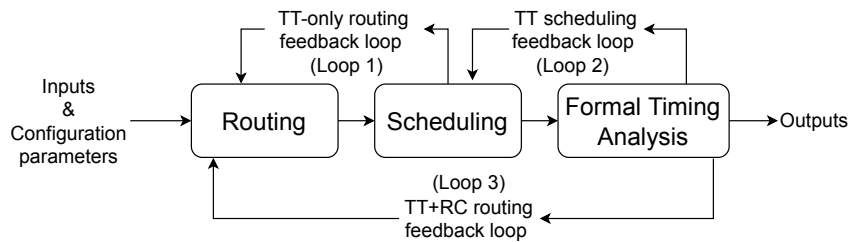
A flow $f \in \mathcal{F}$ is characterized by the tuple $\langle f.path, f.deadline \rangle$, wherein $path \in \mathcal{P}$ relates to the selected network route between the flow sender and receiver(s), and $deadline \in \mathbb{N}$ corresponds to a maximum end-to-end latency bound requirement. Note that, for the sake of simplicity, we omit to characterize in detail the implicit $1 : N$ relation of flows with its sender node and one, or multiple, receiver nodes. A TT flow $f \in \mathcal{F}^{TT}$ is further characterized by its time-triggered transmission instant¹, $f.offset \in \mathbb{N}$.

We also define \mathcal{O} as the set of solutions in the configuration space, wherein a solution $o \in \mathcal{O}$ represents a possible output of the general framework. The subsets $\mathcal{O}^{RC}, \mathcal{O}^{TT} \subset \mathcal{O}$ represent, respectively, the RC and TT solution space.

We further introduce $\mathcal{F}_s^1 : \langle f_{s_i}, i \in \mathbb{N} \rangle$, and respectively, $\mathcal{F}_s^3 : \langle f_{s_i}, i \in \mathbb{N} \rangle$, as sorted lists, or sequences, with index in the natural numbers, wherein each sequence is equivalent to the respective set, namely $set(\mathcal{F}_s^1) \equiv \mathcal{F}^{TT}$ and $set(\mathcal{F}_s^3) \equiv \mathcal{F}$. Note that the sort operation is described in Section 3.3.

3.2 General search algorithm

The main workflow behind the search algorithm, represented in Algorithm 1, consists of incrementally (re)routing or (re)scheduling one flow at a time following sorted lists maintained by the scheduling and FTA modules. Thanks to three feedback loops, depicted in Figure 1, one flow is identified in each iteration as a candidate to be (re)routed and/or (re)scheduled aiming at iteratively converging towards a better solution. After trying to find an initial solution (line 2), we start the search by ensuring all the TT traffic is routed and scheduled (Loop 1, line 5), as it is a necessary step before doing the formal timing analysis of the RC traffic. Secondly, if the RC traffic does not fulfill its constraints, we attempt to find an acceptable solution by keeping the same routing and only rescheduling TT flows one at the time (Loop 2, line 8). If this fails, then we attempt to reroute a flow (Loop 3, line 11). Hence, with these three feedback loops we explore both routing and scheduling alternatives extensively while limiting time expensive steps such as rescheduling all the TT flows at once.



■ **Figure 1** General framework workflow.

¹ Note that we characterize the output of the schedule operation applied to a TT Flow to comprise *offsets*, referring to the transmission instant of said TT Flow on each hop along its route. However, certain time-triggered networks may require additional information, like for example a priority queue assignment in the case of IEEE 802.1Qby with multiple TT queues (cf. [3]), or alternatively, a mapping to a GCL transmission window (cf. [18]). We claim that accounting for additional elements in the characterization of the schedule output is a trivial generalization and we remain with the simplified notation for the sake of clarity.

Algorithm 1 Main algorithm.

Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$

```

1: mem  $\leftarrow$  memories
2: initialize( $\mathcal{N}, \mathcal{F}, \text{mem}$ ) ▷ See Algorithm 2
3: repeat
4:   if  $\exists f \in \mathcal{F}^{TT}$ : sch.checkConstraints( $f$ ) = false then
5:     execute( $Loop_1$ ) ▷ Algorithm 3 in Section 3.5
6:   end if
7:   if  $\exists f \in \mathcal{F}^{RC}$ : fta.checkConstraints( $f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$ ) = false then ▷ Formal timing
   analysis ▷ Algorithm 4 in Section 3.6
8:     execute( $Loop_2$ )
9:   end if
10:  if  $\exists f \in \mathcal{F}^{RC}$ : fta.checkConstraints( $f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$ ) = false then
11:    execute( $Loop_3$ ) ▷ Algorithm 5 in Section 3.7
12:  end if
13: until  $\forall f \in \mathcal{F}^{RC}$ : fta.checkConstraints( $f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$ ) = true  $\wedge$ 
14:     $\forall f \in \mathcal{F}^{TT}$ : sch.checkConstraints( $f$ ) = true

```

▼ **Inputs:** The main algorithm takes the typical inputs when computing network routing and scheduling, i.e. the specification of network topology, \mathcal{N} , and flows, \mathcal{F} , as well as optional timing and routing constraints, \mathcal{C} .

▲ **Outputs:** The output of the algorithm consist of the typical outputs generated by network routing and scheduling algorithms, namely a route for each RC and TT flows as well as a TT flow schedule. Additional shaping or scheduling parameters (e.g. AVB reserved bandwidth, WRR weights if TTE is extended to include these) as well as custom metrics (e.g. minimum end-to-end deadlines, minimum memory configurations) can be optionally added to the output as required. In the particular case of RC shaping and scheduling parameters, these are optionally computed in the FTA module, in Algorithms 1 and 4.

As the solution space can be very extensive, we limit the search algorithm in three dimensions: first, we limit the number of routes that will be tested for each flow. Secondly, we limit the number of iterations performed when computing a new schedule for a particular set of paths. Thirdly, we limit the number of flows that may be selected before re-sorting the list of flows, i.e. rearranging the selection order of those flows. Therefore, we define the following configuration parameters, directing the solution space exploration:

▼ **Configuration parameters:**

- **conf.maxExploredPaths:** maximum number of paths that may be explored for each flow in both routing feedback loops (see Section 3.8.2).
- **conf.maxSchedIterations:** maximum number of iterations in a TT scheduling feedback loop (see Section 3.6);
- **conf.maxExploredFlowReset:** maximum number of flows from the sorted list of flows that can be explored before re-sorting the list (see Section 3.8.1).

These parameters expose different trade-offs enabling the customization of the search to the specifications of given use cases. For instance, limiting the number of routes per flows, **conf.maxExploredPaths**, allows testing more flows within a reasonable amount of time. Limiting the number of schedule search iterations, **conf.maxSchedIterations**, allows testing more routes for each flow. However, the down-side of these limitations is that an optimal solution may be missed or the search may remain within a local optima.

3.3 Module requirements

There are three main functions to be fulfilled for our general framework: routing, scheduling and formal timing analysis. Each function is decoupled from the other two as an independent module, with a defined interface between them. We define the scope of each module with an enumeration of requirements.

The framework allows to leverage existing methods for the implementation of each of its modules. We describe here the requirements for the general case and detail an implementation in Section 4, which is later used in the evaluation in Section 4.2.

- **Routing (rt)**, manages functions related to finding network routes for RC and TT flows. The module shall provide:
 - `findRoute($f \in \mathcal{F}, \mathcal{N}, \mathcal{C}$)`: finds an initial path in \mathcal{N} for flow f , subject to constraints \mathcal{C} ;
 - `allPaths($f \in \mathcal{F}, \mathcal{N}, \mathcal{C}$)` a list of possible paths in \mathcal{N} for flow f , subject to constraints \mathcal{C} .
- **Scheduling (sch)**, manages functions related to TT traffic. It shall provide:
 - `schedule(f, \mathcal{C}) $\rightarrow \mathcal{O}^{TT}$` attempts to schedule TT flow $f \in \mathcal{F}^{TT}$, subject to constraints \mathcal{C} ;
 - `checkConstraints($f \in \mathcal{F}^{TT}$) $\rightarrow \text{boolean}$` evaluates whether flow $f \in \mathcal{F}^{TT}$ has been successfully scheduled (i.e. has transmission offset(s));
 - `sortFlows(1, \mathcal{F}^{TT})` sort operation over the set \mathcal{F}^{TT} of TT flows, for Loop 1, used to prioritize the flows and guide the search toward a better solution;
 - `sortPaths($f \in \mathcal{F}$)`: sorted list of paths of TT flow f for Loop 1, used to prioritize the paths to guide the search toward a better solution, optionally supported by scheduling information;
 - `costFunction($\mathcal{O}^{TT} \rightarrow \mathbb{R}$)`: cost function to assess a partial solution or save the best solution, in Loop 1.
 - `output $\in \mathcal{O}^{TT}$` : module output, including a TT schedule.
- **Formal timing analysis (fta)**, manages network analysis, related to either both RC and TT flows, or only RC flows. It shall provide:
 - `checkConstraints($f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$) $\rightarrow \text{boolean}$` : evaluates whether the RC flow $f \in \mathcal{F}^{RC}$ fulfills the constraints in \mathcal{C} ;
 - `impossibilityTest($\mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$) $\rightarrow \text{boolean}$` necessary test for constraints in \mathcal{C} being fulfilled by flows \mathcal{F}^{RC} in \mathcal{N} for any flow path and/or TT offset (e.g. the maximum end-to-end latency constraint required is below the minimum possible end-to-end latency on the shortest path);
 - `feasibilityTest($\mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$) $\rightarrow \text{boolean}$` evaluates whether the constraints in \mathcal{C} can possibly be fulfilled for the flows \mathcal{F}^{RC} in \mathcal{N} with the current flow paths;
 - `portImpact($f \in \mathcal{F}$)`: evaluates if the path flow f has any port in common with the paths of RC flows which are not yet fulfilling their constraints;
 - `sortFlows(3, \mathcal{F})`: sort operation over the set \mathcal{F} , for Loop 3;
 - `sortFlows(2, \mathcal{F}^{TT})`: sort operation over the set \mathcal{F}^{TT} ;
 - `sortPaths($f \in \mathcal{F}$)`: sorted list of paths of flow f , for Loop 3;
 - `costFunction(\mathcal{O}) $\rightarrow \mathbb{R}$` : cost function to assess a partial solution or save the best solution, in Loop 3.
 - `output $\in \mathcal{O}^{RC}$` : module output, including the parameters for RC shaping/scheduling (e.g. WRR weights, AVB bandwidth reservation) and output requirements (e.g. necessary memory reservation)

3.4 Search initialization

■ **Algorithm 2** Search Initialization algorithm.

Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}, \text{mem}$

- 1: **for all** $f \in \mathcal{F}$ **do** ▷ Attempts to route all flows
- 2: $\text{rt.findRoute}(f, \mathcal{N}, \mathcal{C})$
- 3: **end for**
- 4: **for all** $f \in \mathcal{F}^{TT}$ **do** ▷ Attempts to schedule all TT flows
- 5: $\text{sch.schedule}(f, \mathcal{N}, \mathcal{C})$
- 6: **end for**
- 7: **if** $\forall f \in \mathcal{F}^{TT} : \text{sch.checkConstraints}(f) = \text{true}$ **then**
- 8: $\text{mem.savedOffsets}[f] \leftarrow f.\text{offset}, f.\text{path}$ ▷ Save current paths and offsets
- 9: **end if** ▷ Initialize search memories
- 10: **for all** $f \in \mathcal{F}$ **do**
- 11: $\text{mem.exploredPath_3}[f] \leftarrow f.\text{path}$
- 12: $\text{mem.defaultPaths}[f] \leftarrow f.\text{path}$
- 13: $\text{mem.exploredPathSets} \leftarrow \text{mem.exploredPathSets} \cup \langle f, f.\text{path} \rangle$
- 14: **end for**
- 15: $\text{mem.currentFlow_3} \leftarrow \text{NULL}$
- 16: $\text{mem}.\mathcal{F}_s^1 \leftarrow \emptyset$
- 17: $\text{mem}.\mathcal{F}_s^3 \leftarrow \emptyset$
- 18: $\text{mem.allTTPathsExplored} = \text{false}$

In a first step, the search computes an initial routing and scheduling solution, followed by the initialization of the memories necessary to keep track of the progress, as defined in Algorithm 2. First, all the routes are computed (line 2). Note that there is an implicit failure termination if an initial route cannot be found for each of the flows, meaning that the set of flows is not feasible with the given topology, and hence causing the search to abort with failure. Secondly, the algorithm attempts to schedule TT flows (line 5). Afterward, the memories (`mem`) used in the global search are initialized, namely:

- $\text{mem.savedOffsets}[f \in \mathcal{F}^{TT}] \leftarrow \{\langle p_1, O_1 \rangle \dots \langle p_n, O_n \rangle : f_i \in \mathcal{F}^{TT}, p_j \in \mathcal{P}, O_i = \{o_i^0 \dots o_i^k\}, o_i^j \in \mathbb{N}\}$: stores the latest successfully scheduled set of TT paths and their offsets. Note that offsets are represented as a set of values corresponding to the transmission offset on each port of a multicast route; `mem.savedOffsets` is used as a restore point to a previous state in which TT Flows were both routed and scheduled, before continuing the search. The use of `mem.savedOffsets` will be detailed in Loops 1 and 2.
- $\text{mem.exploredPathSets} \leftarrow \{S_0 \dots S_n\} : S_i = \{\langle f_0^i, p_0^i \rangle \dots \langle f_m^i, p_m^i \rangle : f_j^i \in \mathcal{F}, p_j^i \in \mathcal{P}\}, i \in \mathbb{N}$: storing the set of explored path sets (each flow in a set is associated to one path), shared by the routing feedback loops. This is mainly used to determine whether all solution have been explored;
- $\text{mem.defaultPaths}[f \in \mathcal{F}] \leftarrow \{p_0 \dots p_n : p_i \in \mathcal{P}\}$: storing the so-called *default paths* previously used by the routing searches. The definition and use of the default paths is detailed in Section 3.5.
- $\text{mem.exploredPaths_3}[f \in \mathcal{F}] \leftarrow \{p_0 \dots p_n : p_i \in \mathcal{P}\}$: storing the sets of explored paths for each flow (each flow is associated to a list of explored paths) for Loop 3.
- $\text{mem.currentFlow_3} \leftarrow f \in \mathcal{F}^{TT}$: stores the flows currently being re-routed in Loop 3. It is used to select a new flow to reroute.

- $\text{mem}.\mathcal{F}_s^1$: stores a sorted list of the elements of \mathcal{F}^{TT} , wherein $\mathcal{F}_{s_k}^{TT}$ is the $k + 1^{\text{th}}$ element in the list.
- $\text{mem}.\mathcal{F}_s^3$: stores a sorted list of the elements of \mathcal{F} , wherein \mathcal{F}_{s_k} is the $k + 1^{\text{th}}$ element in the list.
- $\text{mem}.\text{allTTPathsExplored} \leftarrow (\text{boolean})$ flags when all the TT paths have been explored;

Once the initialization is performed the search of a solution begins as described in the following sections.

3.5 Loop 1: TT-only routing feedback loop

Algorithm 3 (Loop 1) tries to re-route and schedule TT flows if not all TT flows were scheduled, either following the initial routing or a re-routing step from Loop 3. The goal is to find a solution in which all TT flows are routed and scheduled, regardless of the RC traffic.

■ **Algorithm 3** Loop 1: TT-only routing feedback loop.

Require: $\mathcal{N}, \mathcal{F}^{TT}, \mathcal{C}, \text{mem}$

```

1: for all  $f_k \in \mathcal{F}^{TT}$  do
2:    $\text{mem}.\text{exploredPaths}_1[f_k] \leftarrow \{f_k.\text{path}\}$            ▷ Initialize with current paths
3: end for
4:  $\text{mem}.\text{currentFlow}_1 \leftarrow \text{NULL}$ 
5:  $\text{mem}.\mathcal{F}_s^1 \leftarrow \text{sch}.\text{sortFlows}(1, \mathcal{F}^{TT})$            ▷ Sort the list of TT flows
6: while  $\exists f_k \in \mathcal{F}^{TT} : \text{sch}.\text{checkConstraints}(f_k) = \text{false} \wedge \text{mem}.\text{allTTPathsExplored} = \text{false}$  do
7:    $f' \leftarrow \text{selectFlowPath}(1, \text{mem}.\text{exploredPaths}_1)$    ▷ See Algorithm 7
8:    $\text{schedule}(f, \mathcal{N}, \mathcal{C})$ 
9:    $\text{mem}.\text{currentFlow}_1 \leftarrow f'$ 
10:  if  $\text{sch}.\text{costFunction}(\text{output}) < \text{sch}.\text{costFunction}(\text{mem}.\text{bestOutput})$  then
11:     $\text{mem}.\text{bestOutput} \leftarrow \text{current output}$ 
12:  end if
13:  if  $\forall f_i \in \mathcal{F}^{TT} : \text{sch}.\text{checkConstraints}(f_i) = \text{true}$  then
14:    for all  $f_j \in \mathcal{F}^{TT}$  do                                 ▷ Save paths and offsets
15:       $\text{mem}.\text{savedOffsets}[f_j] \leftarrow f.\text{offset}, f.\text{path}$ 
16:    end for
17:  end if
18:   $\text{updateMemories}(1, f', \text{mem})$                            ▷ See Algorithm 6
19: end while
20: if  $\exists f_k \in \mathcal{F}^{TT} : \text{sch}.\text{checkConstraints}(f_k) = \text{false}$  then
21:  if  $\text{mem}.\text{savedOffsets} \neq \emptyset$  then
22:    for all  $f_j \in \mathcal{F}^{TT}$  do                                 ▷ Reset paths and offsets
23:       $f_j.\text{offset} \leftarrow \text{mem}.\text{savedOffsets}[f_j].\text{offset}$ 
24:       $f_j.\text{path} \leftarrow \text{mem}.\text{savedOffsets}[f_j].\text{path}$ 
25:    end for
26:  else
27:     $\text{exitPartialSolution}(\text{mem}.\text{bestOutput})$                ▷ Or fail without output
28:  end if
29: end if

```

The search starts from an initial set of paths, called *default path* (stored in $\text{mem}.\text{defaultPaths}$) and a flow selected to be rerouted. If no schedule is found with any of its different possible paths, the selected flow is set back to the default path, and another

flow is chosen. Restoring the path to the already explored default path adds stability to the search process. Algorithm 6 details how new default paths can be selected in order to test the different permutations.

Algorithm 3 (Loop 1) uses three memories:

- `mem.exploredPaths_1` [$f \in \mathcal{F}^{TT}$] $\leftarrow \{p_0..p_n : p_i \in \mathcal{P}\}$: storing the sets of explored paths for each TT flow (each flow is associated to a list of explored paths) for Loop 1;
- `mem.currentFlow_1` $\leftarrow f \in \mathcal{F}^{TT}$: stores the flows currently being rerouted in Loop 1;
- `mem.bestOutput` $\leftarrow o \in \mathcal{O}$: stores the best solution found so far. This includes all the outputs listed in Section 3.2 which are already available in the current state of the search.

The algorithm begins initializing `exploredPaths_1` with the current paths (lines 1 to 3), and then sorting the set of flows (line 5). Following, a search for a feasible solution is initiated, until either all TT flows are scheduled or all path permutations have been explored (line 6). Within the search, a flow is selected using Algorithm 7, then rerouted, and attempted to be scheduled (lines 7 and 8), following the update of `mem.currentFlow_1` and `mem.bestOutput` (lines 9 to 12).

If a schedule has been found for all TT flows, the paths and offsets are stored in `mem.savedOffsets` (lines 13 to 17), which if needed, can be used to restore a solution with feasible TT traffic offsets (see lines 22 to 25). This is necessary when all available TT path combinations have been tested and Loop 1 finishes, but there remain still untested RC paths that may be explored via Loop 3. Note that the search algorithm may fail and exit in Loop 1, either with a partial or no solution at all, if, directly after the initialization, no valid TT schedule has been found after having explored all paths (line 27).

Finally, after saving the paths and offsets, `mem.exploredPathsSets` and `mem.defaultPaths` are updated as described in Algorithm 6 (line 18).

3.6 Loop 2: TT scheduling feedback loop

Algorithm 4 (Loop 2), identifies, supported by the FTA module, the TT flow with a higher impact on RC traffic (line 17), which is then re-scheduled (line 18) with the aim of finding a solution improving RC traffic performance.

Algorithm 4 uses two memories:

- `mem.exploredOffsets` [$f \in \mathcal{F}^{TT}$] $\leftarrow \{O_0..O_n : O_i = \{o_i^0..o_i^k\}, o_i^j \in \mathbb{N}\}$: stores the sets of explored offsets for each flow (each flow is associated to a list of explored offsets). Note that offsets are represented as a set of values corresponding to the transmission offset on each port of a multicast route;
- `mem.diversification` $\leftarrow \{f_0..f_n : f_i \in \mathcal{F}\}$: tracks the flows already selected for diversification purposes, allowing to select alternative flows and explore different part of the solution space. Hence, avoiding iterations over stable regions of the solution space by always choosing the same flows.

The search ends when a feasible solution is found or when either the maximum number of iterations, defined in the configuration parameter `conf.maxSchedIterations` (see Section 3.2), is reached, or else when the diversification memory contains all flows (line 16), meaning that no other flow is left to be selected (lines 24 to 28).

Note that Algorithm 4 is a generalization of Algorithm 1 in [7], so we only detail here the main improvements, namely

1. allowing an arbitrary number of constraints \mathcal{C} , including the end-to-end latency, as well as the possibility of storing the best solution found so far at any given time (cf. lines 7 and 33);

■ **Algorithm 4** Loop 2: TT scheduling feedback loop.

Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}, \text{mem}, \text{conf}, \text{currentOutput}$

```

1: it_loop2  $\leftarrow$  0
2: mem.exploredOffsets, mem.diversification  $\leftarrow$   $\emptyset$ 
3: if mem. $\mathcal{F}_s^1 = \emptyset$  then
4:   mem. $\mathcal{F}_s^1 \leftarrow$  fta.sortFlows(2,  $\mathcal{F}^{TT}$ )
5: end if
6: if costFunction(currentOutput) < costFunction(mem.bestOutput) then
7:   mem.bestOutput  $\leftarrow$  currentOutput
8: end if
9: fullfilled  $\leftarrow$   $\forall f \in \mathcal{F}^{RC} : \text{fta.checkConstraints}(f, \mathcal{F}, \mathcal{C})$   $\triangleright$  Run the FTA Analysis
10: if fullfilled = false  $\wedge$  fta.impossibilityTest( $\mathcal{N}, \mathcal{F}^{RC}, \mathcal{C}$ ) = true then
11:   exit
12: else if fullfilled = false  $\wedge$  fta.feasibilityTest( $\mathcal{N}, \mathcal{F}^{RC}, \mathcal{C}$ ) = true then
13:   for all  $f_k \in \mathcal{F}^{TT}$  do  $\triangleright$  Reset offsets
14:      $f_k.offset \leftarrow$  mem.savedOffsets[ $f_k$ ].offset
15:   end for
16:   while  $\exists f_j \in \mathcal{F}^{RC} : \text{fta.checkConstraints}(f_j, \mathcal{N}, \mathcal{F}^{RC}, \mathcal{C}) = \textit{false} \wedge$   

|mem.diversification| < | $\mathcal{F}^{TT}$ |  $\wedge$  it_loop2 < conf.maxSchedIterations do
17:      $f' \leftarrow$  mem. $\mathcal{F}_{s_0}^{TT}$   $\triangleright$  Select flow impacting most RC
18:     sch.schedule( $f', \mathcal{N}, \mathcal{C}$ )
19:     if  $f'.offset \notin$  mem.exploredOffsets[ $f'$ ] then
20:       mem.diversification  $\leftarrow$   $\emptyset$ 
21:     end if
22:     while  $f'.offset \in$  mem.exploredOffsets[ $f'$ ]  $\wedge$  |mem.diversification| < | $\mathcal{F}^{TT}$ |  

do
23:       mem.diversification  $\leftarrow$  mem.diversification  $\cup f'$ 
24:        $k \leftarrow$  0
25:       repeat  $\triangleright$  Select first flow not in mem.diversification
26:          $f' \leftarrow$  mem. $\mathcal{F}_{s_k}^{TT}$ 
27:          $k \leftarrow k + 1$ 
28:       until  $f' \notin$  mem.diversification
29:       sch.schedule( $f', \mathcal{N}, \mathcal{C}$ )
30:     end while
31:      $\forall f \in \mathcal{F}^{RC} : \text{fta.checkConstraints}(f, \mathcal{F}, \mathcal{C})$ 
32:     if costFunction(currentOutput) < costFunction(mem.bestOutput) then
33:       mem.bestOutput  $\leftarrow$  currentOutput
34:     end if
35:     for all  $f_k \in \mathcal{F}^{TT}$  do  $\triangleright$  Update explored offsets
36:       mem.exploredOffsets[ $f_k$ ]  $\leftarrow$  mem.exploredOffsets[ $f_k$ ]  $\cup f_k.offset$ 
37:     end for
38:     it_loop2  $\leftarrow$  it_loop2 + 1
39:   end while
40: end if

```

2. the addition of an impossibility check (cf. line 10) as well as a feasibility check (cf. line 12) to avoid searching for solutions when none exists;
3. before a flow is re-scheduled (line 18), all other TT flow offsets are reset to the values stored in `mem.savedOffsets` (lines 13 to 15), already presented in Loop 1. We found that the re-scheduling in Loop 2 can lead to a stable but non-optimal area of the solution space. By restoring the state stored before Loop 2 after re-routing a flow, it is more likely to avoid this area and, hence, find solutions otherwise inaccessible.

3.7 Loop 3: RC+TT routing feedback loop

Finally, the third feedback loop, represented in Algorithm 5, uses the same principle as Loop 1. Namely, if no feasible solution is found after all paths of a specific flow have been tested, the flow is set back to the default path and a new search iteration begins (see line 3 in Algorithm 1). In the case of Loop 3, the selected flow can be either a TT or an RC flow, which enables testing a large array of solutions while trying to prioritize the more likely to succeed first.

■ **Algorithm 5** Loop 3: TT+RC routing feedback loop.

Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}, \text{mem}, \text{conf}$

- 1: **if** `mem. $\mathcal{F}_s^3 = \emptyset$` **then**
- 2: `mem. $\mathcal{F}_s^3 \leftarrow \text{fta.sortFlows}(3, \mathcal{F})$`
- 3: **end if**
- 4: `$f' = \text{selectFlowPath}(3, \text{mem.exploredPaths}_3)$` ▷ See Algorithm 7
- 5: `$\text{mem.currentFlow}_3 \leftarrow f'$`
- 6: **if** `$f' \in \mathcal{F}^{TT}$` **then**
- 7: **for all** `$f_k \in \mathcal{F}^{TT}$` **do** ▷ Reset offsets
- 8: `$f_k.offset \leftarrow \text{mem.savedOffsets}[f_k].offset$`
- 9: **end for**
- 10: `$\text{sch.schedule}(f', \mathcal{N}, \mathcal{C})$`
- 11: **if** `$\forall f_j \in \mathcal{F}^{TT} : \text{sch.checkConstraints}(f_j) = \text{true}$` **then**
- 12: `$\text{mem.savedOffsets}[f_j] \leftarrow f_j.offset, f_j.path, \forall f_j \in \mathcal{F}^{TT}$` ▷ Save path and offsets
- 13: **end if**
- 14: **end if**
- 15: `$\text{updateMemories}(3, f', \text{mem})$` ▷ See Algorithm 6

Algorithm 5 (Loop 3) begins sorting the flows if they have not been sorted yet (lines 1 to 3). Then a new flow is selected and rerouted (line 4), followed by the update of `mem.currentFlow3` in line 5. If the selected flow is TT, it must then be rescheduled. As Loop 3 follows Loop 2, the offsets are restored to the saved values to avoid stable but non-optimal solution space areas (lines 7 to 9), similar to Subsection 3.6. Next the new offsets are computed for the selected flow (line 10) and if successful, they are stored in `mem.savedOffsets` (line 12). If the TT flow cannot be rescheduled, then Loop 1 follow (see line 4 in Algorithm 1).

3.8 Common support algorithms

In this section, we describe two algorithms supporting both Loop 1 and Loop 3. Algorithm 6 updates the memories and manage the default paths, while Algorithm 7 implements the selection of a new flows and paths.

3.8.1 Update of memories

The goal of Algorithm 6 is to update the memories tracking the progress of the search, such as the set of explored paths `mem.exploredPaths_i` and `mem.exploredPathSets`.

■ **Algorithm 6** `updateMemories(i)`.

```

Require:  $\mathcal{N}, \mathcal{F}, \mathcal{C}, \text{mem}, \text{conf}, \text{mod} \in \{\text{sch}, \text{fta}\}, i \in \{1, 3\}$   $\triangleright$  Current loop index (1,3)
1:  $f_c = \text{mem.currentFlow}_i$   $\triangleright$  Current flow in Loop  $i, i \in \{1, 3\}$ 
2:  $\text{mem.exploredPaths}_i[f_c] \leftarrow f_c.path$   $\triangleright$  Update path selected flow
3: if  $(\bigcup \langle f_k, f_k.path \rangle : f_k \in \text{mem}.\mathcal{F}_s^i) \notin \text{mem.exploredPathSets}$  then
4:    $\text{mem.exploredPathSets} \leftarrow \text{mem.exploredPathSets} \cup \langle f_i, f_i.path \rangle : \forall f_i \in \mathcal{F}$ 
5:   if  $|\text{mem.exploredPaths}_i| \geq \text{conf.maxExploredFlowReset} \vee [i = 3 \wedge$ 
    $\text{fta.portImpact}(f_c.path) = \text{false}]$  then
6:      $\text{mem.exploredPaths}_i[f_c] \leftarrow \emptyset$ 
7:      $\text{mem}.\mathcal{F}_s^i \leftarrow \text{mod.sortFlows}(i, \mathcal{F}^i)$ 
8:   end if
9: else if  $\forall f_p \forall p_q : f_p \in \text{mem}.\mathcal{F}_s^i, p_q \in \text{rt.allPaths}(f_p, \mathcal{N}, \mathcal{C}), p_q \in \text{mem.exploredPath}_i$ 
   then  $\triangleright$  All flow paths tested for current default paths
10:  if  $i = 1 \wedge \text{savedOffsets} \neq \emptyset \wedge \forall f_p \forall p_q : f_p \in \text{mem}.\mathcal{F}_s^i, p_q \in \text{rt.allPaths}(f_p, \mathcal{N}, \mathcal{C}), p_q \in$ 
    $\text{mem.defaultPaths}$  then
11:    $\triangleright$  All flow paths tested as default paths
12:    $\text{mem.allTTPathsExplored} \leftarrow \text{true}$ 
13:   for all  $f_l \in \mathcal{F}^{TT}$  do  $\triangleright$  Reset paths and offsets
14:      $f_l.path = \text{savedOffsets}[f_l].path$ 
15:      $f_l.offset = \text{savedOffsets}[f_l].offset$ 
16:   end for
17: else
18:    $f'.path = \text{selectFlowPath}(i, \text{mem.usedDefaultPaths})$   $\triangleright$  Algorithm 7
19:   if  $f' \in \mathcal{F}^{TT}$  then
20:     for all  $f_l \in \mathcal{F}^{TT}$  do  $\triangleright$  Reset offset
21:        $f_l.offset \leftarrow \text{mem.savedOffsets}[f_l]$ 
22:     end for
23:      $\text{sch.schedule}(f', \mathcal{C})$ 
24:     if  $\forall f \in \mathcal{F}^{TT} : \text{sch.checkConstraints}(f) = \text{true}$  then
25:       for all  $f_l \in \mathcal{F}^{TT}$  do  $\triangleright$  Save path and offsets
26:          $\text{mem.savedOffsets}[f_l] \leftarrow f_l.offset, f_l.path$ 
27:       end for
28:     end if
29:   end if  $\triangleright$  Update default path
30:    $\text{mem.defaultPaths}[f'].path \leftarrow \text{mem.defaultPaths}[f'].path \cup f'.path$ 
31:    $\text{mem.exploredPaths}_i \leftarrow \emptyset$ 
32:    $\text{mem}.\mathcal{F}_s^i \leftarrow \text{mod.sortFlows}(i, \mathcal{F}^i)$ 
33: end if
34: else if  $f_c = \text{mem}.\mathcal{F}_{s_k}^i : k = |\text{mem}.\mathcal{F}_s^i| - 1$  then  $\triangleright$  All flows of the current set were tested
35:    $\text{mem.exploredPaths}_i \leftarrow \emptyset$ 
36:    $\text{mem}.\mathcal{F}_s^i \leftarrow \text{mod.sortFlows}(i, \text{mem}.\mathcal{F}^i)$ 
37: end if

```

For Loop 1 and 3 we use the *default path* memory to explore around a stable set of paths, i.e. the default paths, and only after that exploration is concluded the set of default paths is updated and a new exploration around the new stable set of paths begins. With this, we avoid the excessive time it would take to explore all the solutions around the default paths, when it is likely that not all of them lead to feasible solutions.

A first part of our strategy to guide the search toward a better part of the solution space is to sort the flows, as already explained. However, this is not sufficient due to the fact that when a new path is found, or the default paths change, it has a global impact on the totality of flows. This leads to the need to regularly re-sort the flows so as to continue testing those with the highest impact on the flows causing more trouble to the algorithm. It is important to note that always re-sorting is also not a good approach, as it can lead to selecting always the same flows and not making progress.

In Section 3.2 we define `conf.maxExploredFlowReset`, which lets the user configure after how many tested flows the memories `mem.exploredPaths_i` are reset and the flows resorted. This parameter can be tailored based on the characteristics of the network.

The algorithm begins setting the current flow (line 1) and updating `mem.exploredPaths_i` (line 2). Then, if the current set of path is unknown to `mem.exploredPathSets` it is added (line 3). If enough flows have been explored or if, in Loop 3, the path of the current flow has no impact on the flows not fulfilling their constraints, then `mem.exploredPaths_i` is reset and the flows resorted (lines 6 and 7).

However, if the path set has already been explored and all combinations of flow paths have been testes for the current default paths (line 9), it is checked if, in Loop 1, all the TT path combinations have been tested (lines 10). If that is the case and there are saved offsets in `mem.savedOffsets`, then `mem.allTTPathsExplored` is set to *true* and the paths and offsets are resets to the saved values (lines 13 to 16). If that is not the case, then a new default path is selected (line 18). If it consists of a TT flow, its offsets are reset to the saved values, if possible (line 20) and the flow is rescheduled (line 23). If all TT flows are scheduled, then the paths and offsets are saved in `mem.savedOffsets` (line 25). Following, the memory `mem.defaultPaths` is updated with the new default path (line 30), and the flows are resorted to select the more promising flows around the newly chosen default paths. Finally, if the selected flow was the last on the list (line 34), `mem.exploredPaths_i` is updated and the flows again resorted (lines 35 and 36) in preparation for the next iteration.

3.8.2 Selection of a flow and path

Algorithm 7 is used to select a flow and its new path within Loops 1 and 3. The algorithm selects the first untested path of the current flow f_c in the sorted list of flows (line 25).

Additionally, we define `sortPaths(f_c , conf.maxExploredPaths)` instantiating the functions `sortPaths(f_c)`, respectively from the FTA module in Loop 1, or the Scheduling module in Loop 3, and selecting the first `conf.maxExploredPaths` items of the provided sorted list.

The algorithm tries setting the current flow (line 1) or, if none (line 2), selects the first flow from the sorted list (line 3). If the maximum configured number of flows (i.e. `conf.maxExploredPaths`) has been reached (line 4), the flow path is reset to the latest default path (lines 5). If the flow happens to be TT, the offsets are resets, if possible, (lines 7 to 9) and the flow is reschedule (line 10). Upon success, the paths and offsets are saved (lines 12 to 14).

The next flow in the sorted list is selected as the new current flow (line 17) and the first path not in memory is selected to reroute the flow (lines 19 to 24).

■ **Algorithm 7** SelectFlowPath(i): Flow and Path selection.

Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$, conf, $i \in \{1, 3\}$, mem

```

1:  $f_c = \text{mem.currentFlow}_i$  ▷ Current flow for Loop i,  $i \in \{1, 3\}$ 
2: if  $f_c = \text{NULL}$  then
3:    $f_c \leftarrow \text{mem.F}_{s_0}^i$  ▷ First flow in the sorted list
4: else if  $|\text{memory}[f_c]| = \text{conf.maxExploredPaths}$  then
5:    $f_c.path \leftarrow \text{mem.defaultPaths}[f_c].path[k]$ :  $k = |\text{mem.defaultPaths}[f_c]| - 1$ 
6:   if  $f_c \in \mathcal{F}^{TT}$  then
7:     for all  $f_n \in \mathcal{F}^{TT}$  do
8:        $f_n.offset \leftarrow \text{mem.savedOffsets}[f_n]$  ▷ Reset offset
9:     end for
10:    sch.schedule( $f_c, \mathcal{N}, \mathcal{C}$ )
11:    if  $\forall f_k \in \mathcal{F}^{TT} : \text{sch.checkConstraints}(f_k) = \text{true}$  then
12:      for all  $f_n \in \mathcal{F}^{TT}$  do
13:         $\text{mem.savedOffsets}[f_n] \leftarrow f_n.offset, f_n.path$  ▷ Save offset and path
14:      end for
15:    end if
16:  end if
17:   $f_c \leftarrow \text{mem.F}_{s_{k+1}}^i : \text{mem.F}_{s_k}^i = f_c$  ▷ Next flow in the sorted list
18: end if
19:  $P^s \leftarrow \text{sortPaths}(f_c, \text{conf.maxExploredPaths})$ 
20:  $j \leftarrow 0$ 
21: while  $P_j^s \in \text{memory}[f_c]$  do
22:    $j \leftarrow j + 1$ 
23: end while
24:  $f_c.path \leftarrow P_j^s$  ▷ Set first sorted path not in  $\text{memory}[f_c]$ 
25: return  $f_c$ 

```

Note that the Algorithm 7 is instantiated with the inputs $i \in \{1, 3\}$ and $\text{memory} = \text{mem.exploredPaths}_i$ (e.g. line 4 in Algorithm 5) to select the *new current flow*, or with the inputs $i \in \{1, 3\}$ and $\text{memory} = \text{mem.defaultPaths}$ (see line 18 in Algorithm 6) to select a *new default path*.

4 Performance evaluation

We have so far presented a general search framework for routing, scheduling and formal timing analysis. In this section, we present a performance evaluation with an implementation of the framework for TTEthernet, following a detailed industrial case study supporting our analysis, and a discussion of the evaluation results.

4.1 Implementation of modules for TTEthernet

For the performance evaluation, we consider two user-provided input constraints, namely *maximum end-to-end latency* for RC and TT flows, and *maximum available frame memory* for switches. The output provided by the general framework, in addition to the paths and schedule, are the i) jitter of RC flows at the switch ports, ii) queue memory reservation requirements for critical traffic, allowing to properly dimension the switch memory and

maximizing the remaining capacity for best-effort traffic, and iii) minimum end-to-end delay for RC flows (i.e. minimum achievable deadlines) based on the best solution found (i.e. with the minimum cost function).

We detail below the implementation of each module function described in Section 3.3 used for the evaluation of the general framework.

- **Routing (rt)** is implemented as follows:
 - `findRoute()`: is an implementation of the method in [2] with additional load balancing;
 - `allPaths()`: leverages the JAVA library *JGraphT* [13, 15] to compute all possible paths using the class `AllDirectedPaths`;
- **Scheduling (sch)** consists of an *SMT-based* scheduler as described in [7]:
 - `schedule()`: is computed using the constraints in [7], Section III C;
 - `checkConstraints()`: is a trivial check of the existence of offsets for the flow;
 - `sortFlows()`: detailed in Algorithm 8;
 - `sortPaths()`: detailed in Algorithm 9;
 - `costFunction()`: is directly proportional to the number of non-scheduled TT flows;
- **Formal timing analysis (fta)**, implements *Network Calculus* with linear curves²:
 - `checkConstraints()`: checks the RC constraints by implementing the TTE model proposed in [26], additionally considering the higher priority *Protocol Control Frames* (PCFs) flows in TTEthernet³. Therefore, it subtracts the PCF arrival curve (i.e. the maximum amount of data that can arrive in any time interval) alongside the TT arrival curve (cf. Theorems 3 and 7 in [26]);
 - `impossibilityTest()`: implements a simple check returning *true* when at least one flow exists, for which its maximum end-to-end deadline is less than its minimum possible end-to-end latency on the shortest path (based on the fastest possible transmission delays);
 - `feasibilityTest()`: implements a necessary optimistic analysis, whereby instead of considering the impact of the TT flow offsets as [26] to compute the maximum burst of TT traffic impacting RC, it only considers, in each output port, the maximum frame size among all transmitted TT flows. Therefore, the worst-case output port delays will be greater than or equal to the optimistic bound, and consequently, `feasibilityTest()` returns *false* if there is at least one flow with its maximum end-to-end deadline being less than the optimistic computed value;
 - `portImpact()`: checks if the input flow intersects with a flow not fulfilling its constraints;
 - `sortFlows()`, *Loop 2*: the sorted list of TT flows with highest impact on RC flows is computed by sorting TT flows from highest to lowest cardinality (Card). For details refer to lines 2 to 14 of Algorithm 2 in [7];
 - `sortFlows()`, *Loop 3*: the sorted list of RC+TT flows with highest impact on RC flows not fulfilling their deadlines⁴ is computed using Algorithm 8, with `mod=FTA`, wherein `fta.intersections(j ∈ F)` is the number of ports in the current path of flow *j* which are in common with the flows not fulfilling their constraints;

² Network Calculus is a framework allowing to compute upper-bounds for flow delays as well as backlogs, which we use to check the fulfillment of RC constraints, such as end-to-end latency, jitter and memory occupancy (cf. [14]).

³ Protocol Control Frames (PCFs) are Ethernet frames periodically transmitted by Synchronization Master nodes to implement the fault-tolerant clock synchronization in TTEthernet (cf. [12])

⁴ Note that Loop 3 is only iterated if all TT flows are scheduled.

8:16 General Framework for Routing, Scheduling and Formal Timing Analysis

- `sortPaths()`: Algorithm 9 implements the sorted list of paths;
- `costFunction()`: is assessed by adding i) the number of non-scheduled TT flows, plus ii) among all flows, the average difference between the flow deadline and its assigned end-to-end delay;

■ **Algorithm 8** Best flow to re-route for `sch.sortFlows(1, \mathcal{F}^{TT})` and `fta.sortFlows(3, \mathcal{F})`.

Require: $\mathcal{F}, i \in \{1, 3\}, f_A, f_B \in \mathcal{F}, \text{mod} \in \{\text{sch}, \text{fta}\}$

```

1:  $\text{sch}_A \leftarrow \text{mod.checkConstraints}(f_A)$ 
2:  $\text{sch}_B \leftarrow \text{mod.checkConstraints}(f_B)$ 
3: if  $f_A \in \mathcal{F}^{RC} \wedge f_B \in \mathcal{F}^{TT}$  then ▷ Case Loop 3
4:   return  $f_A$ 
5: else if  $\text{sch}_A = \text{false} \wedge \text{sch}_B = \text{true}$  then
6:   return  $f_A$ 
7: else if  $\text{sch}_A = \text{false} \wedge \text{sch}_B = \text{false} \wedge f_A.\text{deadline} > f_B.\text{deadline}$  then
8:   return  $f_A$ 
9: else if [ $\text{sch}_A = \text{true} \wedge \text{sch}_B = \text{true} \wedge \{ \text{mod.intersections}(f_A) >$ 
    $\text{mod.intersections}(f_B) \vee \{ \text{mod.intersections}(f_A) = \text{mod.intersections}(f_B)$ 
    $\wedge (f_A.\text{deadline} - f_A.\text{delay}) > (f_B.\text{deadline} - f_B.\text{delay}) \}$ ] then
10:  return  $f_A$ 
11: else
12:  return  $f_B$ 
13: end if

```

In Algorithm 8 we denote deadline_j the deadline, and delay_j the end-end-end latency, of TT flow f_j . When instantiated with $\text{mod} = \text{sch}$, `sch.intersections($f_j \in \mathcal{F}^{TT}$)` computes the number of ports in the path of TT flow f_j which are in common with the non-scheduled TT flows (note that a port in common with n flows is accounted n times).

RC flows are the best candidates to be rerouted as they do not necessitate rescheduling, which is very time expensive (line 3), therefore they are sorted first. The next best flows are flows not fulfilling their constraints (line 5), followed by flows not fulfilling their constraints with larger deadlines (line 7). The rationale of this strategy is the following: during initialization, in Algorithm 2, the selected paths tend to be among the shortest paths available. Therefore, by rerouting flows on potentially longer (and hopefully less loaded) paths, there is a higher chance to both find an acceptable path for the current flow and decrease its impact on other flows having shorter deadlines, which may likely not fulfill their deadlines on longer paths anyway. Finally, for flows fulfilling their constraints, those with the highest impact on flows not fulfilling their constraints are chosen, with the expectation of decreasing this impact by using a new path (lines 9 and 13). If the resulting impact is equivalent, the flow with the largest difference between deadline and their calculated end-to-end latency is selected, for the the reason of having a larger leeway.

In algorithm 9 we denote `sch.totalBandwidth($p \in \mathcal{P}$)` the sum of the bandwidth of TT flows in each output port of path. We denote `fta.totalBandwidth(path)` the sum of the bandwidth of RC+TT flows in each output port of path (the computation is done as the maximum of the sum per receiver, same as for `fta.totalTime(path_i)`). The function `fta.intersections(path)` is equivalent to `fta.intersections($j \in \mathcal{F}$)`, but applied to the input path, instead of the current path of flow j ;

To compare different paths of an RC flow (lines 1 to 4), a rough estimation of the RC end-to-end latency is used, using data previously computed via Network Calculus. Therefore `totalTime(path_i, receiver)` is computed for each flow receiver, as the sum

■ **Algorithm 9** Best path to re-route for $\text{sortPaths}(f \in \mathcal{F})$.

Require: $f \in \mathcal{F}, p_1, p_2 \in \mathcal{P}, \text{mod} \in \{\text{sch}, \text{fta}\}$

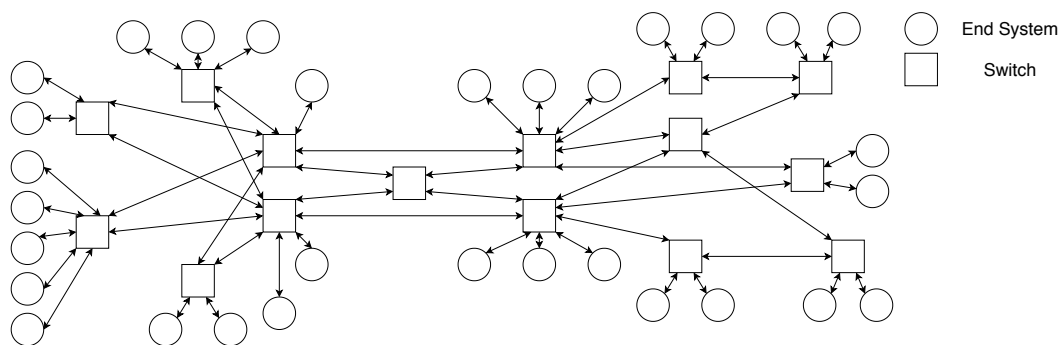
- 1: **if** $f \in \mathcal{F}^{\text{RC}} \wedge \text{fta.checkConstraints}(f) = \text{false}$ **then**
- 2: **if** $\text{fta.totalTime}(p_1) < \text{fta.totalTime}(p_2)$ **then**
- 3: **return** p_1
- 4: **end if**
- 5: **else if** $[\text{mod.intersections}(p_1) < \text{mod.intersections}(p_2)] \vee [\text{mod.intersections}(p_1) == \text{mod.intersections}(p_2) \wedge (\text{mod.totalBandwidth}(p_1) < \text{mod.totalBandwidth}(p_2))]$ **then**
- 6: **return** p_1
- 7: **end if**
- 8: **return** p_2

of the delays in each output port of path_i from the sender to a receiver. We denote $\text{fta.totalTime}(\text{path}_i)$ as the maximum of $\text{totalTime}(\text{path}_i, \text{receiver})$ over all the receivers of flow_i of path_i . Hence, the algorithm begins by selecting the path with the strictly smaller estimated total end-to-end-delay using $\text{totalTime}(p_1)$ (line 2). This is more likely to be a valid path for the current flow.

For a TT flow, the preferred path is that with less intersections with flows not fulfilling their constraints, to lessen the impact of this flow on them (line 5). If the number of intersections are identical, then the less loaded path is preferred, since the flow should have better chances of fulfilling its deadline as well as interfering with fewer other flows (line 5).

4.2 Industrial case study: the Orion network

For the performance evaluation we consider the Orion network, illustrated in Figure 2, based on the *Orion Crew Exploration Vehicle* (CEV), 606E baseline as presented in [9] and described in [23, 16]. The network consists of i) 99 TT flows with periods varying from 7.5 ms to 187.5 ms and maximum frame sizes between 87 bytes and 1518 bytes; ii) 87 RC flows with periods from 4 ms to 128 ms and maximum frame sizes from 89 to 1499 bytes. Each TT and RC flow i has a defined deadline constraint, denoted as $\text{deadline_initial}_i$.



■ **Figure 2** Orion network topology.

We have empirically determined two sets of input parameters listed in Table 1. We have observed that 10 is a good limit for trying to re-schedule TT flows in Loop 2 and, similarly, we have settled to 70% of the total number of RC+TT flows for the parameter $\text{conf.maxExploredPaths}$. Both settings show to be a good compromise exploring TT re-

routing without exhausting all possibilities, which result in a highly computationally expensive step. We have selected two different values for `conf.maxExploredPaths`, i.e. 2 and 10, to show the large impact this specific parameter has. Nevertheless, we foresee that a further study of the sensitivity of the parameters in a wider range of use cases would be beneficial in future work.

■ **Table 1** Sets of input parameters.

Parameter	GF: MEP 10	GF: MEP 2
<code>conf.maxSchedIterations</code>	10	10
<code>conf.maxExploredPaths</code>	10	2
<code>conf.maxExploredFlowReset</code>	0.7×187	0.7×187

To assess our proposal, we compare *GF: MEP 2* and *GF: MEP 10* to three other results from literature:

- **GF Shortest path**: we consider that the general framework is not limited by the configuration parameters `conf.maxSchedIterations`, `conf.maxExploredPaths`, and that `conf.maxExploredFlowReset=0`, i.e. the flows are always resorted and the memory `mem.exploredPaths_i` cleared. Moreover, the flows not fulfilling their constraints are sorted as proposed in Algorithm 8, and the flows fulfilling their constraints are sorted from highest to lowest end-to-end latency, while the path are sorted from shortest to longest. The schedule is computed with SMT, using the constraints of Section III C [7];
- **Scheduling loop** [7]: we implement the solution described in [7]
- **Static (no loop)** [19]: we consider that routing and scheduling are set with the initial solutions described in Section 4.1 and [19].

We would have liked to compare our proposal to [27], and [9], but as will be explained in Section 4.4, we lack information about their use cases to be able to do a proper comparison. However, with the three methods selected for comparison, we will be able to assess the impact of both re-routing and re-scheduling (i.e. **Static (no loop)**), the importance of re-routing in addition to re-scheduling (i.e. **Scheduling loop**), and the importance of selecting the best parameters and heuristics in the modules (i.e. **GF Shortest path**).

We have defined two test cases: i) computation of minimum end-to-end deadline constraints, and ii) analysis of deadline reduction. For i), we set all the deadlines to their initial values, except the deadline of the flows for which we want to obtain the minimum possible. Those are initialized to their minimum end-to-end latency based on the fastest possible transmission (i.e. minimum latency without any queuing delay). We set a timeout to conclude the search after 1 hour with the best found solution. For ii), we analyze the execution time (denoted *exec. time*), and cost function (denoted *cost*), when varying the deadlines of all RC flows proportionally to the initial deadline within the range 50%..100%, as shown in Table 2. For this experiment we set the timeout value to 24 hours.

It is important to note that for a small network, limiting `conf.maxExploredPaths` may limit the number of explored flows and paths, due to some part of the solution space not being accessible. Indeed, experiments run on a network with only 4 switches and 10 flows showed *GF: MEP 10* to perform better. However, for larger networks, the solution space becomes so large that exploring all possibilities tends to be intractable. Therefore, guiding the search with the parameter `conf.maxExploredPaths` shows effective. In the proposed Orion Network, both *GF: MEP 10* and *GF: MEP 2* explore the solution space until a solution is found or until a timeout expires. Limiting `conf.maxExploredPaths` does not significantly affect the total number of explored solutions, but instead guides more strongly the search toward regions of the solution space more likely to contain better solutions.

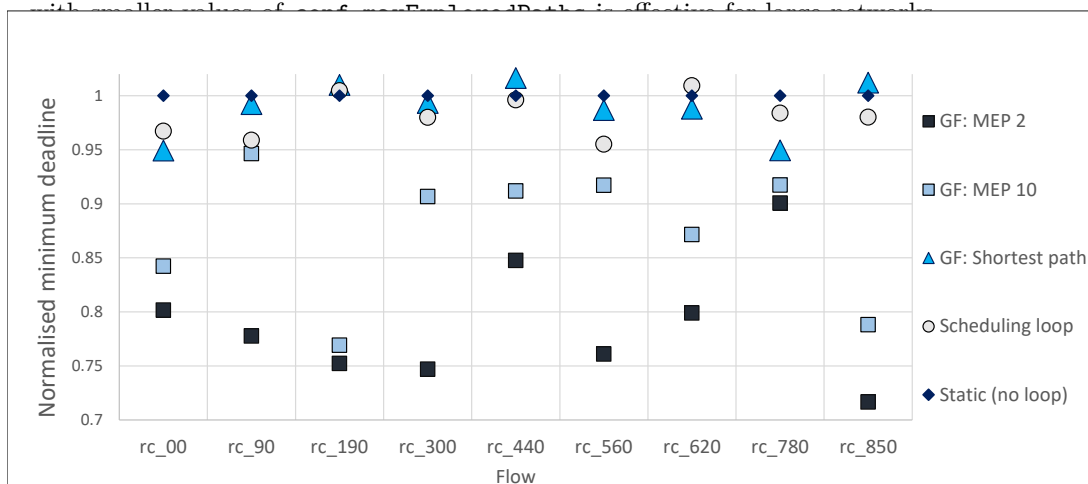
4.3 Evaluation results

In our tests, the switch memory constraints were always fulfilled, so we concentrated our efforts on the end-to-end deadlines. Note that an evaluation similar to the one we provide here could also address the switch jitter and switch memory allocation. The results of test case i) are presented in Figure 3. The minimum deadlines provided by the search are normalized using the minimum deadlines found for the *Static (no loop)* case.

The results in Figure 3 show that *GF: MEP 2* and *GF: MEP 10* find much lower deadlines than either the *Scheduling loop* and the *Static (no loop)* searches. In fact, on average, the deadlines found with *GF: MEP 2* (resp. *GF: MEP 10*) are smaller by 21% (resp. 12%) compared to *Static (no loop)*, with a maximum decrease of 28.3% for *rc_850* (resp. 23.1% for *rc_190*). On the contrary, *Scheduling loop* only reduces the minimum deadlines by 1.8% on average, with results ranging from a slight increase of 0.9% for *rc_620* to a decrease of 4.5% for *rc_560*.

The increase of minimum deadlines compared to *Static (no loop)* is due to the use of an optimization function added in the SMT as described in [7]. While it typically decreases the average delays, due to the nature of the optimization constraint and SMT solvers, in some cases it can also do the opposite.

With respect to *GF: Shortest path*, we see that the minimum deadlines found are on average 1.1% lower than the deadlines found with *Static (no loop)*, from an increase of 1.6% to a decrease of 5.1%.



■ **Figure 3** Normalised minimum deadlines for 9 randomly selected flows.

The results of test case ii) are presented in Table 2. In addition, we run an experiment with *GF: MEP 2* with deadline settings to 55%, for which we obtained an acceptable solution (i.e. cost = 0) in 18h 32min. With the exception of *Static (no loop)*, all other results in the range 100% to 80% are equivalent, finding an acceptable solution after the first iteration within 20 min. We can see that without the SMT optimization function, i.e. *Static (no loop)*, the computation time is shorter, i.e. 13 min. However, with *Static (no loop)* the deadlines cannot be reduced under 75%, with a cost function value of 1.000013. With *Scheduling loop* (resp. *GF: Shortest path*) however, the deadlines are reduced to 70%, but at the cost of a large computation time, i.e 56 min (resp. 2h 43min). With *GF: MEP 10*, we are able to reduce the deadlines down to 65%, and with *GF: MEP 2* we find a solution for 55%.

Unsurprisingly, the execution time increases when the deadlines decrease, i.e. when a solution is more difficult to find. Nevertheless, at 70%, *GF: MEP 2* finds a solution twice as fast as *Scheduling loop*, and 6 times faster than *GF: Shortest path*.

■ **Table 2** Results when varying deadlines from 100% to 50% of the initial deadlines.

deadlines	100% & 80%		75%		70%	
metrics	exec. time	cost	exec. time	cost	exec. time	cost
GF: MEP 2	20 min	0	22 min	0	26 min	0
GF: MEP 10	20 min	0	22 min	0	32 min	0
GF: shortest path	20 min	0	28 min	0	2h 43min	0
Scheduling loop [7]	20 min	0	53 min	0	56 min	0
Static (no loop) [19]	13 min	0	13 min	0	13 min	1.000013
deadlines	65%		60%		50%	
metrics	exec. time	cost	exec. time	cost	exec. time	cost
GF:MEP 2	45 min	0	4h 20min	0	24h	2.00094
GF: MEP 10	55 min	0	24h	1.000028	24h	3.00055
GF: shortest path	24h	1.00026	24h	2.00044	24h	6.00100
Scheduling loop [7]	24h	1.00040	24h	2.00041	24h	6.00066
Static (no loop) [19]	13 min	2.00027	13 min	4.00029	13 min	6.00075

Hence, in our test case ii), we have shown that *GF: MEP 2* improves the maximum deadline reduction by at least 26.7%, from 75% to 55% compared to *Static (no loop)*, and finds an acceptable solution quicker than the other searches we compared it to, when the deadlines are constraining (e.g. 70%). When no solution is found within the allocated time, *GF: MEP 2* is the search that finds the solution with the smallest cost (e.g. 50%). The number of flows not fulfilling their constraints is divided by 3 when the deadlines are divided by 3 when the deadlines are divided by 2 (i.e. 50%).

The two test cases show that with our proposed general framework, we can largely reduce the deadlines with regard to the compared state-of-the-art, i.e. *Scheduling loop* [7] and *Static (no loop)* [19]. This is because both methods explore a much reduced solution space compared to our proposal and, in particular, due to the fixed routing (and scheduling for *Static(no loop)*), they are unable to find better solutions.

The comparison between *GF: MEP 2* and *GF: MEP 10* shows the importance of selecting good parameters for the search, and the comparison with *GF: shortest path* shows the importance of selecting good parameters and good heuristics to obtain good results. In the case of shortest path, we observe that i) many paths which are longer in terms of number of hops but shorter in terms of delays are disregarded, and ii) the lists are constantly resorted, which is time expensive and can cause a lack of diversity in the selected flows and paths.

We can see that selecting a low value for `conf.maxExploredPaths` works well on large network in which exploring all solution within an acceptable time limit is not reasonable, and so potentially, reducing the solution space does not affect the number of explored solution compared to setting a larger value of `conf.maxExploredPaths`. Indeed, the only difference is which solution are being tested within the time limit. However, for smaller network larger values of `conf.maxExploredPaths` are advisable so as not to limit the number of explored solutions.

4.4 Comparison to related work

To conclude the performance evaluation, we compare our results to those found in four previous works: [19], [7], [27], and [9]. In Section 4.3, we have compared our proposed method against previous literature, namely: *Scheduling loop* [7] and *Static (no loop)* [19]. We have shown that compared to *Static (no loop)* (resp. *Scheduling loop*), our proposed solution can decrease the minimum deadlines by up to 28.3% (resp. 26.9%).

In [27], the authors compare their proposed method to the shortest paths (SPA) in a TTE network. They used SMT to compute the TT schedule as described in [19]. So, *Static (no loop)* is very close to the SPA implemented in [27]. In the performance evaluation of [27], we see that they obtain a maximum reduction of 6.41% of the worst-case delays compared to SPA, which is significantly less than the 28.3% we have obtained with our proposed method (the minimum deadline being equal to the worst-case delay). The execution times are not provided in [27], so we cannot compare the results for this metric.

In [9], the evaluation is done on a TSN network, using the schedulability of the flows to assess the solution. Unfortunately, not enough information about deadlines and traffic load is provided, which prevents a performance comparison in our evaluation.

5 Conclusion

In this paper we have presented a general framework for routing, scheduling and formal timing analysis in deterministic networks (e.g. TSN, TTE). The general framework leverages user-defined modules (i.e. routing, scheduling and Formal Timing analysis) to search for a solution fulfilling arbitrary constraints (e.g. end-to-end RC and TT delays) and outputs the best found solution (i.e. TT and RC routing, TT schedule) based on a defined cost-function.

We have provided implementation details for an instantiation of the general framework for TTEthernet, with example module implementations, input and output constraints, and cost functions. With this implementation we have evaluated the performance of our proposed general framework, compared to two state of the art methods.

We have shown that selecting good heuristics and good parameters is of paramount importance to obtain good results, and that the minimum deadlines (i.e. worst-case delays) can be reduced up to 28.3% with our proposed method, compared to the state-of-the-art solution. We have also shown that we are able to divide by up to 3 the number of flows not fulfilling their constraints compared to prior work.

The importance of good parametrization has been highlighted to select each parameter value and obtain the best solutions in the least amount of time. However, future work is necessary to analyze the impact of each parameter on the cost function and execution time, subject to networks and flow characteristics.

References

- 1 AEEC. *ARINC PROJECT PAPER 664, AIRCRAFT DATA NETWORKS, PART7, AFDX NETWORK (DRAFT)*. AERONAUTIC RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7465, November 2003.
- 2 Moses Charikar, Chandra Chekuri, To-yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed steiner problems. *Journal of Algorithms*, 33(1):73–91, 1999.
- 3 Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelik, and Wilfried Steiner. Scheduling real-time communication in IEEE 802.1Qbv Time Sensitive Networks. In *24th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2016.

- 4 J. Diemer, D. Thiele, and R. Ernst. Formal worst-case timing analysis of Ethernet topologies with strict-priority and AVB switching. In *Proc. International Symposium on Industrial Embedded Systems (SIES)*. IEEE Computer Society, 2012.
- 5 Frank Dürr and Naresh Ganesh Nayak. No-wait packet scheduling for IEEE Time-sensitive Networks (TSN). In *Proc. RTNS*. ACM, 2016.
- 6 Jonathan Falk, Frank Dürr, and Kurt Rothermel. Exploring practical limitations of joint routing and scheduling for TSN with ILP. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 136–146. IEEE, 2018.
- 7 Anaïs Finzi and Silviu S. Craciunas. Integration of SMT-based scheduling with RC network calculus analysis in TTEthernet networks. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 192–199. IEEE, 2019.
- 8 Fabrice Frances, Christian Fraboul, and Jérôme Griedu. Using network calculus to optimize the AFDX network. In *Embedded Real Time Software and Systems (ERTS)*, 2006.
- 9 Voica Gavriluț, Luxi Zhao, Michael L Raagaard, and Paul Pop. AVB-aware routing and scheduling of time-triggered traffic for TSN. *Ieee Access*, 6:75229–75243, 2018.
- 10 Jérôme Griedu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, INPT, 2004.
- 11 Institute of Electrical and Electronics Engineers, Inc. Time-Sensitive Networking Task Group. <http://www.ieee802.org/1/pages/tsn.html>, 2016. retrieved 30-Nov-2020.
- 12 Issuing Committee: As-2d2 Deterministic Ethernet And Unified Networking. SAE AS6802 Time-Triggered Ethernet. <https://www.sae.org/standards/content/as6802/>, 2011. retrieved 30-Nov-2020.
- 13 JGraphT team and contributors. JGraphT, September 2016. version: 1.0.0. URL: <https://jgrapht.org/>.
- 14 J.Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the Internet*, chapter 1, pages 3–81. Springer-Verlag, 2001.
- 15 Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Trans. Math. Softw.*, 46(2), May 2020.
- 16 M Paulitsch, E Schmidt, B Gstottenbauer, C Scherrer, and Kantz H. Time-triggered communication (industrial applications). *Time-Triggered Communication*, pages 121–152, 2011.
- 17 Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. ILP-based joint routing and scheduling for time-triggered networks. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 8–17, 2017.
- 18 Ramon Serna Oliver, Silviu S. Craciunas, and Wilfried Steiner. IEEE 802.1Qbv gate control list synthesis using array theory encoding. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018.
- 19 Wilfried Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In *Proc. RTSS*. IEEE, 2010.
- 20 Wilfried Steiner, Günther Bauer, Brendan Hall, and Michael Paulitsch. TTEthernet: Time-Triggered Ethernet. In Roman Obermaisser, editor, *Time-Triggered Communication*. CRC Press, August 2011.
- 21 Domițian Tămaș-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proc. CODES+ISSS*. ACM, 2012.
- 22 Domițian Tămaș-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proceedings of the eighth IEEE/ACM/I-FIP international conference on Hardware/software codesign and system synthesis*, pages 473–482, 2012.
- 23 Domițian Tămaș-Selicean, Paul Pop, and Wilfried Steiner. Design optimization of TTEthernet-based distributed real-time systems. *Real-Time Systems*, 51(1):1–35, 2015.

- 24 Daniel Thiele, Philip Axer, and Rolf Ernst. Improving formal timing analysis of switched Ethernet by exploiting FIFO scheduling. In *Proceedings of the 52nd Annual Design Automation Conference*, page 41. ACM, 2015.
- 25 Qinghan Yu and Ming Gu. Adaptive group routing and scheduling in multicast time-sensitive networks. *IEEE Access*, 8:37855–37865, 2020.
- 26 Luxi Zhao, Paul Pop, Qiao Li, Junyan Chen, and Huagang Xiong. Timing analysis of rate-constrained traffic in TTEthernet using network calculus. *Real-Time Systems*, 53(2):254–287, 2017.
- 27 Zhong Zheng, Feng He, and Huagang Xiong. Routing optimization of Time-Triggered Ethernet based on genetic algorithm. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE, 2020.

Correctness and Efficiency Criteria for the Multi-Phase Task Model

Rémi Meunier ✉ 

IRIT, AUSY, INSA Toulouse, France

Thomas Carle ✉ 

IRIT, Université Toulouse 3 Paul Sabatier, CNRS, France

Thierry Monteil ✉ 

IRIT, INSA Toulouse, CNRS, France

Abstract

This paper investigates how the multi-phase representation of real-time tasks impacts their implementation and the precision of the interference analysis in a multi-core context. In classical scheduling and interference analyses, tasks are represented as a single phase with a duration equal to their Worst-Case Execution Time (WCET) in isolation, annotated with their worst-case number of accesses. We propose a general formal definition of a task model in which tasks are represented as a sequence of such phases: the multi-phase model. We then provide a set of general correction criteria for the implementation of tasks represented in the multi-phase model, which is agnostic of the analysis method applied on the tasks. We also use the multi-phase model on an avionics case-study and study its impact on the interference analysis. Finally, we define a set of efficiency criteria using a statistical study of the most efficient multi-phase shapes.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Multicore architectures; Computer systems organization → Embedded software

Keywords and phrases Task model, Interference, Multicore architectures

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.9

Funding This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the MeSCAliNe (ANR-21-CE25-0012) project.

1 Introduction

The growing adoption of multi-core processors in industrial real-time systems [21, 22] raises the challenge of providing safe and tight Worst-Case Execution Time (WCET) bounds for tasks running in parallel on separate cores. Indeed, in multi-core architectures, the cores execute their processes/threads independently from one another, but they share some hardware components such as caches, buses and memories. Interference may happen in these shared components: when a task requires to access a component which is already in use by another task running on another core, it has to wait until the component is free again. This phenomenon incurs execution delays which depend on the context of the task execution (which other tasks are running in parallel, and are they accessing the shared resources?). In traditional single-core WCET analysis [1, 3], each task is analysed in isolation i.e. as if no other task was running in parallel. Then a schedulability or Worst-Case Response Time (WCRT) analysis is performed using a model in which each task is represented by its WCET, in order to guarantee that each individual task meets its deadline or that the system as a whole meets an end-to-end timing constraint. A direct consequence of the delays incurred by interference is that traditional WCETs no longer represent a safe upper-bound on the execution time of the tasks when they are run on multi-core processors. It becomes necessary to model tasks using at least their WCET in isolation and their worst-case number



© Rémi Meunier, Thomas Carle, and Thierry Monteil;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 9; pp. 9:1–9:21



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of accesses to shared components, and to perform an additional interference analysis in order to obtain a safe over-estimation of their execution time. However, this classical model, which maps one task to one temporal phase was not designed with interference analysis in mind, and may not be the best-suited to analyse tasks running in parallel.

More recent models represent each task as a sequence of *phases*, each characterized by a WCET and a number of accesses, either as an attempt to increase the precision of the interference analysis [2, 4, 19], or in order to build schedules in which there is no interference [9, 20]. This *multi-phase* abstraction maps temporal phases to actual sections of code which are separated by synchronizations. The shape of the phases may be dictated by a programming model in which the programmers insert synchronizations at particular points (e.g. [9]), or may be designed during the analysis and enforced by synchronizations that are injected in the code afterwards (e.g. [5]).

In the remainder of the paper, we focus on memory accesses as the sole source for interference in the system. However, the abstractions that we describe naturally support any other kind of interference source: the only thing that changes is the analyses that must be performed on the code in order to obtain the abstract models of the tasks, which are not considered in the scope of the paper. In particular, we consider non-preemptive static (or fixed-priority time-triggered) scheduling, but limited preemptions could be supported by adapting classic cache-related preemption delay (CRPD) computation techniques.

In this paper we make the following contributions:

- we give a general, formal description of the multi-phase model that is agnostic of the method used to analyze the tasks. As such it can be seen as a generalization of the 2-phase [18] and 3-phase [9] models, and as an extension of the model of [5] to the notion of synchronizations.
- we investigate the relationship between the phases and the synchronizations that enforce them, that is to say between the multi-phase analysis model of tasks and their actual implementation. We show that synchronizations are not mandatory at the boundaries between two consecutive phases, and provide criteria to validate the implementation of a task w.r.t. the assumptions that were made during the analysis. Once again the criterion is agnostic about the way the analysis is actually performed.
- we apply the multi-phase model on an avionics case-study, and show that correlating the shape of the model to the actual behavior of the tasks may not lead to the best results.
- As a result we investigate how the shape of multi-phase representations impacts the result of static interference analysis, and deduce a promising heuristic objective to build multi-phase models of tasks. Since we discuss a general model and not a particular method, this study is based on a statistical experiment relying on synthetic artifacts.

The paper is organized as follows: we present the related work in Section 2 and we introduce formal definitions in Section 3. Then, in Section 4 we study the relationship between phases and synchronizations. In Section 5, we present our results on the ROSACE case-study. In Section 6, we detail our statistical study of the multi-phase model, and finally we conclude in Section 7.

2 Related Work

The problem of identifying, quantifying and possibly reducing the interference between real-time tasks running on multi-core processors is one of the most pressing issues that the real-time community is facing. Consequently, a lot of work has been done already, and various methods have been developed, although none of them seems to be entirely satisfying [13].

One of the most comprehensive approaches so far, presented in [6], uses the execution traces of the tasks composing the system, and models all the hardware components shared between cores (e.g. buses, RAMs) in order to quantify the exact worst-case amount of interference in the system, given a fixed priority scheduling policy. This analysis method is thus able to provide a safe and tight interference-aware WCRT for a task system. However the authors point out that working with all the possible execution traces of all the tasks composing a system is not feasible for realistic, industrial systems. They advocate the use of a more abstract representation of the tasks execution in order to overcome this intractability issue, but do not provide such an abstraction. It thus remains open to find a suitable candidate abstraction to achieve a trade-off between tractability and precision of the analysis.

Other approaches, inspired by compilation methods and low-level code analysis, use static scheduling approaches to handle, reduce or suppress the interference in the system. These methods are inspired by the PRedictable Execution Model (PREM) introduced in [18]. This model, originally designed for single-core processors with Input/Output registers, abstracts the execution of tasks as a sequence of so called memory and execution phases. In a memory phase, the core executes only memory (or I/O register) operations, which require to send or receive data across an interconnect. In an execution phase however, the code is executed locally in the core, and is guaranteed not to make accesses to the interconnect. As a result, only the memory phases are subject to interference. The Acquisition Execution Restitution (AER) model [9] can be seen as an extension of the PREM model to the context of avionics systems running on multi-core architectures. In AER, each task is divided into exactly three phases: an acquisition phase that loads to a core-local memory (cache or scratchpad) all the code and data which may be necessary to execute the task, an execution phase that executes the task code locally, and a restitution phase in which the results of the task are written back to the shared memory. A static schedule of the tasks is computed in which the acquisition and restitution phases of tasks are guaranteed not to happen simultaneously, and synchronizations are inserted to enforce this schedule. As a result, the system is completely free from interference. This line of work simplifies the analysis, by completely suppressing the interference in the system, but also introduces some limitations of its own. First, in these approaches, the tasks software has to be written with the phases in mind: this imposes constraints on the way the code is written, and makes it challenging to use legacy code. Automatic transformations of functions into the AER/PREM model have been developed as an attempt to shift the design burden from the programmer to the compiler [10, 11, 14–16, 23]. However, all these methods perform consequent modifications of the applications in order to make them comply with the AER/PREM model, and thus do not solve the limitations regarding legacy code. As a second limitation, the pre-loading of the tasks code and data increases the memory requirements of the system by forcing the load of code and data which may not be used during the execution phase (e.g. due to conditional execution). Finally, the static nature of these solutions is compatible to systems in which certification constraints are very high (e.g. avionics systems), but may be too rigid for less critical applications.

In [23], the authors evaluate the PREM and AER models in different scenarios in which interference is either prohibited or can be tolerated. They conclude that for a given task system, tolerating interference is more effective in terms of WCRT than building a schedule without interference at all. In the same spirit, the Time Interest Points (TIPs) approach [5] has been developed to reconcile the multi-phased task model with legacy code. In this model, a multi-phase representation of the tasks is obtained through static analysis of the binary code of the tasks. As a consequence, no restriction is put on the way the source code must be written. Each phase of a task may perform a certain amount of memory accesses,

which is determined by the analysis. Using this representation, an interference analysis can be performed as part of a WCRT analysis such as the one in [6], or as part of a static scheduling/compiling approach (e.g. [7, 8]). Moreover, the analysis can be tuned in order to produce different representations of the same task (as in [5]). However, to the best of our knowledge, no study has yet been performed in order to define what a “good” multi-phased representation is. In this paper we provide a first answer to this question.

3 Formal Models

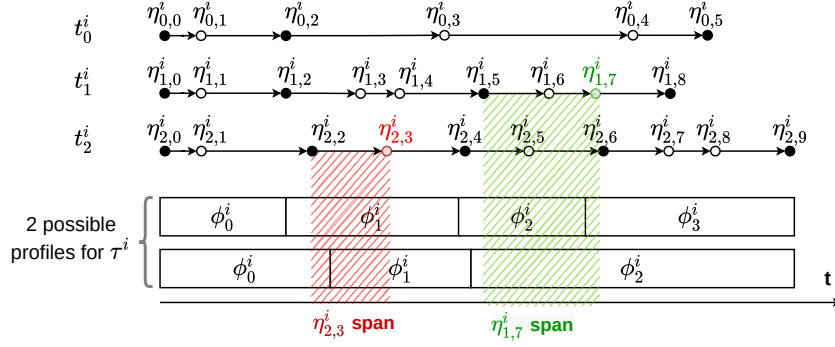
Notation	Definition
τ^i	task i
ϕ_k^i	phase k in the representation of τ^i
$\phi_k^i.d$	start date of ϕ_k^i without interference
$\phi_k^i.dur$	worst-case duration of ϕ_k^i without interference
$\phi_k^i.m$	maximum number of memory accesses performed within ϕ_k^i
t_j^i	execution trace j of task τ^i
$\eta_{j,k}^i$	node k in trace t_j^i
$\eta_{j,k}^i.it$	instruction represented by $\eta_{j,k}^i$
$\eta_{j,k}^i.d$	worst-case execution date of $\eta_{j,k}^i$ without interference
$\eta_{j,k}^i.m$	maximum number of memory accesses performed by $\eta_{j,k}^i$
$\eta_{j,k}^i.sync$	True if the node is synchronized, i.e. cannot be executed before its $\eta_{j,k}^i.d$
$slast(\eta_{j,k}^i)$	last synchronized node before $\eta_{j,k}^i$ in trace t_j^i
$t_j^i _{\phi_k^i}$	restriction of trace t_j^i to ϕ_k^i , i.e. the set of nodes in t_j^i that may execute during ϕ_k^i

We model a system of real-time tasks τ^i ($i \geq 0$). Each task is represented in two separate ways, as depicted in Figure 1:

- a “time-centric” representation called *multi-phase*. In this abstraction, the task is modelled by a sequence of time slots, called *phases*, which covers its WCET. We call this sequence of phases a *profile*. Each phase is associated with an upper bound on the number of memory accesses that the task can perform during the corresponding time slot. This representation is used to statically compute the schedule and perform the interference analysis of the system (in a timing-compositional approach). The mapping between tasks and multi-phase profiles is not bijective: multiple profiles can be found which represent the same task.
- a “code-centric” representation. In this abstraction, a task is represented by all its possible *execution traces* (i.e. all the possible sequences of instructions executed from the start of the task to its end). Since this set may be too large to analyze in practice, we consider memory-centric traces: only instructions which may perform memory accesses¹ are represented in the traces, and the rest of the instructions is abstracted by computing local WCETs. This representation is an intermediate step to go from the binary code

¹ In modern processors, the actual accesses may not be performed as soon as the corresponding instruction is executed e.g. if a store buffer delays store operations. However it is possible to statically bound the time window during which the access can be performed. For clarity reasons, we consider in this paper only the date when the instruction initiates the access in the pipeline, but the model can be easily extended to work with an interval of potential access dates.

of a task to its multi-phase representation, and back: it allows the number of memory accesses in each phase to be bounded correctly, and to insert synchronization code at the correct locations in the binary to enforce the scheduling choices.



■ **Figure 1** Three traces and two profiles for task τ^i . Red and green rectangles show the potential span of nodes $\eta_{2,3}^i$ and $\eta_{1,7}^i$ respectively.

The scope of this paper does not include the methods required to obtain trace-based and phase-based representations of tasks. We focus instead on the relationship between these two abstraction levels. In this section, we describe these models formally and present this relationship.

3.1 Task Models

We denote $\mathbb{P}^i = \{\phi_l^i | 0 \leq l < \Phi^i\}$ the ordered set of phases (i.e. the multi-phase *profile*) representing the execution of task τ^i , with Φ^i the number of phases. Each ϕ_l^i is defined by:

- $\phi_l^i.d$: its start date.
- $\phi_l^i.dur$: its worst-case duration in isolation (without interference).
- $\phi_l^i.m$: the worst-case number of memory accesses that may be performed within $[\phi_l^i.d, \phi_l^i.d + \phi_l^i.dur[$.

The date of ϕ_0^i , which is also the start date of task τ^i without interference, is set when the static schedule of the system is built. Then, for each ϕ_l^i ($l > 0$) the start date is defined by:

$$\phi_l^i.d = \phi_0^i.d + \sum_{0 \leq q < l} \phi_q^i.dur \quad (1)$$

Alternatively, we can define recursively the start date of each phase (except the first one) by:

$$\forall l > 0, \phi_l^i.d = \phi_{l-1}^i.d + \phi_{l-1}^i.dur$$

In order to compute the worst-case number of memory accesses performed during a given phase (i.e. $\phi_l^i.m$), the code portions of τ^i that may be executed during ϕ_l^i must be identified and analyzed. To do so, we introduce $\mathbb{T}^i = \{t_j^i | 0 \leq j < T^i\}$ the set of *execution traces* of τ^i , where T^i is the number of traces. Each trace corresponds to a possible execution of τ^i (corresponding to a particular set of inputs) and is a sequence of nodes $\eta_{j,k}^i$ representing instructions with $0 \leq k < N_j^i$ the node's index in its sequence. $\eta_{j,0}^i$ is the *entry point* of task τ^i and each node is defined by:

- $\eta_{j,k}^i.it$: the instruction represented by $\eta_{j,k}^i$. Here, an instruction is not just understood as an element of the core ISA (e.g. the ADD instruction), but as a particular instruction in the binary code of the task. Thus, nodes from different traces may reference the same instruction in the code.
- $\eta_{j,k}^i.m \in \mathbb{N}$: the worst-case number of memory accesses performed by $\eta_{j,k}^i.it$.
- $\eta_{j,k}^i.d$: the worst-case execution date of $\eta_{j,k}^i.it$ in trace t_j^i .

3.2 Synchronized Nodes

As pointed out in Section 2, working on the complete set of execution traces of all tasks composing the system is not realistic. As a consequence, we formulate our correctness criteria using memory-centric *abstract traces*: the nodes composing the traces that we consider only represent the instructions that may perform memory accesses. The rest of the instructions is abstracted by computing local WCETs between consecutive memory accesses and accounting for these durations in the worst-case execution date of the nodes $(\eta_{j,k}^i.d)^2$. As a result, in this model each node is guaranteed not to execute after its worst-case date, but is *a priori* able to execute anytime before this date. In order to account safely for the accesses in the phases, we thus would have to account for the accesses performed by a node in all phases that start before the worst-case date of this node. This would lead to huge over-approximations. In order to limit this approximation, some selected nodes must be *synchronized*: synchronization code is inserted in the program to ensure that the synchronized nodes cannot be executed before their worst-case date. The synchronization code can be added by the programmer directly in the source code of the tasks, by the compiler as part of a low-level compilation pass, or during an automatic code re-engineering process to adapt legacy code to the multi-phase model. We attract the reader's attention to two particular aspects of the model described in Section 3.1: (i) the execution date for nodes that reference the same instruction in different traces and (ii) the modelling of instructions inside loops which may appear multiple times in the same trace at different dates. Both these aspects have to do with the way synchronizations are implemented in the tasks. When complex synchronization mechanisms are used (e.g. that are aware of the current execution trace or of the iteration count in the current loop), the same memory instruction in the code may be modelled as two (or more) nodes with different dates, which perfectly fits the model. If, however, the synchronization mechanism is unaware of the context, the worst-case execution date of nodes that reference the same instruction on separate traces must be the same. Since the model uses worst-case dates, the date chosen for all these nodes must be the maximum date amongst them. Additionally, without a context-aware mechanism, synchronizations inside loops become impossible to implement, so the model naturally fits this case. In this paper, we voluntarily keep the model as general as possible and make no assumption on the implementation of the synchronization mechanisms in order to formulate correctness criteria that apply in all circumstances.

To keep track of the synchronized nodes, we add the boolean attribute $\eta_{j,k}^i.sync$ which is true if the node is synchronized and false otherwise.

Using these synchronizations, the accesses performed by any node must only be accounted for in the phases that: (1) finish after the last synchronization prior to the node **AND** (2) start before the worst-case date of the node.

² Our criteria are also valid for simple tasks for which obtaining and manipulating the exact timed execution traces is possible

This is illustrated in Figure 1, which depicts 3 execution traces (t_0^i , t_1^i and t_2^i) and 2 possible profiles for a task τ^i . Synchronized nodes are depicted in black in the traces. The red (resp. green) rectangle shows the time window in which the accesses of node $\eta_{2,3}^i$ (resp. $\eta_{1,7}^i$) must be accounted for. In the first profile, the accesses of $\eta_{1,7}^i$ must be considered in phases ϕ_2^i and ϕ_3^i , whereas in the second profile, they would only be considered in ϕ_3^i .

It is important to note that since $\eta_{j,k}^i.d$ is a worst-case date, if node $\eta_{j,k}^i$ is synchronized, then its execution date is exactly³ $\eta_{j,k}^i.d$. We denote $s_{last}(\eta_{j,k}^i)$ the last synchronized node before $\eta_{j,k}^i$ in trace t_j^i . By convention, we set $s_{last}(\eta_{j,k}^i) = \eta_{j,k}^i$ when $\eta_{j,k}^i.sync$.

To account for the tasks schedule, for all tasks τ^i , the entry node (on any trace t_j^i) is synchronized and its worst-case execution date is set to the start of the first phase of the profile:

► **Property 1.** $\forall i, j : \eta_{j,0}^i.sync \wedge \eta_{j,0}^i.d = \phi_0^i.d$

The worst-case date of any other node $\eta_{j,k}^i$ with $k > 0$ is defined according to the date of the last synchronized node on its trace:

► **Property 2.** $\eta_{j,k}^i.d = \eta_{j,s}^i.d + \sum_{s \leq t < k} wct(\eta_{j,t}^i.it, \eta_{j,t+1}^i.it)$

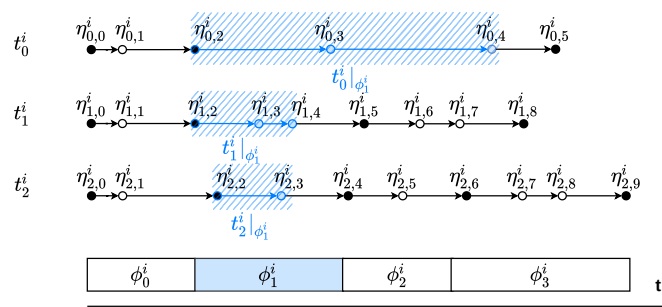
where $wct(\eta_{j,t}^i.it, \eta_{j,t+1}^i.it)$ is the WCET between instructions $\eta_{j,t}^i.it$ and $\eta_{j,t+1}^i.it$, and $\eta_{j,s}^i$ is $s_{last}(\eta_{j,k}^i)$ if $\neg \eta_{j,k}^i.sync$ and $s_{last}(\eta_{j,k-1}^i)$ otherwise.

A node $\eta_{j,k}^i$ can only be executed in the interval $[s_{last}(\eta_{j,k}^i).d, \eta_{j,k}^i.d]$. As we saw in the example of Figure 1, this interval may overlap with several phases of the task profile.

We denote $t_j^i|_{\phi_i^i}$ the set of nodes in trace t_j^i that may be executed within $[\phi_i^i.d, \phi_i^i.d + \phi_i^i.dur]$, called the *restriction* of trace t_j^i to phase ϕ_i^i :

$$t_j^i|_{\phi_i^i} = \{\eta_{j,k}^i | (\eta_{j,k}^i.d \geq \phi_i^i.d) \wedge (s_{last}(\eta_{j,k}^i).d < \phi_i^i.d + \phi_i^i.dur)\}$$

The notion of restriction of a trace to a phase is illustrated in Figure 2 on 3 traces over phase ϕ_1^i .



■ **Figure 2** Restrictions of traces t_0^i , t_1^i and t_2^i to phase ϕ_1^i .

3.3 Maximum Number of Accesses in a Phase

The number of accesses that may be performed during a phase for an individual trace is equal to the sum of the accesses of the nodes from this trace that may be executed in the phase. During the execution of a task, only one trace executes (which one depends on the

³ With a precision of a few cycles depending on the implementation of the synchronization mechanism.

execution context): as a consequence, the worst-case number of accesses performed during a phase is equal to the maximum number of accesses that may be performed by any execution trace during that phase.

► **Property 3.** *The worst-case number of accesses that may be performed during phase ϕ_l^i , denoted $\phi_l^i.m$, is equal to the maximum of accesses per trace during phase ϕ_l^i :*

$$\phi_l^i.m = \max_{0 \leq j < T^i} \left(\sum_{\eta_{j,k}^i \in t_j^i | \phi_l^i} \eta_{j,k}^i.m \right)$$

► **Correctness criterion 1.** *The formula of Property 3 provides a conservative estimation of the number of memory accesses that can occur during the phases of a multi-phase profile.*

Since nodes may span over multiple phases, the number of accesses counted task-wise may be overestimated, even when some nodes are synchronized. However, nodes from a trace which span over multiple phases may be “covered” by other nodes from another trace performing more accesses on a given phase. For example, in Figure 2, if we consider that each node performs 1 access, trace t_2^i is the local worst trace on ϕ_3^i with 4 nodes performing accesses and trace t_1^i is the local worst trace on ϕ_2^i with 3 nodes performing accesses. On phase ϕ_1^i , traces t_0^i and t_1^i both have 3 nodes performing accesses. In such circumstances, although node $\eta_{0,4}^i$ spans over ϕ_3^i , ϕ_2^i and ϕ_1^i , it does not contribute to any over-approximation.

We quantify the task-wise over-approximation of memory accesses compared to the 1-phase model, by computing the difference between the sum of accesses accounted for in each phase, and the worst trace-wise number of accesses.

► **Property 4.** *The memory access over-approximation in a multi-phase profile of a task τ^i compared to its 1-phase representation is equal to:*

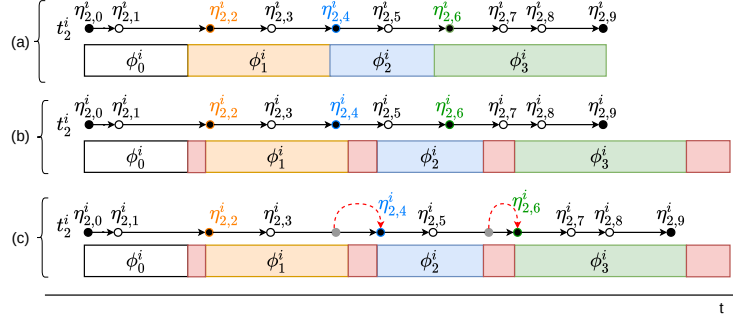
$$\Delta = \left(\sum_{0 \leq l < \Phi^i} \phi_l^i.m \right) - \max_{0 \leq j < T^i} \left(\sum_{0 \leq k < N_j^i} \eta_{j,k}^i.m \right)$$

4 Consequences of the Interference Analysis

Notation	Definition
$\phi_l^i.p$	timing penalty added to ϕ_l^i due to potential interference
$\phi_l^i.d^\#$	<i>post-analysis</i> start date of ϕ_l^i
$\eta_{j,k}^i.d^\#$	worst-case date of node $\eta_{j,k}^i$ in the presence of interference

In this section, we consider a task system for which an analysis has provided a multi-phase model as well as a selection of synchronized nodes for each task. We assume that this task system is scheduled statically (the $\phi_0^i.d$ for each τ^i are selected and the start dates of the other phases are computed using equation 1), and that an interference analysis (such as e.g. [7]) is applied to compute and account for the effect of potential interference between the tasks phases, assuming the timing-compositionality of the target processor [12]. In practice, each phase that potentially suffers from interference is extended using a time penalty, and the next phases are postponed accordingly. This extension may violate assumptions that were made on the correspondence between phases and traces: in particular the restrictions of traces to phases that were computed prior to the interference analysis may no longer be correct, resulting in the possibility that some contentions between cores may happen in phases in which they were not accounted for.

4.1 Example



■ **Figure 3** A trace and its corresponding phases representation : (a) in isolation, (b) after the interference analysis, red rectangles are the timing penalty added for each phase, (c) after a correction on nodes dates.

Figure 3 displays trace t_2^i and the profile from Figure 2, at three stages of the analysis:

- (a) depicts the trace and phases before the interference analysis. We have:

$$t_2^i|_{\phi_0^i} = \{\eta_{2,0}^i, \eta_{2,1}^i\}; t_2^i|_{\phi_1^i} = \{\eta_{2,2}^i, \eta_{2,3}^i\}; t_2^i|_{\phi_2^i} = \{\eta_{2,4}^i, \eta_{2,5}^i\}; t_2^i|_{\phi_3^i} = \{\eta_{2,6}^i, \eta_{2,7}^i, \eta_{2,8}^i, \eta_{2,9}^i\}$$
 Additionally, we consider that for this task, $\phi_1^i.m = 2$ and $\phi_2^i.m = 2$.
- (b) shows the same trace and profile after the interference analysis (assuming other tasks in the system): the effect of interference is materialized by timing penalties on the phases (the red rectangles after each phase). $t_2^i|_{\phi_1^i}$, $t_2^i|_{\phi_2^i}$ and $t_2^i|_{\phi_3^i}$ are different than in (a):

$$t_2^i|_{\phi_0^i} = \{\eta_{2,0}^i, \eta_{2,1}^i\}; t_2^i|_{\phi_1^i} = \{\eta_{2,2}^i, \eta_{2,3}^i, \eta_{2,4}^i, \eta_{2,5}^i\}; t_2^i|_{\phi_2^i} = \{\eta_{2,5}^i, \eta_{2,6}^i, \eta_{2,7}^i, \eta_{2,8}^i\}; t_2^i|_{\phi_3^i} = \{\eta_{2,8}^i, \eta_{2,9}^i\}$$
 As a consequence, the worst-case amount of accesses that can happen during phases ϕ_1^i and ϕ_2^i is higher than what was assumed and therefore their interference penalty and those of the tasks scheduled in parallel are no longer conservative.
- (c) represents a solution to respect the model's assumptions of (a): the synchronized date of $\eta_{2,4}^i$ (resp. $\eta_{2,6}^i$) is set to the new starting date of ϕ_2^i (resp. ϕ_3^i), which is the unique phase in which it was accounted for in (a). With this slight modification, the restrictions of t_2^i to each phase are identical to the ones in (a) and the $\phi_i^i.m$ that was computed in isolation for each phase remains correct.

4.2 Enforcing the Model's Assumptions and the Analysis Results

Since the duration and start dates of phases can be changed as a result of the interference analysis, new attributes are added to the formal model of the phases:

- $\phi_i^i.p \geq 0$ is the timing penalty added to ϕ_i^i due to potential interference. It is a conservative bound computed during the interference analysis.
- $\phi_i^i.d^\#$ is the *post-analysis* date of ϕ_i^i , i.e. its start date taking into account the potential interference in the system.

After the interference analysis, the start date of some tasks may be postponed due to interference that delays previous tasks. $\phi_0^i.d^\#$ is thus fixed by applying the interference analysis results to the initial schedule. The start dates of all other phases ϕ_l^i describing the execution of τ^i are computed as:

$$\phi_l^i.d^\# = \phi_0^i.d^\# + \sum_{0 \leq q < l} (\phi_q^i.dur + \phi_q^i.p) \quad (2)$$

► **Correctness criterion 2.** *The synchronization dates in the final implementation of tasks must at least be equal to the start date of the corresponding phase: for each synchronization node $\eta_{j,k}^i \in t_j^i|_{\phi_n^i}$, the synchronization date is set to at least $\phi_n^i.d^\#$. This way it is guaranteed that nodes after $\eta_{j,k}^i$ cannot execute and thus produce accesses before the start of ϕ_n^i .*

It seems straightforward that, by construction, a task set implemented using this rule is guaranteed to fulfill the assumptions made during the interference analysis: during the execution of the system, memory accesses will only occur at times that were accounted for during the analysis, and thus the amount of interference cannot be larger in practice than what was accounted for. However, although this implementation rule directly guarantees that accesses are not performed before the phases in which they are accounted for, it may be harder to convince oneself that they cannot occur later than the end of these phases. Consequently, and given the potentially critical nature of the tasks modelled in the multi-phase representation, we provide in the remainder of the section a formal proof of the correctness of this implementation scheme w.r.t. the result of the interference analysis. Once again, this is completely agnostic of the analysis method, as long as it correctly provides a conservative bound on the interference level.

We denote $\eta_{j,k}^i.d^\#$ the *post-analysis* worst-case date of node $\eta_{j,k}^i$. The post-analysis dates of nodes are upper bounds on the worst-case execution dates of nodes in the presence of interference. We start by characterizing those bounds in our formal model (Properties 5, 6 and 7), and then use them to prove the correctness of the implementation of a multi-phase model of tasks.

First, the post-analysis execution date of the entry point of each task τ^i is the post analysis start date of its first phase ϕ_0^i .

► **Property 5.** *For any task τ^i : $\forall j < T^i, \eta_{j,0}^i.d^\# = \phi_0^i.d^\#$*

Second, correctness criterion 2 has the following consequences for the post-analysis execution date of any synchronized node $\eta_{j,k}^i$ (except the entry point) of any task τ^i :

- if the phase ϕ_n^i in which the node was supposed to be executed is postponed due to interference penalties on previous phases, the node cannot be executed before the post-analysis start date of ϕ_n^i .
- if previous synchronized nodes see their execution dates postponed, the synchronization date of $\eta_{j,k}^i$ must be postponed accordingly, and thus computed from the post-analysis date of the previous synchronized node $\eta_{j,s}^i$. In this case, we must consider the interference that can take place between $\eta_{j,s}^i$ and $\eta_{j,k}^i$. If there exists one or more phases that span entirely between the two nodes, their penalties are added to the post-analysis date of $\eta_{j,k}^i$ (which is conservative). Moreover, by convention we count in the post-analysis date of $\eta_{j,k}^i$ the entire amount of penalty of the phase to which it belongs (which is also conservative since it accounts for the interference that can occur on each access in the phase prior to the synchronization node, and on each access that may occur until the next synchronization node).

► **Property 6.** *For any synchronized node $\eta_{j,k}^i$ of any trace t_j^i of task τ^i :*

$$(k > 0 \wedge \eta_{j,k}^i.sync \wedge \eta_{j,k}^i \in t_j^i|_{\phi_n^i} \wedge \eta_{j,s}^i = slast(\eta_{j,k-1}^i) \wedge \eta_{j,s}^i \in t_j^i|_{\phi_m^i})$$

$$\Rightarrow \eta_{j,k}^i.d^\# = max(\phi_n^i.d^\#, \eta_{j,s}^i.d^\# + \sum_{s \leq l < k} wcet(\eta_{j,l}^i.it, \eta_{j,l+1}^i.it) + \sum_{m < b \leq n} \phi_b^i.p)$$

► **Correctness criterion 3.** *The synchronization dates in the final implementation of tasks must not be set to a value higher than the date computed in Property 6.*

Finally, for any non-synchronized node, its post-analysis date accounts for the possible postponing of the previous synchronized node $\eta_{j,s}^i$. Note that the potential interference occurring between them has been accounted for entirely in the post-analysis date of the previous synchronized node.

► **Property 7.** *For any non-synchronized node $\eta_{j,k}^i$ of any trace t_j^i of task τ^i :*

$$(\neg \eta_{j,k}^i.\text{sync} \wedge \eta_{j,s}^i = \text{slast}(\eta_{j,k}^i)) \Rightarrow \eta_{j,k}^i.d^\# = \eta_{j,s}^i.d^\# + \sum_{s \leq l < k} \text{wcet}(\eta_{j,l}^i.it, \eta_{j,l+1}^i.it)$$

4.3 Proof of Correctness

We now prove that any task system which respects the 3 correctness criteria is correct w.r.t. the results of the interference analysis i.e. cannot generate interference that was not accounted for.

First, the difference between the start date of a synchronized node $\eta_{j,k}^i$ before and after the interference analysis is bounded by the difference between the start date of the phase ϕ_l^i in which it is executed, before and after the interference analysis, added to the maximum amount of interference that can occur in ϕ_l^i .

► **Lemma 1.** $\forall \eta_{j,k}^i: (\eta_{j,k}^i.\text{sync} \wedge \eta_{j,k}^i \in t_j^i|_{\phi_l^i}) \Rightarrow \eta_{j,k}^i.d^\# - \eta_{j,k}^i.d \leq \phi_l^i.d^\# - \phi_l^i.d + \phi_l^i.p$

Proof. We will prove by induction that the property is true for all synchronized nodes. If $\eta_{j,k}^i$ is the entry node of τ^i , the proof is direct using Properties 1 and 5. Otherwise, using Property 6, $\eta_{j,k}^i.d^\#$ is either equal to $\phi_l^i.d^\#$ or must be computed from the previous synchronized node on trace t_j^i . Let $\eta_{j,s}^i = \text{slast}(\eta_{j,k-1}^i)$, and assume that the property is true for $\eta_{j,s}^i$. Then,

■ If $\eta_{j,k}^i.d^\# = \phi_l^i.d^\#$:

since $\eta_{j,k}^i \in t_j^i|_{\phi_l^i}$, by definition $\eta_{j,k}^i.d \geq \phi_l^i.d$, and thus $\eta_{j,k}^i.d^\# - \eta_{j,k}^i.d \leq \phi_l^i.d^\# - \phi_l^i.d$.

■ Otherwise:

$\eta_{j,k}^i.d^\# = \eta_{j,s}^i.d^\# + \sum_{s \leq a < k} \text{wcet}(\eta_{j,a}^i.it, \eta_{j,a+1}^i.it) + \sum_{m < b \leq l} \phi_b^i.p$. Using Property 2, we

get: $\eta_{j,k}^i.d^\# - \eta_{j,k}^i.d = \eta_{j,s}^i.d^\# - \eta_{j,s}^i.d + \sum_{m < b \leq l} \phi_b^i.p$. The induction hypothesis gives us:

$\eta_{j,s}^i.d^\# - \eta_{j,s}^i.d \leq \phi_m^i.d^\# - \phi_m^i.d + \phi_m^i.p$, where ϕ_m^i is the phase in which $\eta_{j,s}^i$ executes. If $m = l$ (i.e. both nodes execute in the same phase) the property is directly proven for node $\eta_{j,k}^i$. Otherwise, $m < l$ and then $\phi_l^i.d^\# - \phi_l^i.d = \phi_m^i.d^\# - \phi_m^i.d + \sum_{m \leq b < l} \phi_b^i.p$ (using

Equations 1 and 2), and thus the property is also proven.

By induction, we just proved that the property holds for all synchronized nodes. ◀

We are now ready to prove the correctness property:

► **Theorem 1.** *For any task system that respects correctness criteria 1, 2 and 3, for any $\eta_{j,k}^i$ of any task τ^i , if $\eta_{j,k}^i$ spans over a phase ϕ_l^i after the interference analysis, then $\eta_{j,k}^i$ was necessarily accounted in the restriction of trace t_j^i to ϕ_l^i before the analysis:*

$$\forall 0 \leq j < T^i, \forall 0 \leq k < N_j^i, \forall 0 \leq l < \Phi^i :$$

$$[\text{slast}(\eta_{j,k}^i).d^\#, \eta_{j,k}^i.d^\#] \cap [\phi_l^i.d^\#, \phi_l^i.d^\# + \phi_l^i.dur + \phi_l^i.p] \neq \emptyset \Rightarrow \eta_{j,k}^i \in t_j^i|_{\phi_l^i}$$

Proof. The case where $\eta_{j,k}^i$ is the entry node is direct. For all other nodes we consider separately the case of synchronized nodes and of non-synchronized nodes.

Case 1: $\eta_{j,k}^i \cdot \text{sync}$ is true:

By convention, $s_{last}(\eta_{j,k}^i) = \eta_{j,k}^i$. Let us assume ϕ_l^i such that $\eta_{j,k}^i \cdot d^\# \in [\phi_l^i \cdot d^\#, \phi_l^i \cdot d^\# + \phi_l^i \cdot dur + \phi_l^i \cdot p[$. Let us denote ϕ_z^i the phase such that $\eta_{j,k}^i \in t_j^i|_{\phi_z^i}$ (z is unique because $\eta_{j,k}^i$ is synchronized). We want to prove that $l = z$. Using Property 6, either $\eta_{j,k}^i \cdot d^\# = \phi_z^i \cdot d^\#$ or it is greater. If it is equal, then directly $\phi_l^i = \phi_z^i$ because phases of the same task do not overlap. Otherwise, if $z > l$ then $\eta_{j,k}^i \cdot d^\# > \phi_z^i \cdot d^\# \geq \phi_l^i \cdot d^\# + \phi_l^i \cdot dur + \phi_l^i \cdot p$ which contradicts the assumption. So z would have to be less than l . Now, since $\eta_{j,k}^i \in t_j^i|_{\phi_z^i}$, $\eta_{j,k}^i \cdot d - \phi_z^i \cdot d < \phi_z^i \cdot dur$. At the same time, $\eta_{j,k}^i \cdot d^\# \geq \phi_l^i \cdot d^\# \geq \phi_z^i \cdot d^\# + \phi_z^i \cdot dur + \phi_z^i \cdot p$, so $\eta_{j,k}^i \cdot d^\# - \phi_z^i \cdot d^\# \geq \phi_z^i \cdot dur + \phi_z^i \cdot p$. This contradicts Lemma 1, from which we conclude that $l = z$. This concludes the proof for case 1.

Case 2: $\eta_{j,k}^i \cdot \text{sync}$ is false:

Let ϕ_l^i such that $[s_{last}(\eta_{j,k}^i) \cdot d^\#, \eta_{j,k}^i \cdot d^\#] \cap [\phi_l^i \cdot d^\#, \phi_l^i \cdot d^\# + \phi_l^i \cdot dur + \phi_l^i \cdot p] \neq \emptyset$. Let us denote ϕ_m^i the phase to which $s_{last}(\eta_{j,k}^i) \cdot d^\#$ belongs, and assume by absurd that $\eta_{j,k}^i \notin t_j^i|_{\phi_l^i}$.

Then by definition either $(s_{last}(\eta_{j,k}^i) \cdot d > \phi_l^i \cdot d + \phi_l^i \cdot dur)$ or $(\eta_{j,k}^i \cdot d < \phi_l^i \cdot d)$.

If $s_{last}(\eta_{j,k}^i) \cdot d > \phi_l^i \cdot d + \phi_l^i \cdot dur$: then $m > l$, and thus using Property 6: $s_{last}(\eta_{j,k}^i) \cdot d^\# \geq \phi_m^i \cdot d^\# \geq \phi_l^i \cdot d^\# + \phi_l^i \cdot dur + \phi_l^i \cdot p$, which contradicts the original assumption.

If $\eta_{j,k}^i \cdot d < \phi_l^i \cdot d$, then using Property 2: $s_{last}(\eta_{j,k}^i) \cdot d + \sum_{s \leq t < k} w_{cet}(\eta_{j,t}^i \cdot it, \eta_{j,t+1}^i \cdot it) < \phi_l^i \cdot d$.

Then, we can deduce:

$$\begin{aligned}
 s_{last}(\eta_{j,k}^i) \cdot d^\# + \sum_{s \leq t < k} w_{cet}(\eta_{j,t}^i \cdot it, \eta_{j,t+1}^i \cdot it) &< \phi_l^i \cdot d + s_{last}(\eta_{j,k}^i) \cdot d^\# - s_{last}(\eta_{j,k}^i) \cdot d \\
 \stackrel{\text{Prop. 7}}{\Rightarrow} \eta_{j,k}^i \cdot d^\# &< \phi_l^i \cdot d + s_{last}(\eta_{j,k}^i) \cdot d^\# - s_{last}(\eta_{j,k}^i) \cdot d \\
 \Rightarrow \eta_{j,k}^i \cdot d^\# &< \phi_l^i \cdot d + s_{last}(\eta_{j,k}^i) \cdot d^\# - s_{last}(\eta_{j,k}^i) \cdot d + \sum_{b=m+1}^{l-1} \phi_b^i \cdot p \\
 \stackrel{\text{Lemma 1}}{\Rightarrow} \eta_{j,k}^i \cdot d^\# &< \phi_l^i \cdot d + \phi_m^i \cdot d^\# - \phi_m^i \cdot d + \phi_m^i \cdot p + \sum_{b=m+1}^{l-1} \phi_b^i \cdot p \\
 \Rightarrow \eta_{j,k}^i \cdot d^\# &< \phi_m^i \cdot d^\# + \phi_m^i \cdot p + \sum_{b=m+1}^{l-1} \phi_b^i \cdot p + \sum_{b=m}^{l-1} \phi_b^i \cdot dur \\
 \Rightarrow \eta_{j,k}^i \cdot d^\# &< \phi_l^i \cdot d^\#
 \end{aligned}$$

which contradicts the initial hypothesis. We conclude that necessarily $\eta_{j,k}^i \in t_j^i|_{\phi_l^i}$. ◀

We just proved that the correctness criteria that we enumerated in the first part of the paper guarantee that the implementation of a task system described in the multi-phase model is correct w.r.t. a chosen interference-aware static schedule. These criteria are very simple, which makes them easy to verify and offers a lot of room for optimizations in the analysis of tasks, both in order to derive a profile for tasks and to select the synchronization nodes. In the remainder of the paper, we concentrate on the efficiency of the model w.r.t. the interference analysis. We start by experimenting the multi-phase model on a case study, and then perform a statistical analysis in order to derive general efficiency criteria which can in the future serve as an objective function for analysis heuristics.

5 Efficiency of the Multi-Phase Model on the ROSACE Case-Study

ROSACE [17] is a flight controller case-study composed of 15 communicating tasks running at different frequencies. We followed the Time Interest Points methodology described in [5] to obtain the worst-case execution traces and multi-phase profiles for the ROSACE tasks. Basically, we used the OTAWA static analysis tool to:

- Detect the instructions that are not statically guaranteed to result in a cache hit.
- Build an “abstract” CFG in which the nodes are the instructions that were detected in the previous step. Each edge of this graph is decorated with the WCET of the code portion between its source and sink nodes, computed using OTAWA.
- Build the execution traces by enumerating this graph. In our experiments on ROSACE, the average number of traces by task was around 88, with a peak at 1280 for the *aircraft_dynamics* task. The graph enumeration may lead to combinatorial explosion for arbitrarily complex applications. This issue can be mitigated by adding extra synchronizations in the traces (e.g. at the end of a if-then-else or loop construct) that factorize multiple traces for the rest of the enumeration.
- For each trace, generate a multi-phase profile in which each memory access has a dedicated phase spanning the duration of the access, using the worst-case dates in the trace.
- Build the intersection of the profiles of all traces. This intersection is a profile that keeps all access phases from all traces. The rest of the profile is composed of phases guaranteed to feature no access.
- From this profile, extract the phases with a size larger than a parameter δ in which no access occurs. Parameter δ , which we varied in our experiments, specifies a minimum size threshold for the phases of the generated profiles.
- For the remainder of the phases in the intersection profile, fuse them together if their duration is less than δ .

This method creates multi-phase profiles for tasks but says nothing about the selection of the synchronization nodes. We thus added a very simple method to select synchronization nodes, using the correctness criteria of Sections 3 and 4. For each phase ϕ_l^i , we selected as synchronized node for each trace t_j^i the first node $\eta_{j,k}^i$ with $\phi_l^i.d \leq \eta_{j,k}^i.d < \phi_l^i.d + \phi_l^i.dur$ (if such a node exists). None of the tasks needed context-aware synchronizations so the synchronization date was always chosen as the worst-case date of the synchronized instruction in the program. The combination of the heuristic and of our node selection pass does not perform any optimization. In our analysis, we considered a target hardware architecture composed of a multicore processor (2, 4 or 8 cores) in which each core features a L1 LRU data cache, and an instruction scratchpad which holds the totality of the code needed by the core to execute. Additionally, we considered a memory latency of 50 cycles for non-cached accesses. The tasks were compiled for ARM targets, and we considered that the cores were running at a frequency of 10MHz (otherwise the tasks WCETs were too small compared to their periods so there was no interference in the schedule).

Table 1 presents statistics on the multi-phase profiles of the ROSACE tasks for 3 values of δ . This parameter has a consequent impact on the number of phases, on the number of synchronizations and on the over-approximation (defined in Property 4) in the generated profiles. With a δ equal to 50 cycles (the memory latency of our targets), 569 phases are generated for all tasks. Our method determined 315 synchronizations nodes which corresponds to roughly 1 synchronization every 6 or 7 instructions on average. This number may be too high to be realistically used in practice, but is in part due to the small tasks of the case-study which are composed of only a few tens of instructions. As δ grows, the

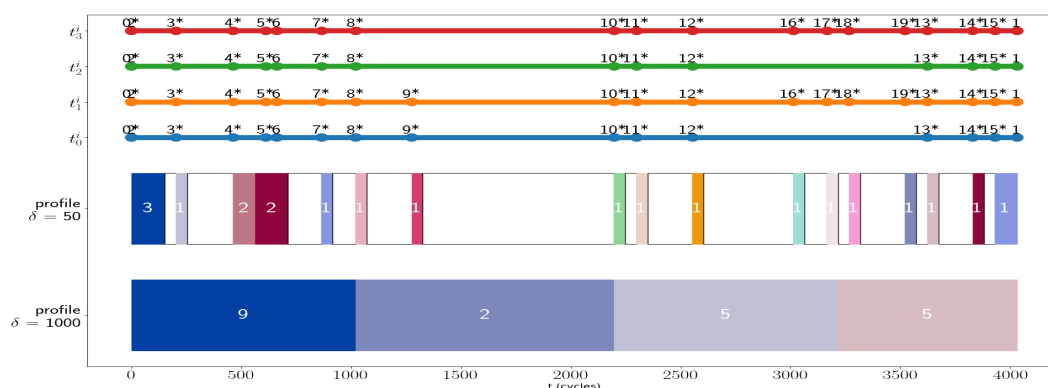
■ **Table 1** Rosace multi-phase tasks statistics for 3 multi-phase minimum duration (δ).

Task	instr (#)	traces (#)	$\delta = 50$			$\delta = 500$			$\delta = 1000$		
			ov_app (%)	sync (#)	phases (#)	ov_app (%)	sync (#)	phases (#)	ov_app (%)	sync (#)	phases (#)
engine	40	1	0	10	18	0	4	4	0	2	2
elevator	47	1	0	9	15	0	3	4	0	2	2
aircraft_dyn	1217	1280	21.18	92	167	12.94	37	38	11.76	22	23
h_filter	77	4	0	18	34	0	7	7	0	5	4
az_filter	77	4	0	17	32	0	7	7	0	5	4
q_filter	106	12	14.29	32	55	0	11	9	0	8	5
vz_filter	106	12	10.34	32	55	0	11	9	0	8	5
va_filter	77	4	0	17	32	0	7	7	0	5	4
h_command	18	1	0	7	13	0	2	2	0	1	1
altitude_hold	65	3	6.25	15	26	0	7	6	0	5	3
vz_control	70	1	0	23	44	0	7	8	0	4	4
va_control	73	1	0	24	45	0	7	8	0	4	4
va_command	18	1	0	7	13	0	2	2	0	1	1
delta_th	15	1	0	6	10	0	2	2	0	1	1
delta_e	15	1	0	6	10	0	2	2	0	1	1
TOTAL	2021	1327	7.88	315	569	3.33	116	115	3.03	74	64
Gain (2 cores)			7.83%			6.79%			6.01%		
Gain (4 cores)			13.21%			8.45%			7.48%		
Gain (8 cores)			7.60%			3.74%			2.14%		

number of generated phases and of synchronization nodes get lower: 115 phases and 116 synchronizations for $\delta = 500$, and 64 phases and 74 synchronizations for $\delta = 1000$, with 4 tasks having only 1 phase in their representation. Our method for selecting the synchronizations and counting the accesses in each phase is basic: it respects correctness criterion 1, but does not optimize the number of synchronizations. As a result, on each trace one node is selected as synchronization for each phase, even when it is not necessary. This explains in part the high count for synchronizations. These synchronizations can be implemented by a variety of methods. One naive method is to poll a register that counts the number of cycles (e.g. a time stamp counter) until the start date of the corresponding phase is reached. This can be implemented with a simple loop composed of only a few instructions (depending on the ISA it can be as small as 3 instructions – compare, conditional jump, jump back). The precision of each synchronization depends on the depth of the pipeline (which imposes the duration of the jumps), but these small overheads can be taken into account in the analysis, and they do not accumulate as the number of phases grows, because the synchronizations are based on dates, not durations. Context-aware synchronizations (e.g. inside loops) are more complex to implement, and may impose restrictions on the analysis. Since there was no need for those in the ROSACE case-study, we leave for future work the efficient and correct implementation of context-aware synchronizations.

Figure 4 displays the generated traces and two profiles for the *az_filter* task. When $\delta = 1000$, the profile is composed exclusively of phases in which the number of memory accesses is strictly positive. On the other hand, when $\delta = 50$ most of the profile is composed of phases guaranteed to perform no memory access.

Using the generated profiles, we produced static schedules of the application for target processors featuring 2, 4 and 8 cores. The schedules represent one hyper-period of the tasks. The tasks were mapped to cores following their utilization (using a simple greedy algorithm) and scheduled using the rate monotonic policy with the frequencies specified in the original paper [17]. We then performed an interference analysis: we detected the phases that overlap with each other 2 by 2 on different cores and extended them using a penalty computed as the maximum number of contentions multiplied by the cost of a



■ **Figure 4** *az_filter* task: traces (top) and generated profile (bottom).

memory access. Moreover, the total penalty that a phase can suffer from any other core (taken separately) is bounded by the number of accesses of the phase. This is a classical interference computation assuming e.g. a FIFO bus. We measured the total reserved time for the tasks (including the interference penalties) on the hyper-period, using the classical 1-phase model, and using our generated multi-phase profiles. We computed the gain as: $gain = (time_1_phase - time_multi_phase)/time_1_phase$ and reported it in Table 1. The multi-phase model yields a gain of 7.83% (resp. 13.21% and 7.60%) with δ equal to 50 cycles on 2 cores (resp. 4 and 8 cores). The gain gets reduced as δ grows and the generated profiles resemble more and more the 1-phase model. However, even in the case of $\delta = 1000$, the multi-phase model outperforms the 1-phase model by 6.01% (resp. 7.48%, 2.14%) on 2 cores (resp. 4 cores, 8 cores). One noticeable point is that the profiles with a lower δ have a higher over-approximation and still get the best gains compared to the 1-phase model: over-approximation at the profile level is not an indicator of the good performance of the model at the task-system level.

Our conclusions from the case-study are the following:

- the multi-phase model can yield a substantial gain compared to the 1-phase model.
- the over-approximation in the profile of an individual task is not correlated to the gain obtained at the task system level. Consequently, the optimal profile for a task may not be derived from its execution behavior (when do the memory accesses happen?), but from extrinsic properties. As we show in Section 6.2, particular shapes of profiles behave significantly better than others during the interference analysis.
- a trade-off must be found during the construction of the tasks profiles between the number of synchronizations and the efficiency (the gain) of the model.
- this gain is computed after the schedule is built and the interference analysis is performed and is thus not accessible when the profiles are being constructed: other criteria must be found, which can be computed directly during the construction of the profiles.

In an attempt to find such criteria and to confirm these conclusions, we performed a statistical study which we describe in the next section.

6 Profile Shape-Based Efficiency Criteria

In this section, we investigate how the shape of multi-phase profiles impacts the result of the interference analysis. As we show in this section, we found efficiency criteria which concern the multi-phase model itself and are thus extrinsic to the analysed tasks. To do

so, we conducted a statistical study on synthetic profiles generated using multiple input parameters that are summed up in Table 2. Profiles are generated by choosing the values for the attributes of the phases, using random draws from normal distributions centered around the input parameters: for example inside a generated profile, each phase has its own duration and number of accesses, but in average the durations and number of accesses meet the input parameters. We do not consider system-level parameters such as task periodicity, data dependencies or elaborate mapping and scheduling strategies in this section because our focus is on showing how the model reacts in the presence of interference. We thus choose a setting in which a lot of tasks are executed in parallel with no slack time.

■ **Table 2** Tests input parameters.

Parameter	Values section 6.2	Values section 6.3
Over-approximation of accesses (%)	0 to 30 (step 5)	0 to 30 (step 5)
Nb tasks per core	{1, 2, 3}	{2, 3, 4, 5}
Interference time penalty (cycles)	{5, 10, 20}	{5, 10}
Accesses per cycle	0.01 to 0.1 (step 0.01)	0.01 to 0.1 (step 0.01)
Nb long phases per UP L type task	{5, 10, 15, 20}	{5, 10, 15, 20}
Avg. number of short phases per long phase	{2, 3, 4, 5}	{2, 3, 4, 5}
Task duration (cycles)	17,500 to 175,000	17,500 to 175,000

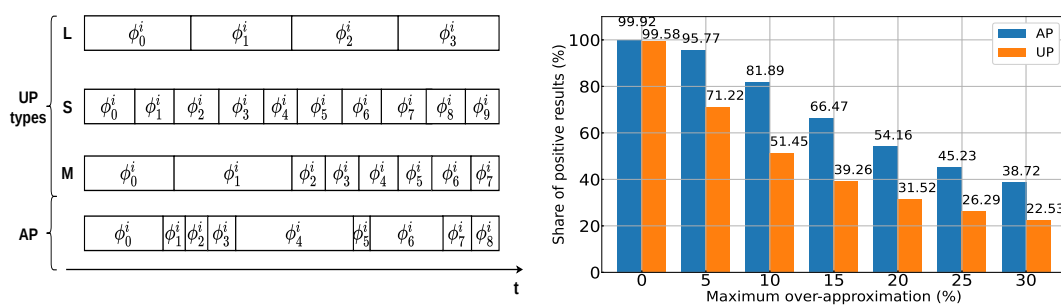
6.1 Tests Execution and Metrics

The execution of a test consists in three steps. First, we generate the set of multi-phase profiles corresponding to the input parameters. The generated tasks all share the same period, have no explicit data dependency and have a synchronous release. Then, we map the tasks into a number of cores specified for each test and schedule them as soon as possible, with no optimization in the mapping or scheduling choices. This context stresses the multi-phase model. Finally, we perform an interference analysis on the scheduled task system, as we did with the ROSACE case-study. The over-approximation level (cf. Property 4) is an input parameter to the tests. The ROSACE case-study showed that this over-approximation is not directly correlated with the gain yielded by the multi-phase model, so the study of synthetic profiles will provide us valuable insight on the performance of the model. Since we schedule the tasks as soon as possible with no slack, we redefine the notion of *gain* using the end dates of the schedules obtained with the 1-phase and multi-phase models: $gain = (end_1_phase - end_multi_phase) / end_1_phase$.

6.2 Looking for the Best Multi-Phase Profile Shapes

We started by generating shapes composed of sequences of phases of similar durations (with a standard deviation of 500 cycles), which we call Uniform Profiles (UP). The 3 UP kinds generated are depicted in Figure 5a: long (L) profiles composed only of long phases, short (S) profiles composed only of short phases or mix (M) profiles divided in 2 equal parts with respectively long and short phases. We performed 352,800 tests with different combinations of generation parameters. Each core is only assigned tasks of one UP kind (S, L or M) and has the same number of tasks, which all have the same duration and are released synchronously. With this setup, only the kind of UP assigned may differ for each core, so an easy comparison between the various combinations of UP kinds can be performed.

We tested all the combinations of the profiles on 2, 4 and 8 cores. Our results showed that the best profiles hosted the 3 categories S, M and L with a majority of S profiles. This is coherent because the multi-phase model increases its gain whenever, for a given phase, the



(a) Examples of possible UP types and AP.

(b) Tests with positive gain according to the maximum over-approximation value included.

■ Figure 5 AP vs UP profiles.

number of accesses it may perform is higher than the number of possible concurrent accesses from phases running in parallel (and it does not exceed the total accesses of the tasks in parallel). In this experiment each task has strictly the same size, so the probability to fall into this case is higher with S profiles which are composed of more phases.

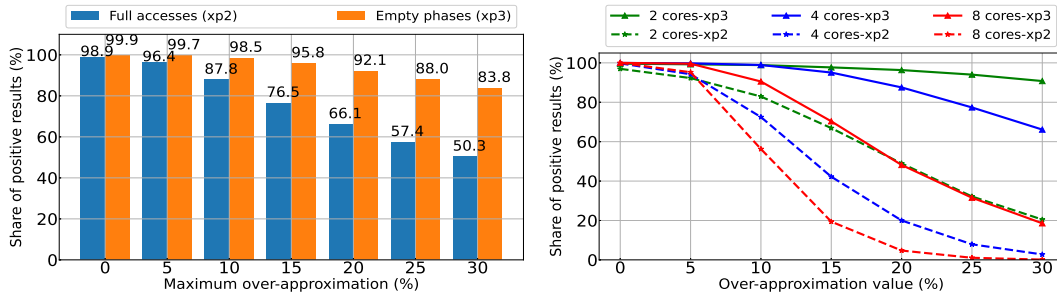
As the combination of short and long phases, with more short phases than long ones, performed better in our first experiment, we then tested a different profile shape that allows an alternation between short and long phases in the tasks as depicted in the bottom of Figure 5a. We name this profile shape Alternation Profile (AP). We compared the results of AP with the other profile shapes using the same test parameters. The results, given in Figure 5b, indicate that AP systematically outperforms UP regardless of the over-approximation value and of the other parameters. Therefore, we focus on AP as the best multi-phase profile for the rest of the experiments.

► **Efficiency criterion 1.** *Multi-phase profiles which alternate between long phases and grapes of small phases tend to perform better than other shapes.*

6.3 Comparison Between Multi-Phase AP and 1-Phase Model

In this section we assess the performance of the AP shape using a pool of 806,400 tests in which tasks lengths vary in a range of $\pm 25\%$ around a value that is provided as input to the test generator. We first consider task profiles in which all phases perform memory accesses. Figure 6a gives the share of experiments in which the multi-phase AP outperforms its 1-phase counterpart for different values of maximum over-approximation (blue bars). When there is no over-approximation, the multi-phase model almost always performs better. The 1-phase model does not perform as well as the AP until all experiments with over-approximations ranging from 0% to 30% are included. According to Figure 6b (dashed lines), the share of positive results significantly decreases with the number of cores in our tests when the system-wise over-approximation value is superior to 10%, in particular when more than two cores are involved. This is coherent with the fact that the over-approximation of accesses is amplified as the number of cores grows when performing the interference analysis. In our experiments on the ROSACE case-study, the over-approximation always remained under 10% system-wise, regardless of the size of the phases. As a conclusion:

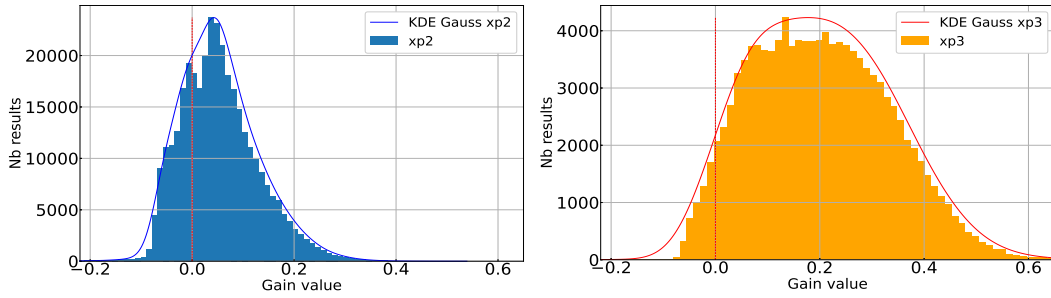
► **Efficiency criterion 2.** *The over-approximation plays a lesser role than the shape of the profile at the level of individual tasks, but should still be kept within acceptable levels system-wise, preferably under 10%.*



(a) Ratio of tests with positive gain.

(b) Tests with positive gain by core number.

■ **Figure 6** Experiments with accesses in each phase ($xp2$) and with phases without accesses ($xp3$).

(a) AP profiles without empty phases ($xp2$).(b) AP profiles with 1/3 of empty phases ($xp3$).

■ **Figure 7** Gain distribution for AP profiles with 0 to 10% of over-approximation.

This can be achieved by increasing the number of synchronizations. A special effort must be put on the traces with the most accesses as the over-approximation at the phase-level is propagated to the task-level only if it concerns, for each phase, the trace having the most accesses. The over-estimation can also be reduced by fusing phases together when locally some traces perform most of their accesses in different phases.

Moreover, when the over-approximation cannot be lowered, scheduling optimizations can be applied in the same spirit as the ones used in AER in order to contain the negative effect of over-approximation.

In the profiles generated for the above experiments, all phases perform accesses. Nonetheless, the presence of phases without accesses in profiles is expected in practice e.g. due to cache effects, and is likely to improve the AP results. Indeed, phases running in parallel with other phases without accesses are guaranteed not to cause contention, while they would in the 1-phase model. Therefore, we performed a new series of tests to estimate the impact of phases without accesses in AP profiles. We modified the profiles already generated for the previous experiments, by randomly selecting one third of each task's phases and setting their accesses count to 0. The results are presented in Figures 6a (orange bars) and 6b (full lines). First, the share of positive results is significantly improved and still at more than 80% when including results with 30% of over-approximation, so the model is less sensitive to over-approximation. This is linked to situations where accesses due to over-approximation are in parallel with no other accesses so they do not lead to additional penalties. Second, the gain distribution with over-approximation ranging from 0 to 10%, given by Figures 7a and 7b, is also significantly improved with an average value of 20.1% while it is 5.3% for profiles without empty phases. Consequently:

► **Efficiency criterion 3.** *Phases that perform no access have a significant positive effect on the interference analysis results.*

As we saw with ROSACE the number of synchronizations, of phases that perform no access and the over-approximation can be adjusted as a trade-off in the analysis.

7 Conclusion and Future Work

We presented a formal framework for the multi-phase task model including a set of properties that guarantee the correctness of the implemented task system. These properties are agnostic about the methods that generate the profiles and select the location of the synchronizations that enforce them. We combined our criteria to a simple heuristic to obtain multi-phase representations of tasks on the ROSACE case-study, and concluded that the shape of the model impacts the efficiency of the interference analysis, regardless of the analyzed task. We thus conducted a statistical study in order to investigate which kinds of profiles perform the best. We concluded that profiles alternating long and short phases tend to perform better, and that profiles featuring phases that do not perform accesses are particularly efficient. As part of future work, we plan on defining an automatic method for the design of optimized multi-phase profiles from the tasks binary code, and on defining scheduling optimizations which benefit from the multi-phase model. Moreover, we will work on the efficient implementation of context-aware synchronizations, focusing on regular synchronization patterns inside loop iterations.

References

- 1 AbsInt. aiT. <https://www.absint.com/ait/index.htm>.
- 2 Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling. In Audrey Queudet, Iain Bate, and Giuseppe Lipari, editors, *RTNS'2021: 29th International Conference on Real-Time Networks and Systems, Nantes, France, April 7-9, 2021*, pages 123–133. ACM, 2021. doi:10.1145/3453417.3453433.
- 3 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P.uschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.
- 4 Thomas Carle and Hugues Cassé. Reducing timing interferences in real-time applications running on multicore architectures. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018, July 3, 2018, Barcelona, Spain*, volume 63 of *OASIScs*, pages 3:1–3:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/OASIScs.WCET.2018.3.
- 5 Thomas Carle and Hugues Cassé. Static extraction of memory access profiles for multi-core interference analysis of real-time tasks. In Christian Hochberger, Lars Bauer, and Thilo Pionteck, editors, *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings*, volume 12800 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2021. doi:10.1007/978-3-030-81682-7_2.
- 6 Robert I. Davis, Sebastian Altmeyer, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real Time Syst.*, 54(3):607–661, 2018. doi:10.1007/s11241-017-9285-4.

- 7 Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. Scaling up the memory interference analysis for hard real-time many-core systems. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 330–333. IEEE, 2020. doi:10.23919/DATE48585.2020.9116460.
- 8 Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, and Amaury Graillat. Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware. *ACM Trans. Archit. Code Optim.*, 16(3):24:1–24:27, 2019. doi:10.1145/3328799.
- 9 G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In *ERTS'14*, 2014.
- 10 Björn Forsberg, Marco Solieri, Marko Bertogna, Luca Benini, and Andrea Marongiu. The predictable execution model in practice: Compiling real applications for COTS hardware. *ACM Trans. Embed. Comput. Syst.*, 20(5):47:1–47:25, 2021. doi:10.1145/3465370.
- 11 Frédéric Fort and Julien Forget. Code generation for multi-phase tasks on a multi-core distributed memory platform. In *25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2019, Hangzhou, China, August 18-21, 2019*, pages 1–6. IEEE, 2019. doi:10.1109/RTCSA.2019.8864558.
- 12 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In Alain Plantec, Frank Singhoff, Sébastien Faucou, and Luís Miguel Pinho, editors, *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 299–308. ACM, 2016. doi:10.1145/2997465.2997471.
- 13 Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019. doi:10.1145/3323212.
- 14 Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-prem: Automated software refactoring for predictable execution on COTS embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20-22, 2014*, pages 1–10. IEEE Computer Society, 2014. doi:10.1109/RTCSA.2014.6910515.
- 15 Joel Matejka, Björn Forsberg, Michal Sojka, Zdenek Hanzálek, Luca Benini, and Andrea Marongiu. Combining PREM compilation and ILP scheduling for high-performance and predictable mpsoc execution. In Quan Chen, Zhiyi Huang, and Pavan Balaji, editors, *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2018, February 25, 2018, Vienna, Austria*, pages 11–20. ACM, 2018. doi:10.1145/3178442.3178444.
- 16 Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated generation of time-predictable executables on multicore. In Yassine Ouhammou, Frédéric Ridouard, Emmanuel Grolleau, Mathieu Jan, and Moris Behnam, editors, *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018, Chasseneuil-du-Poitou, France, October 10-12, 2018*, pages 104–113. ACM, 2018. doi:10.1145/3273905.3273907.
- 17 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE case study: From simulink specification to multi/many-core execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 309–318. IEEE Computer Society, 2014. doi:10.1109/RTAS.2014.6926012.
- 18 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 269–279. IEEE Computer Society, 2011. doi:10.1109/RTAS.2011.33.

- 19 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In Giovanni De Micheli, Bashir M. Al-Hashimi, Wolfgang Müller, and Enrico Macii, editors, *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, pages 741–746. IEEE Computer Society, 2010. doi:10.1109/DATE.2010.5456952.
- 20 Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding communication delays in contention-free execution for spm-based multi-core architectures. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, volume 133 of *LIPICs*, pages 25:1–25:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ECRTS.2019.25.
- 21 O. Sander, F. Bapp, L. Dieudonne, T. Sandmann, and J. Becker. The promised future of multi-core processors in avionics systems. *CEAS Aeronautical Journal*, 2017. doi:10.1007/s13272-016-0228-x.
- 22 J. Schneider, M. Bohn, and R. Rößger. Migration of automotive real-time software to multicore systems: First steps towards an automated solution. In *22nd EUROMICRO Conference on Real-Time Systems*, 2010.
- 23 Matheus Schuh, Claire Maiza, Joël Goossens, Pascal Raymond, and Benoît Dupont de Dinechin. A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 283–295. IEEE, 2020. doi:10.1109/RTSS49844.2020.00034.

Overrun-Resilient Multiprocessor Real-Time Locking

Zelin Tong ✉

University of North Carolina at Chapel Hill, NC, USA

Shareef Ahmed ✉

University of North Carolina at Chapel Hill, NC, USA

James H. Anderson ✉

University of North Carolina at Chapel Hill, NC, USA

Abstract

Existing real-time locking protocols require accurate worst-case execution time (WCET) estimates for both tasks and critical sections (CSs) in order to function correctly. On multicore platforms, however, the only seemingly viable strategy for obtaining such estimates is via measurements, which cannot produce a true WCET with certainty. The absence of correct WCETs can be partially ameliorated by enforcing *execution budgets* at both the task and CS levels and by using a locking protocol that is *resilient to budget overruns*, i.e., that ensures that the schedulability of non-overrunning tasks is not compromised by tasks that do overrun their budgets. Unfortunately, no fully overrun-resilient locking protocol has been proposed to date for multiprocessor systems. To remedy this situation, this paper presents two such protocols, the OR-FMLP and the OR-OMLP, which introduce overrun-resiliency mechanisms to two existing multiprocessor protocols, the spin-based FMLP and suspension-based global OMLP, respectively. In devising such mechanisms, undo code can be problematic. For the important locking use case of protecting shared data structures, it is shown that such code can be avoided entirely by using *abortable critical sections*, a concept proposed herein that leverages obstruction-free synchronization techniques. Experiments are presented that demonstrate both the effectiveness of the mechanisms introduced in this paper and their cost.

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

Keywords and phrases Real-Time Systems, Real-Time Synchronization, Budget Enforcement

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.10

Related Version *Full Version*: <https://cs.unc.edu/~anderson/papers/ecrts22-long.pdf>

Supplementary Material *Software (Experiment Code)*: <https://cs.unc.edu/~anderson/papers/Experiments.tar>

Funding Supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

1 Introduction

Many safety-critical systems require *real-time safety certification* that hinges on both timing analysis and schedulability analysis. The goal of *timing analysis* is to produce *worst-case execution times* (WCETs) for executable code. *Schedulability analysis* then determines whether a system's timing constraints are met, assuming valid WCETs are provided. Due to the advent of multicore technologies, work on timing and schedulability analysis has largely focused on the multiprocessor case in recent years [11, 17, 34].

A troubling disconnect. In the multiprocessor case, a largely unnoticed fundamental disconnect exists when using timing- and schedulability-analysis together to validate real-time correctness. There is consensus today that static timing-analysis tools may never be



© Zelin Tong, Shareef Ahmed, and James H. Anderson;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 10; pp. 10:1–10:25



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Overrun-Resilient Multiprocessor Real-Time Locking

a practical reality for multicore machines due to the highly complex nature of multicore architectures [43]. The only alternative is to use *measurement-based* timing analysis, a topic that has been the focus of considerable recent work [16,18]. With measurement-based timing analysis, however, one can never be certain that the true WCET of a piece of code is ever captured (hereafter, we assume “WCET” means “true WCET”). Thus, it is necessary to distinguish between the WCET of a piece of code and its *provisioned execution time* (PET) as obtained via measurements and assumed in schedulability analysis. Note that WCETs are generally unknowable, and a PET may likely be less than the corresponding WCET.

Mitigating this disconnect. Any safety risk introduced by assuming such PETs can be avoided by instead using (*guaranteed*) *execution-time (upper) bounds* (ETBs) obtained under unrealistically pessimistic conditions. For example, any execution speedups due to caches and pipelining might be defined away, bus contention might be over-approximated, etc. While such ETBs might be safe to use as PETs, this would likely be impractical on a multicore machine, as an ETB could easily be many times larger than the corresponding WCET. In fact, system-wide pessimism could be high enough to negate the processing capacity of all “additional” cores [32]. Thus, more reasonable measurement-based PETs are inevitable.

A system provisioned assuming such PETs can have *correct tasks* whose PETs are at least their WCETs, and *faulty tasks* whose PETs are less than their WCETs. Hopefully, PET overruns (faults) should be rare. Moreover, when they do occur, they should be *contained*. The usual approach here is to treat PETs as *execution-time budgets* that are enforced by the operating system (OS). Such an approach can ensure the following desirable property.

P1 The response times of correct tasks, as derived using PETs, are not increased by a PET overrun of a faulty task.

Task-level budgeting is not enough. Unfortunately, due to various realities of real systems, task-level budget management alone is an incomplete solution to the timing/schedulability disconnect. This paper is directed at providing a deep look at one such reality: *the need to support locking protocols for arbitrating accesses to shared resources*. In this setting, we actually care about various different PETs, WCETs, and ETBs. For example, in addition to task-level PETs, PETs are needed for individual critical sections (CSs), and various locking-protocol and OS code sequences. To avoid confusion, we will add a qualifying prefix when referring to non-task-level terms – e.g., “CS PET” refers to the PET of a CS, while “PET” (without qualification) refers to a task-level PET. When locking protocols are introduced, CS PETs (and also various protocol- and OS-related PETs) are used to determine blocking times when tasks access shared resources. *Incorrect blocking-time estimates due to inaccurate PETs can completely compromise schedulability guarantees.*

Overrun-resilient locking protocols. To address this issue, we propose in this paper the notion of an *overrun-resilient locking protocol*. In addition to not causing a violation of P1, such a protocol must also uphold its CS-level variant:

P2 The response times of correct tasks, as derived using PETs, are not increased by a CS PET overrun of a faulty task.

It’s not so easy. The obvious solution to satisfying P1 and P2 is to assign budgets to both tasks and CSs as given by their respective PETs. The main new complication that arises when doing this is the need to abort the CS of some task when one of these budgets is overrun. Such aborts should be avoided if possible, but they cannot be entirely precluded.

■ **Table 1** Properties satisfied by prior work. (“NC” means the work does not consider how to satisfy the specified property. As explained in Sec. 9, some of these “NC” entries can be changed to “Y” at the expense of very pessimistic provisioning assumptions).

Protocol	Multi-processor	P1	P2	P3	Protocol	Multi-processor	P1	P2	P3
ICSs [30]	N	Y	Y	Y	FMLP [9]	Y	N	N	N
RRP [3]	N	Y	Y	NC	M-BWI [20]	Y	N	N	NC
RACPwP [39]	N	Y	Y	NC	vMPCP [31]	Y	Y	N	NC
SIRAP [6]	N	Y	N	NC	M-BROE [8]	Y	Y	N	NC
OMLP [12]	Y	N	N	N	This Work	Y	Y	Y	Y

For example, a CS budget overrun will necessarily cause a CS abort. The usual approach to aborting work is to execute undo code. Presumably, a PET would have to be associated with such code. What happens if the abort code overruns its PET? Additionally, certain code sequences exist pertaining to lock and budget management for which overruns are similarly problematic. For example, when a CS is aborted, the unlock logic must execute to free the resource. What if the PET associated with this unlock code is overrun? It is not clear how these perplexing “chicken and egg” problems can be addressed. Whatever the solution, an overrun-resilient locking protocol must uphold a third property:

P3 The response times of correct tasks, as derived using PETs, are not increased by the budget-enforcement mechanism.

Related work. Various locking protocols have been proposed in prior work that considers budget overruns. However, no prior work focusing on multiprocessors fully considers properties P1–P3. Relevant prior work is summarized in Tbl. 1 and discussed in detail in Sec. 9.

Contributions. In this paper, we present overrun-resilient multiprocessor locking protocols that satisfy P1–P3. Our contributions are fourfold. First, we introduce the *overrun-resilient flexible multiprocessor locking protocol* (OR-FMLP), an overrun-resilient extension of the spin-based FMLP [9]. Second, we introduce the *overrun-resilient global optimal multiprocessor locking protocol* (OR-OMLP), an overrun-resilient extension of the suspension-based global OMLP [12]. Third, for the important locking use case of coordinating accesses to shared data structures, we propose the concept of an *abortable CS*, which facilitates satisfying P2 and P3. Finally, we present the results of an experimental evaluation of the cost of overrun-resilient locking and its isolation benefits with respect to timing faults.

Both the OR-FMLP and OR-OMLP use a concept called a “forbidden zone” [28] to satisfy P1. A *forbidden zone* (FZ) is a length of time at the end of a job’s task budget during which any lock request will be denied. However, the application of this concept is very different in the two protocols. To circumvent the various chicken-and-egg problems related to P3, ETBs must be used for certain code sequences. As ETBs can be very pessimistic, reliance on them should be minimized. With this in mind, we carefully sift through the various design choices and conclude that in a spinlock like the OR-FMLP, coarse-grained FZs should be used that include both CS execution time and blocking time, while in a suspension-based lock like the OR-OMLP, fine-grained FZs based on CS execution times only are better.

Our notion of an abortable CS requires no undo code when aborting CSs. An abortable CS uses word-based *obstruction-free* [25] software transactional memory (STM) techniques to linearize a CS to a single write instruction. Obstruction-freedom is a type of non-blocking synchronization that must be used with a contention manager to ensure progress under

contention. In our case, the contention manager is a locking protocol. We show that using such a strong contention manager enables significant simplifications in obstruction-free code.

Organization. In the rest of this paper, we provide necessary background information (Sec. 2), delve further into task and CS budget management (Sec. 3), present the OR-FMLP, the OR-OMLP, and the abortable CS concept (Secs. 4–6), present our experimental results (Sec. 7), discuss certain practical implications of our work (Sec. 8), review related work (Sec. 9), and conclude (Sec. 10).

2 System Model and Background

Task model. We consider a system of n implicit-deadline¹ sporadic *tasks* $\tau_1, \tau_2, \dots, \tau_n$ to be scheduled on m identical processors by a global job-level fixed-priority scheduler; we assume global earliest-deadline-first (G-EDF) scheduling, unless stated otherwise. Each task τ_i releases a potentially infinite sequence of *jobs* $J_{i,1} J_{i,2} \dots$ (we omit the job index if it is irrelevant). Each task τ_i has a *period* T_i specifying the minimum spacing between consecutive job releases. Each task has a PET obtained via measurement-based timing analysis.

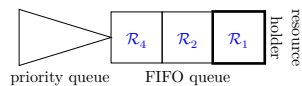
Resource model. We consider a system that has a set $\{\ell_1, \dots, \ell_{n_r}\}$ of serially reusable shared resources. To ensure mutually exclusive resource access, a *locking protocol* must be employed. When a job J_i requires a resource ℓ_k , it *issues* a *request* \mathcal{R} for ℓ_k . \mathcal{R} is *satisfied* as soon as J_i holds ℓ_k , and *completes* when J_i releases ℓ_k . \mathcal{R} is *active* from its issuance to its completion. J_i must wait until \mathcal{R} can be satisfied if it is held by another job. It may do so either by *busy-waiting* (or *spinning*) in a tight loop, thereby wasting processor time, or by being *suspended* by the OS until \mathcal{R} is satisfied. A resource access is called a *critical section* (CS). Each CS has a CS PET obtained via measurement-based timing analysis. We consider non-nested resource requests only. We let Γ_k to denote the set of tasks that share ℓ_k .

Priority inversions. *Priority-inversion blocking* (or *pi-blocking*) occurs when a job is delayed and this delay cannot be attributed to higher-priority demand for processing time. Under a given real-time locking protocol, a job may experience pi-blocking each time it requests a resource – this is called *request blocking*. In addition, a preemptive ready job may experience pi-blocking due to the non-preemptive execution of lower-priority jobs – this is called *non-preemptive blocking*. On multiprocessors, the formal definition of pi-blocking actually depends on how schedulability analysis is done. For example, of relevance to suspension-based locks, analysis may be either *suspension-oblivious* (*s-oblivious*) or *suspension-aware* (*s-aware*) [12]. Under s-oblivious analysis, suspension time is *analytically* treated as computation time.

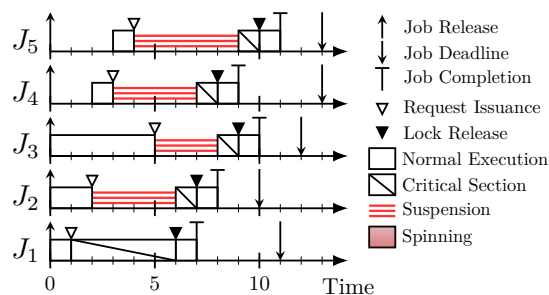
FMLP. Under the FMLP [9], non-preemptive spin locks are used to ensure mutually exclusive resource access.² Each job that is blocked on a resource busy-waits within a FIFO queue.

¹ The results of this paper do not depend on the choice of deadline constraints. Implicit deadlines are assumed for simplicity.

² There are actually two FMLP variants: short (spin-based) and long (suspension-based). We are considering the short variant here.



■ **Figure 1** The global OMLP queue structure for 3 processors.



■ **Figure 2** Jobs issuing requests to the global OMLP with $m = 3$. (The notation in this figure is also used in subsequent figures).

Global OMLP. The global OMLP [12] is a suspension-based locking protocol that has asymptotically optimal pi-blocking under s-oblivious analysis. The global OMLP ensures $O(m)$ pi-blocking by utilizing a dual-queue structure, with an m -element FIFO queue fed into by a priority queue, as shown in Fig. 1. A new request is enqueued in the FIFO queue (resp., priority queue) if there are fewer than (resp., at least) m active requests. When the request at the head of the FIFO queue (i.e., the resource holder) completes, it is dequeued, the next request (if any) in the FIFO queue becomes satisfied, and the highest-priority request (if any) in the priority queue is moved to the tail of the FIFO queue.

► **Example 1.** Fig. 2 shows five jobs that issue requests to the global OMLP with $m = 3$. Fig. 1 shows the global OMLP queues at time 3.5, where J_i 's request is denoted by \mathcal{R}_i . The first three issued requests are enqueued directly in the FIFO queue. Thus, \mathcal{R}_4 is satisfied before \mathcal{R}_3 , although J_4 has lower priority than J_3 . Since the FIFO queue is full, \mathcal{R}_5 and \mathcal{R}_3 are enqueued in the priority queue upon issuance. When \mathcal{R}_1 completes at time 6, \mathcal{R}_2 becomes satisfied, and \mathcal{R}_3 is moved from the priority queue to the FIFO queue, as J_3 has higher priority than J_5 . Thus, \mathcal{R}_3 is satisfied before \mathcal{R}_5 , despite being issued later.

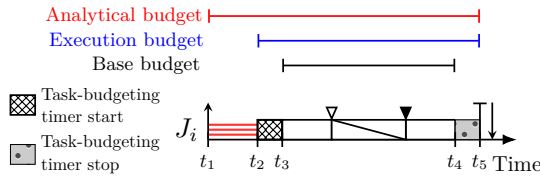
For ease of notation, we henceforth assume that all jobs of each task include one request for the same resource, and this request is preceded and followed by non-resource-accessing code. This assumption enables us to refer to a job's CS without ambiguity. We stress that we are making this assumption only for simplicity; none of our results actually depend on it.

3 Budget Management

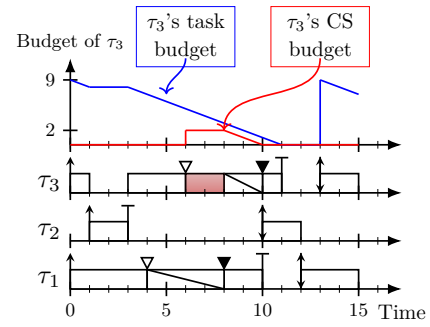
Execution budgets that are enforced at runtime are obtained by inflating *base budgets* that pertain to the execution of task code by adding certain overheads. Additional overheads are then added to obtain *analytical budgets* that are used in schedulability analysis. In this section, we provide details concerning these budgeting notions and relevant overheads.

Base budgets. We define the *base task budget* (resp., *base CS budget*) of a task τ_i (resp., τ_i 's CS), denoted by C_i^b (resp., L_i^b), as its PET (resp., CS PET).

What is and is not included in base budgets. Timing analysis is applied to determine relevant PETs for a task independently of the task system that contains it. As lock-related *blocking times* are system-dependent, we assume that they are not included in base task budgets. The *lock/unlock logic* of a suspension-based lock is executed in the OS and hence



■ **Figure 3** Illustration of base, execution, and analytical budgets.



■ **Figure 4** Budget consumption and replenishment. Overheads/delays other than spinning are omitted to avoid clutter.

would not be included in base task budgets. In contrast, for a spinlock, this logic executes at user level. However, as seen later, to satisfy P3, we must take special care in dealing with this logic, so we assume it is not included in base task budgets.

When measurement-based timing analysis is applied, preemptions are notoriously difficult to deal with due to difficulties in predicting cache interactions. For this reason, we assume that *CSs are executed non-preemptively* and that their base budgets are determined assuming cold caches. However, we do not preclude preemptions outside of CSs, as long as base task budgets include *cache-related preemption and migration delay (CPMD)*, which is a cost that is incurred by a job to re-establish lost cache affinity after a preemption or migration. As the focus of this paper is not timing analysis, determining valid CPMDs is out of scope.

Timers. To enforce base task and CS budgets, we require the usage of timers supported by the OS, which we call *task-* and *CS-budgeting timers*, respectively. Such a timer *starts* when the relevant entity (an entire job or a CS) starts executing and *stops* when the entity is preempted (not allowed for CSs), aborted (see below), or completes execution. Between starting and stopping, a timer is *active*.

Execution budgets. In reality, timers cannot be started and stopped in zero time. To start a timer, timer-handling code executes in the OS. We assume no knowledge of the exact structure of this code but do require that an ETB is specified for it. When this code executes, we know that the timer starts at some time point, but not precisely when. Stopping a timer is similar. In order to safely police base budgets, we must account for these timer activities. To do so, we instead police adjusted execution budgets as defined next.

The *task execution budget* C_i^e of a task τ_i is obtained by inflating its base task budget C_i^b by adding the worst-case cost of all task- and CS-budgeting timer overheads, as provisioned by their ETBs, that may be incurred by a job of τ_i . The *CS execution budget* L_i^e of τ_i 's CS is similarly obtained by inflating its base CS budget L_i^b by adding the worst-case cost of all CS-budgeting timer overheads, as provisioned by their ETBs, associated with that CS.

In overrun-resilient locking protocols that we propose, these execution budgets are enforced at runtime. Specifically, we set the task- or CS-budgeting timer of a job to *expire* when the corresponding task or CS execution budget is exhausted. A job *overruns* its task/CS execution budget if it does not complete execution before the relevant timer expires.

► **Example 2.** Consider job J_i in Fig. 3. (We consider analytical budgets later.) Starting (resp., stopping) J_i 's task-budgeting timer entails executing OS code during $[t_2, t_3)$ (resp.,

$[t_4, t_5)$). Thus, J_i 's task execution budget is derived by inflating its base task budget by $(t_3 - t_2) + (t_5 - t_4)$ units.

We assume that execution budgets are managed via the following rules.

Consumption Rule: A job J_i consumes its task (resp., CS) execution budget at the rate of one execution unit per unit of time when its task-budgeting (resp., CS-budgeting) timer is active.

Since a task- or CS-budgeting timer expires when the corresponding execution budget is exhausted, a job cannot consume that execution budget when it is 0.

Replenishment Rule: J_i 's task execution budget is set to C_i^e when it is released. J_i 's CS execution budget is set to L_i^e when it issues a lock request.

► **Example 3.** Fig. 4 depicts three G-EDF-scheduled tasks on two processors. τ_1 and τ_3 use resource ℓ_1 , which is protected by a spinlock. τ_3 's task (resp., CS) execution budget is 9.0 units (resp., 2.0 units). At time 0, $J_{3,1}$ is scheduled and its task-budgeting timer starts. $J_{3,1}$ consumes 1.0 unit of its task execution budget within the time interval $[0, 1)$ during which its task-budgeting timer is active. $J_{3,1}$ is preempted by $J_{2,1}$ at time 1, causing its task-budgeting timer to stop. Thus, $J_{3,1}$'s task execution budget remains the same during the time interval $[1, 3)$. At time 3, $J_{3,1}$ is scheduled again and it continues executing until completing at time 11, consuming 8.0 units of its task execution budget.

$J_{3,1}$ issues a request for ℓ_1 at time 6 that is satisfied at time 8 (when its CS-budgeting timer starts) and completes at time 10 (after consuming its entire CS execution budget).

Analytical budgets. Some overhead/delay sources do not cause task or CS execution budget to be consumed. However, such sources can impact schedulability. We define the *analytical task budget* of task τ_i , denoted C_i^a , by inflating its task execution budget to account for all overheads/delays. We define the *analytical CS budget* of τ_i 's CS, denoted L_i^a , by inflating its CS execution budget to account for all overheads/delays affecting that CS.

► **Example 2 (Cont'd).** J_i in Fig. 3 suffers pi-blocking during the time interval $[t_1, t_2)$ due to a non-preemptively executing lower-priority job. Since J_i is not scheduled during $[t_1, t_2)$, its execution budget does not decrease during this interval. However, J_i may miss its deadline due to the delay caused by this pi-blocking. Thus, J_i 's analytical task budget is derived by inflating its task execution budget by $t_2 - t_1$.

Overheads/Delays. We consider the following overheads/delays that are either locking- or timer-related overheads. We summarize the overheads/delays that affect base and execution task and CS budgets under the OR-FMLP and OR-OMLP in Tbl. 2 and all introduced notation in Tbl. 3. Note that we require ETBs of these overheads to avoid introducing ‘‘chicken and egg’’ problems in satisfying P3, as discussed in Sec. 1.

- (i) *Budgeting-timer overheads.* We denote the ETBs of starting, stopping, and expiring a budgeting-timer by Δ_{tb} , Δ_{te} , and Δ_{tt} , respectively. Since we focus on timer overheads that are due to a CS execution, accounting for overheads due to starting/stopping a task-budgeting timer for resuming/suspending a job's non-CS code is out of scope.
- (ii) *Locking and unlocking overheads.* We denote the ETBs of executing the lock and unlock logic (for both spinlocks and suspension-based locks) by Δ_{lock} and Δ_{unlock} , respectively.
- (iii) *Request blocking.* We let B_i denote a bound on request blocking incurred by τ_i 's request.

10:8 Overrun-Resilient Multiprocessor Real-Time Locking

■ **Table 2** OR-FMLP and OR-OMLP overhead impact.

Overheads	Base task budgets		Task exec. budgets	
	OR-FMLP	OR-OMLP	OR-FMLP	OR-OMLP
Budgeting-timer	×	×	✓	✓
Locking & unlocking	×	×	✓	✓
Request blocking	×	×	✓	×
Non-preemptive blocking	×	×	×	×

(iv) *Non-preemptive blocking.* We let NPB_i denote a bound on non-preemptive blocking incurred by τ_i .

Accounting for these overheads in *analytical* budgets is a well-researched topic [10]. We detail the required overhead inflation in task and CS *execution* budgets under the OR-FMLP and OR-OMLP in Secs. 4 and 5, respectively.

Simplifying assumptions. In order to focus only on those overheads/delays of direct relevance to overrun-resilient locking and to simplify the description of the OR-FMLP and OR-OMLP, we make the following assumptions.

A1 ETBs of all overheads are known.

A2 The cost of aborting a CS is included in the ETB of expiring the CS-budgeting timer.

A3 All overheads/delays other than task- and CS-budgeting timer overheads, locking and unlocking overheads, and request and non-preemptive blocking are negligible.

We discuss how A1 and A3 can be relaxed in Sec. 8 and how to support A2 in Sec. 6.

Refining P1–P3. Properties P1–P3 can be ensured by maintaining the following properties.

P1.1 If a job’s task execution budget expires, then it has no active request (to satisfy P1).

P2.1 If the CS execution budget of a CS expires, then the CS is aborted without corrupting shared-resource state (to satisfy P2).

P3.1 Execution-time variances in executing timer-handling and lock/unlock logic cannot cause task and CS execution budgets to be exceeded (to satisfy P3).

We show how to satisfy P1.1 and P3.1 in the OR-FMLP and OR-OMLP in Secs. 4 and 5. We also show how to satisfy P2.1 in Sec. 6 for the case of shared data structures. In order to focus on P1.1 and P3.1 for now, we make the following assumption.

A4 Property P2.1 is satisfied.

4 OR-FMLP

In this section, we introduce the *overrun-resilient flexible multiprocessor locking protocol (OR-FMLP)*, an extension of the FMLP [9] that achieves overrun resiliency by enforcing task and CS execution budgets. Like the FMLP, a job is non-preemptive when executing the OR-FMLP (while both spinning and executing its CS). The OR-FMLP satisfies P1.1 by using a previously proposed concept called a *forbidden zone (FZ)* [27] that aids in budget enforcement – in fact, the OR-FMLP is very similar to a protocol called the “Skip Protocol” presented in [27]. In our setting, however, much care is required in deriving execution budgets so that “chicken and egg” problems are avoided. The goal of avoiding such problems has a major bearing on the overall lock design and its analysis.

■ **Table 3** Notation summary.

Symbol	Meaning	Symbol	Meaning
n	Number of tasks	L_i^a	Analytical CS budget of τ_i 's CS
m	Number of processors	Δ_{tb}	ETB of starting overhead for a task- or CS-budgeting timer
τ_i	i^{th} task	Δ_{te}	ETB of stopping overhead for a task- or CS-budgeting timer
$J_{i,j}$	j^{th} job of τ_i	Δ_{tt}	ETB of expiring overhead for a task- or CS-budgeting timer
T_i	Period of τ_i	Δ_{lock}	ETB of locking overhead
ℓ_k	k^{th} shared resource	Δ_{unlock}	ETB of unlocking overhead
C_i^b	Base task budget of τ_i	Δ_{abort}	ETB of overhead for aborting a request
C_i^e	Task execution budget of τ_i	B_i	Maximum request blocking time of τ_i
C_i^a	Analytical task budget of τ_i	NPB_i	Maximum non-preemptive blocking time of τ_i
L_i^b	Base CS budget of τ_i 's CS	Γ_k	Set of tasks that shares a resource ℓ_k
L_i^e	CS execution budget of τ_i 's CS	f_i	Forbidden-zone length for J_i

Design goal. Spinlocks provide mutual exclusion without OS support, eliminating system-call overheads. While some timer-related OS support is needed, our overriding design goal is nonetheless the following.

G1 Minimize the number of the OS invocations.

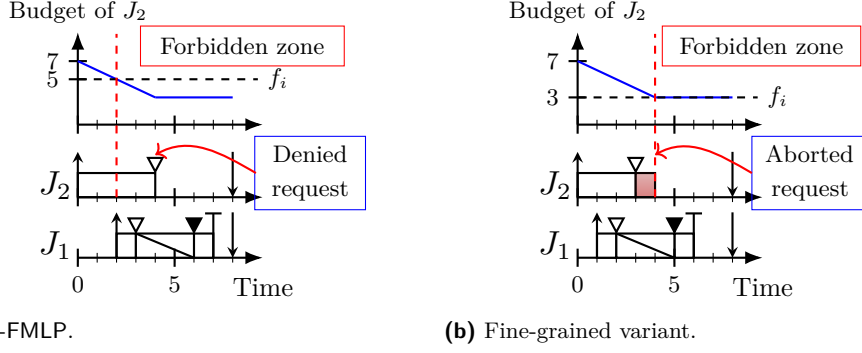
Managing CS-budgeting timers. To prevent CS execution budget overruns, OS invocations are needed, contrary to G1, to manage CS-budgeting timers. It is perhaps theoretically possible to avoid using such timers by having the CS itself repeatedly monitor the CS execution budget remaining, but such an approach would have very high overhead.

Satisfying P1.1. We satisfy P1.1 by employing FZs, as mentioned above. When a job is allocated its task execution budget, a portion of that budget at the end constitutes its FZ. A job is not allowed to issue a resource request during its FZ. The length of J_i 's FZ, denoted by f_i , is the maximum task execution budget of J_i that can be consumed when a request of J_i is active. Under the OR-FMLP, this task execution budget consumption includes both its CS length and spinning time. The OR-FMLP adds an additional “FZ check” prior to performing the locking logic of the FMLP. This check, which is assumed to be part of the locking overhead of the OR-FMLP, can be implemented entirely in user space by having the OS record the current time as given by the local timestamp counter (TSC) in a shared control page whenever a job begins or resumes execution. Using this recorded value and the current local TSC value, a job can determine whether it is in its FZ.

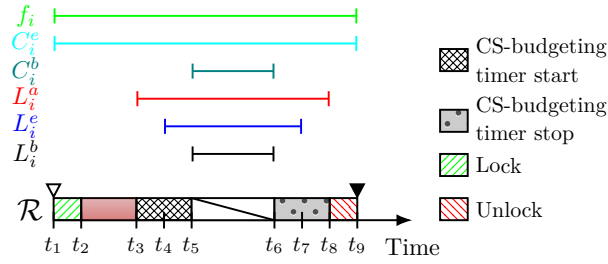
► **Example 4.** Fig. 5(a) depicts two jobs that issue requests to the OR-FMLP for resource ℓ_1 . Assume that J_2 's task execution budget is 7.0 units and its CS execution budget is 2.0 units. J_2 could potentially be blocked by J_1 for 3.0 time units, the length of J_1 's CS execution. J_2 enters its FZ at time 2 as it does not have sufficient task execution budget to spin for 3.0 time units and then execute its CS for 2.0 time units. Thus, its request is denied at time 4.

Satisfying P3.1. To satisfy P3.1, we derive task and CS execution budgets by accounting for all lock- and timer-related overheads/delays, given in Tbl. 2, using their ETBs. Before

10:10 Overrun-Resilient Multiprocessor Real-Time Locking



■ **Figure 5** Illustration of FZs. Overheads/delays other than spinning are omitted to avoid clutter.



■ **Figure 6** OR-FMLP request timeline with overheads included.

deriving these terms, we first give the rules of the OR-FMLP.

OR-FMLP Rules. We assume the following properties, which we justify later.

- B1** The execution and analytical budgets of all tasks and CSs have been determined.
- B2** A job's FZ length can be derived from task/CS base, execution, and analytical budgets. When a job J_i attempts to issue a request \mathcal{R} for a resource ℓ_k , it proceeds according to the following rules (J_i is non-preemptive while executing according to these rules).
- F1** \mathcal{R} is issued only if J_i 's remaining task execution budget is at least f_i ; otherwise, \mathcal{R} is denied. Issued requests spin (if necessary) in per-resource FIFO queues until satisfied. (Policies for handling denied or aborted requests are an application-level concern.)
- F2** When \mathcal{R} is satisfied, J_i 's CS-budgting timer is set to expire L_i^e time units in the future.
- F3** When J_i 's CS completes, J_i 's CS-budgting timer is stopped and J_i releases ℓ_k . If the CS-budgting timer expires prior to CS completion, then ℓ_k is released (i.e., J_i 's CS is aborted, as allowed by Assumption A4).

Addressing B1 and B2. We now address Properties B1 and B2. Fig. 6 depicts the execution of a request \mathcal{R} of a job J_i , with overheads included, during the time interval from \mathcal{R} 's issuance until its completion. During this interval, J_i issues \mathcal{R} by inserting \mathcal{R} into the FIFO spin-queue during $[t_1, t_2)$, spins (if required) during $[t_2, t_3)$, starts its CS-budgting timer during $[t_3, t_5)$, executes its CS during $[t_5, t_6)$, stops its CS-budgting timer during $[t_6, t_8)$, and unlocks its acquired resource during $[t_8, t_9)$. Using this figure as a reference, we now derive the various terms mentioned in B1 and B2.

CS execution budget. We derive L_i^e by inflating L_i^b to account for its CS execution budget consumption due to CS-budgting timer overheads. Since J_i 's CS executes non-preemptively

under the OR-FMLP, J_i incurs CS-budgeting timer overheads only when its CS starts and completes execution. However, the CS-budgeting timer can start or stop at an arbitrary time point within the OS's timer-handling code, as shown by times t_4 and t_7 in Fig. 6, respectively. Since $(t_5 - t_4) \leq \Delta_{tb}$ and $(t_7 - t_6) \leq \Delta_{te}$, J_i 's CS execution budget is consumed by at most $\Delta_{tb} + \Delta_{te}$ units due to starting and stopping its CS-budgeting timer. Expiring the CS-budgeting timer does not consume any CS execution budget because it occurs only when the CS execution budget is fully consumed. Thus, we have

$$L_i^e = L_i^b + \Delta_{tb} + \Delta_{te}. \quad (1)$$

Analytical CS budget. The above derivation of L_i^e pessimistically assumes that t_4 (resp., t_7) is close to t_3 (resp., t_8). In reality, t_4 (resp., t_7) could instead be close to t_5 (resp., t_6), implying that we must inflate again for timer overheads in determining the analytical CS budget. With this in mind, we derive L_i^a , represented by $[t_3, t_8]$ in Fig. 6, by inflating L_i^e to account for its task execution budget consumption due to CS-budgeting timer overheads. J_i 's task execution budget is consumed by at most L_i^e units during $[t_4, t_7]$. Before (resp., after) the CS-budgeting timer actually starts (resp., stops), the timer-handling code may execute for at most Δ_{tb} (resp., Δ_{te}) time units during $[t_3, t_4]$ (resp., $[t_7, t_8]$). If the CS-budgeting timer of J_i expires, then the expiration and CS abort take at most Δ_{tt} time units (by Assumption A2). Since the timer stop and expiration cannot both occur for a CS, we have

$$L_i^a = L_i^e + \Delta_{tb} + \max(\Delta_{te}, \Delta_{tt}). \quad (2)$$

Request blocking time. Under the FMLP, a request \mathcal{R} for a resource ℓ_k by a job J_i can be blocked by at most $m - 1$ requests for ℓ_k by other jobs. A request \mathcal{R}' by a job J_j that blocks \mathcal{R} can do so for the entire duration when \mathcal{R}' is satisfied. This duration includes the time needed for \mathcal{R}' to (i) start its the CS-budgeting timer, (ii) execute its CS, (iii) stop/expire its CS-budgeting timer, and then (iv) unlock ℓ_k . This time interval is analogous to $[t_3, t_9]$ for J_j in Fig. 6. L_j^a upper bounds the total time for (i)–(iii) and Δ_{unlock} upper bounds the time for (iv). It follows that

$$B_i = \sum_{m-1 \text{ largest in } \Gamma_k} (L_j^a + \Delta_{unlock}). \quad (3)$$

FZ length. J_i 's FZ length, f_i , is the maximum task execution budget of J_i that can be consumed during the time interval when its request \mathcal{R} is active. This time interval corresponds to $[t_1, t_9]$ in Fig. 6. J_i issues \mathcal{R} during $[t_1, t_2]$, which takes at most Δ_{lock} time units. It then busy-waits for at most B_i time units during $[t_2, t_3]$. It subsequently executes its CS and timer-handling code for at most L_i^a time units during $[t_3, t_8]$ and then unlocks ℓ_k during $[t_8, t_9]$, which requires at most Δ_{unlock} time units. Therefore,

$$f_i = B_i + \Delta_{lock} + L_i^a + \Delta_{unlock}. \quad (4)$$

Task execution budget. We derive J_i 's task execution budget by inflating its base task budget to account for spinning time, locking and unlocking overheads, and task- and CS-budgeting timer overheads incurred when its request \mathcal{R} is active (see Tbl. 2). These overheads/delays correspond to all of $[t_1, t_9]$ except $[t_5, t_6]$ in Fig. 6. Since f_i (resp., L_i^b) corresponds to $[t_1, t_9]$ (resp., $[t_5, t_6]$), J_i 's task execution budget is

$$C_i^e = C_i^b + f_i - L_i^b.$$

Analytical task budget. J_i 's analytical task budget is obtained by inflating its task execution budget to account for non-preemptive blocking and a potential task-budgeting timer expiration. Expiring the task-budgeting timer takes at most Δ_{tt} time units. Because jobs invoke the OR-FMLP non-preemptively, a newly released job may be blocked by lower-priority jobs for the duration of m CSs (inflated to include overheads). Reasoning similarly to (3), we have

$$NPB_i = \sum_{m \text{ largest}} (L_j^a + \Delta_{unlock}).$$

Therefore, J_i 's analytical task budget is

$$C_i^a = C_i^e + NPB_i + \Delta_{tt}.$$

Fine-grained FZs and why they are problematic. FZs were originally proposed to be policed upon CS entry [27], but here we have policed them in a more coarse-grained way by also including spinning time. We made this choice to avoid interactions with the OS, per Goal G1, to maintain the use of simple user-level synchronization code, and to reduce the length of code sequences that require ETBs. Here we briefly explore the fine-grained choice of defining FZ lengths based on CS execution times only.

The main advantage of the fine-grained approach is that FZ lengths are shorter. However, now a job may exhaust its task execution budget *while executing within the locking protocol*. In this case, its request must be extracted from the FIFO spin-queue. Letting Δ_{abort} denote that time required to do this, it can be shown that (4) can be replaced by

$$f_i = \max(L_{max}^a + \Delta_{unlock}, \Delta_{abort}). \quad (5)$$

► **Example 5.** Fig. 5(b) depicts two jobs that issue requests for resource ℓ_1 . At time 3, J_2 issues a request \mathcal{R} for ℓ_1 . Assume that J_2 's task and CS execution budgets are 7.0 and 2.0 units, respectively, and extracting \mathcal{R} from the FIFO spin-queue requires 3.0 time units. Then, J_2 's fine-grained FZ length is $\max\{2, 3\} = 3$. Thus, J_2 reaches its FZ at time 4 after consuming 4.0 units of its task execution budget.

Significant prior research has been directed at *abortable spinlocks* that allow requests to be aborted [1, 2, 29, 33, 38, 44]. Two approaches have been investigated in designing such locks. The first approach aborts requests “lazily” by setting a removal flag [2, 33, 38, 44]. Proper request removal is performed later by another job whose resource request is pending or satisfied. This removal requires $O(m^2)$ time [33], which can significantly increase FZ lengths. In the second approach, an aborted request is removed immediately [1, 29]. In existing algorithms, such a removal requires $O(\min(m, \log n))$ time complexity or worse. Moreover, abortable spinlocks require complicated lock/unlock/abort logic. This would significantly increase the ETBs associated with that logic. In contrast, a simple ticket lock can be used in the course-grained variant, thus reducing the length of code sequences requiring ETBs.

5 OR-OMLP

In this section, we introduce the *overrun-resilient $O(m)$ locking protocol (OR-OMLP)*. The original OMLP executes CSs preemptively and uses *priority inheritance* [37] as a progress mechanism to ensure that if a job waiting to access a resource ℓ_k is among the m highest-priority jobs, then the currently satisfied request for ℓ_k is scheduled. The OR-OMLP executes CSs non-preemptively (for the timing-analysis-related reasons discussed in Sec. 3) but retains priority inheritance as a progress mechanism. Priority inheritance is still needed to ensure that

when a (non-preemptive) CS ends, the next queued request will be satisfied if it is blocking a job whose priority is among the top m . Non-preemptive CSs do not alter the OMLP's request-blocking bounds, but introduce non-preemptive blocking, which the OMLP avoids.

Similar to the OR-FMLP, the OR-OMLP uses FZs and ETBs of lock- and timer-related overheads to satisfy P1.1 and P3.1, respectively, but here, FZs are fine-grained (i.e., policed on CS entry). Having already seen how these basic mechanisms work in the context of the OR-FMLP, we proceed directly to defining the rules of the OR-OMLP.

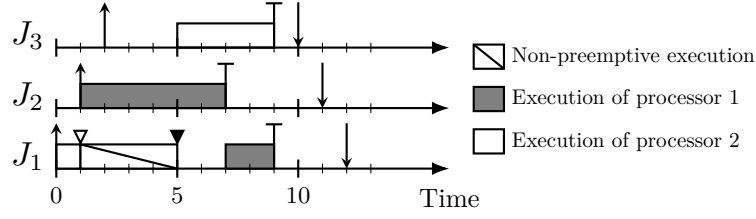
OR-OMLP rules. Our description of the OR-OMLP focuses on a single resource ℓ_k , for which there are two queues, an m -element FIFO queue FQ and a priority queue PQ, as shown in Fig. 1. When a job J_i attempts to issue a request \mathcal{R} for ℓ_k , it follows the rules below, which are specified assuming that B1 and B2 in Sec. 4 hold, and that a job's task-budgeting timer starts (resp., stops) when it begins (resp., ceases) to execute.

- O1** \mathcal{R} is issued only if J_i 's remaining task execution budget is at least f_i ; otherwise, \mathcal{R} is denied. If not denied, \mathcal{R} is enqueued in FQ if fewer than m requests for ℓ_k are already active; otherwise, it is added to PQ.
- O2** All queued jobs except the job at the head of FQ are suspended. The job at the head of FQ inherits the priority of the highest-priority job in FQ or PQ.
- O3** If \mathcal{R} becomes the head of FQ at time t , then it is satisfied and J_i becomes eligible to be scheduled at time t (this depends on its perhaps-inherited priority). If J_j 's request was the head of FQ before time t and J_i is among the m highest-priority jobs at time t but cannot preempt the lowest-priority scheduled job due to non-preemptivity, then J_i preempts J_j (even if J_j is one of the top m priority jobs). Once scheduled, J_i 's CS-budgeting timer is set to expire L_i^c time units in the future, and J_i executes its CS non-preemptively.
- O4** When J_i 's CS completes, its CS-budgeting timer is stopped, J_i releases ℓ_k , and J_i becomes preemptive. If instead its CS-budgeting timer expires prior to its CS completion, then ℓ_k is released (i.e., \mathcal{R} is aborted, as allowed by Assumption A4). In either case, \mathcal{R} is dequeued from FQ and the highest-priority request from PQ is moved to the tail of FQ.

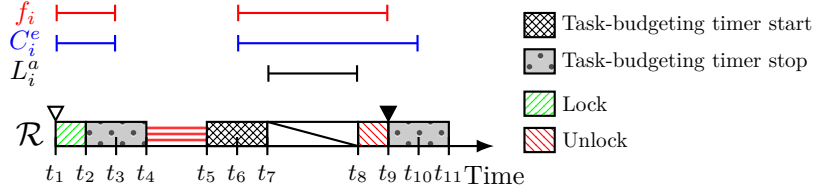
Lazy preemptions. In addition to the above rules, we enact preemptions (except the preemption mentioned by Rule O3) *lazily* by delaying any preemption until the lowest-priority scheduled job becomes preemptable instead of preempting the first-available lower-priority job [10, §3.3.3]. Lazy preemptions prevent a job from incurring repeated pi-blocking each time a higher-priority job is released when the scheduler implementation is *link-based* [10, §3.3.3], which we assume.

► **Example 6.** Lazy preemption is depicted in Fig. 7. Three jobs are scheduled on two processors under link-based G-EDF. When J_3 is released at time 2, the scheduler links J_3 to processor 1, which is executing the lowest-priority job among both processors. Since J_1 is scheduled on processor 1 and is nonpreemptive at time 2, J_3 is not scheduled. At time 5, J_1 becomes preemptive again, and J_3 , the job linked to processor 1, is scheduled on processor 1. Further details concerning lazy preemptions and link-based scheduling are given in an online appendix [41].

Addressing B1 and B2. We deal with these properties similarly as for the OR-FMLP. Fig. 8 depicts the execution of a request \mathcal{R} of a job J_i , with overheads included. We omit CS-budgeting timers in Fig. 8 as their accounting is the same as for the OR-FMLP, but we



■ **Figure 7** Example illustrating lazy preemptions and link-based scheduling.



■ **Figure 8** Timeline of an active request under the OR-OMLP.

include task-budgeting timers as they are relevant to suspension-based locks. J_i 's lock request is issued by enqueueing it in the relevant queue during $[t_1, t_2)$. If ℓ_k is held by another job, then J_i suspends. Before suspending J_i at t_4 , J_i 's task-budgeting timer is stopped during $[t_2, t_4)$. J_i 's task-budgeting timer is started during $[t_5, t_6)$ after it is scheduled again upon satisfaction of \mathcal{R} at t_5 . During $[t_7, t_8)$, J_i 's CS-budgeting timer starts, its CS executes, and then its CS-budgeting timer stops. J_i unlocks ℓ_k during $[t_8, t_9)$. If J_i is preempted due to Rule O3 after it unlocks ℓ_k , then its task-budgeting timer is stopped during $[t_9, t_{11})$. Using this figure as reference, we now derive the various terms mentioned in B1 and B2.

CS execution and analytical budgets. Since CSs execute non-preemptively, the analysis of CS execution and analytical budgets is the same as under the OR-FMLP. Thus, J_i 's CS execution budget and analytical CS budgets are determined by (1) and (2), respectively.

Task execution budget. Locking and unlocking overheads during $[t_1, t_2)$ and $[t_8, t_9)$, respectively, occur when J_i 's task-budgeting timer is active and thus consume at most Δ_{lock} and Δ_{unlock} units, respectively, of J_i 's task execution budget. J_i 's task-budgeting timer actually starts (resp., stops) at an arbitrary time point t_6 (resp., t_3 and t_{10}) within the OS's timer-handling code. Thus, $(t_3 - t_2) + (t_{10} - t_9) \leq 2\Delta_{te}$ (resp., $(t_7 - t_6) \leq \Delta_{tb}$) units are consumed from J_i 's task execution budget when J_i is suspended/preempted (resp., resumed). By the FZ check in Rule O1, J_i 's task-budgeting timer cannot expire when \mathcal{R} is active. The CS-budgeting timer overheads are accounted for in L_i^a , and are at most $L_i^a - L_i^b$. Thus, J_i 's task execution budget is

$$C_i^e = C_i^b + \Delta_{lock} + \Delta_{unlock} + \Delta_{tb} + 2\Delta_{te} + L_i^a - L_i^b.$$

FZ length. f_i is given by the maximum task execution budget of J_i that can be consumed during the time interval when \mathcal{R} is active. J_i consumes its task execution budget throughout all of $[t_1, t_9)$ except within $[t_3, t_6)$. Therefore,

$$f_i = \Delta_{lock} + \Delta_{unlock} + \Delta_{tb} + \Delta_{te} + L_i^a.$$

Request blocking. Under the OMLP, a job J_i can be pi-blocked by the length of at most $2(m-1)$ requests for ℓ_k [12]. This result hinges on a progress mechanism, which ensures the progress of the job J_j holding ℓ_k whenever J_i is request-blocked. Under the OR-OMLP, Rule O3 and priority inheritance ensure the same progress property. When J_j becomes the head of FQ, Rule O3 ensures that it is scheduled if its (perhaps-inherited) priority is one of the top m despite any non-preemptive execution of lower-priority jobs. This may cause non-preemptive blocking for the previous resource holder (if any), which we discuss later. Priority inheritance ensures that J_j can be scheduled when its priority is raised because of J_i 's request issuance. (We give a formal proof in an online appendix [41].)

By preserving the same progress property as the OMLP, the OR-OMLP has the same request-blocking bounds as the OMLP. A request \mathcal{R}' by J_j can pi-block J_i for the duration in which \mathcal{R}' is satisfied, which is analogous to the time interval $[t_5, t_9]$ in Fig. 8. This duration includes the time needed for \mathcal{R}' to (i) start its task-budgeting timer, (ii) start its CS-budgeting timer, (iii) execute its CS, (iv) stop its CS-budgeting timer, and (v) unlock ℓ_k . Δ_{tb} upper bounds (i), L_i^a upper bounds (ii)–(iv), and Δ_{unlock} upper bounds (v). Additionally, J_j can be preempted before the next job holding ℓ_k can be scheduled. This causes J_j 's task-budgeting timer to stop during $[t_9, t_{11})$, which takes at most Δ_{te} time. Therefore,

$$B_i = 2 \cdot (m-1) \cdot (\Delta_{tb} + \Delta_{unlock} + \Delta_{te} + \max_{\tau_j \in \Gamma_k} \{L_j^a\}). \quad (6)$$

Non-preemptive blocking. With lazy preemptions, J_i can incur non-preemptive blocking when it releases ℓ_k (due to Rule O3). (We give an example of the latter case in an online appendix [41].) Note that a job can be pi-blocked when a resource is released even under the OMLP if there is a task with non-preemptive sections [10, §3.3.3]. However, such pi-blocking can be analytically treated the same as pi-blocking incurred upon job release by considering the remaining portion of the job as a new job. Each job release can cause pi-blocking for the length of at most one CS [10, §3.3.3]. Reasoning as above for (6), we have

$$NPB_i = 2 \cdot (\Delta_{tb} + \Delta_{unlock} + \Delta_{te} + \max\{L_j^a\}).$$

Analytical task budget. We derive C_i^a by inflating C_i^e to account for request blocking time B_i , non-preemptive blocking time NPB_i , task-budgeting timer expiration overhead Δ_{tt} , and task-budgeting timer starting/stopping overheads during $[t_3, t_4)$, $[t_5, t_6)$, and $[t_{10}, t_{11})$. Since $(t_4 - t_3) + (t_{11} - t_{10}) \leq 2\Delta_{te}$ and $(t_6 - t_5) \leq \Delta_{tb}$, we have

$$C_i^a = C_i^e + B_i + NPB_i + \Delta_{tb} + 2\Delta_{te} + \Delta_{tt}.$$

Coarse-grained FZs. Is it possible to have a OR-OMLP variant with coarse-grained FZs like the OR-FMLP? Such a variant would actually be quite tricky to implement due to the need to track task budget consumption by waiting jobs. A waiting job's task budget should be consumed only when it is pi-blocked, and under s-oblivious analysis, not all suspension time “counts” as pi-blocking time [12]. This nuance greatly complicates budget tracking.

6 Abortable Critical Sections

In this section, we introduce *abortable* CSs, which enable operations on shared data structures to be aborted without undo code. Abortable CSs are inspired by word-based obstruction-free STM, which *linearizes* multiple operations to a single instruction, but can only ensure progress in the presence of a contention manager. By executing instructions in CSs, the

locking protocol serves as a strong contention manager, allowing us to simplify and address issues in prior obstruction-free techniques.

Undo code problem. The following example shows the necessity of undo code when an ordinary CS is aborted.

► **Example 7.** Consider the MODIFY procedure in Alg. 1, which updates a two-word buffer $M[1..2]$ by incrementing each $M[i]$ by $M[1]$'s value. If the procedure is aborted after completing the first **for**-loop iteration, then the buffer is left in an inconstant state. In order to restore M to a valid state, undo code would need to set $M[1]$ to its old value.

While the undo code above is simple, such code can be much more complicated for operations that make many changes to object state. Undo code also needs to be provisioned using its ETB to satisfy P3.1, which can be as pessimistic as the ETB of the CS itself.

Prior work on versioning techniques. Prior work on versioning techniques attempt to obviate undo code through various means, but they all have unfortunate limitations in our context. Interruptible CSs (ICSs [30]) use the idea of a *continuation* [42] to eliminate undo code by appending memory modifications to a log, which will be applied before the next CS entry. Unfortunately, ICSs can force short CSs to apply the memory modifications of long CSs. When CSs only modify memory, each CS length may increase by the length of the longest CS in the system. Object-based obstruction-free STMs [21, 26, 35] do not face this issue, but may require coarse-grained copies of an entire data structure when only small modifications are performed. Word-based variants [22–24] eliminate the need for coarse-grained copying, but require a garbage collector. Other protocols such as TL2 [19] fix both the problems of word-based STMs and continuations, but can require a lengthy clean-up process on abort.

Our abortable CS concept leverages locks as a strong contention manager. It also addresses the issue present in ICSs, and word-based STMs' reliance on garbage collectors without requiring a lengthy clean-up process like TL2. We now explain this concept by showing how to convert MODIFY into an abortable version, ABORTABLEMODIFY, also given in Alg. 1. We first describe the data structures involved.

Data structures. We represent each $M[i]$ using the structure shown in Fig. 9(a) and associate a CS, e.g., an invocation of ABORTABLEMODIFY, with a *transaction record* as defined in Fig. 9(b). The fields $M[i].old$ and $M[i].new$ contain the *valid* value of $M[i]$ – i.e., the value written by the last unabort request involving a write to $M[i]$ – before and after reaching the linearization point, respectively.³ We modify only $M[i].new$ within a CS before it reaches its linearization point. The *txn* field of $M[i]$ is a pointer to a transaction record, which is set when $M[i]$ is updated in ABORTABLEMODIFY.

The *rc1* and *rc2* fields in a transaction record count the number of $M[i]$ structures that point to that record, which are used to determine when the record is no longer in use, as discussed later. The *done* field indicates whether the CS corresponding to the record has successfully been completed or not. The computation of a CS linearizes to a single write that sets its transaction record's *done* field to *true*. Thus, we maintain the following invariant.

I $M[i].old$ contains $M[i]$'s valid value if $M[i].txn.done$ is *false* or $M[i].txn$ is NULL. $M[i].new$ contains $M[i]$'s valid value if $M[i].txn.done$ is *true*.

³ $M[i]$ is a pointer, so technically we should use notation like $M[i] \rightarrow old$ to indicate that it must be dereferenced before accessing the *old* field. We have opted for simpler notation that is more readable.

Algorithm 1 Example buffer data structure.

Variables: $M[1..2]$: A shared array of words 1: procedure MODIFY(M) 2: $x := M[1]$ 3: for $i \in \{1, 2\}$ do 4: $M[i] := M[i] + x$ 5: end for 6: end procedure Variables: $M[1..2]$: A shared array of type in Fig. 9(a) $data$: Ptr of data structure in Fig. 9(a) txn : Ptr to txn_record in Fig. 9(b) new_txn : Ptr to txn_record in Fig. 9(b) $free_stack$: Stack of free transaction records $free_stack_top$: Ptr to top of $free_stack$ 1: procedure ABORTABLEMODIFY(M) 2: $new_txn := NULL$ 3: if $M[1].txn \neq NULL \wedge M[1].txn.done$ 4: then 5: $x := M[1].new$ 6: else 7: $x := M[1].old$ 8: end if 9: for $i \in \{1, 2\}$ do 10: $txn := M[i].txn$ 11: $data := \&(M[i])$ 12: if $txn \neq NULL$ then 13: if $txn.done$ then	13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41:	$data.old := data.new$ end if $txn.rc1 := txn.rc1 - 1$ if $txn.rc1 = 0$ then $txn.done := false$ $txn.next := free_stack_top$ $free_stack_top := txn$ end if $data.txn := NULL$ $txn.rc2 := txn.rc2 - 1$ end if $txn := new_txn$ if $new_txn = NULL$ then $txn := free_stack_top$ $txn.rc2 := txn.rc2 + 1$ $data.txn := txn$ $free_stack_top :=$ $free_stack_top.next$ $txn.rc1 := txn.rc1 + 1$ else $txn.rc2 := txn.rc2 + 1$ $data.txn := txn$ $txn.rc1 := txn.rc1 + 1$ end if $new_txn := txn$ $data.new := data.old + x$ end for $new_txn.done := true$ end procedure
--	---	---

The *next* field in a transaction record is used to maintain a stack *free_stack* of free transaction records that are not pointed to by any $M[i]$. This *free_stack* is used to reuse a transaction record for future CSs. We now describe the code in ABORTABLEMODIFY.

Reads of shared variables. Lines 3–6 in ABORTABLEMODIFY replace line 2 of MODIFY. These lines read $M[1]$'s valid value from either the *new* or *old* field of $M[1]$ based on Invariant I.

Writes of shared variables. Lines 9–39 in ABORTABLEMODIFY replace the write to $M[i]$ in line 4 of MODIFY. We note that, while these lines reflect our general transformation process for making a CS abortable, it is possible to shorten this code for this simple example. A write to $M[i]$ in ABORTABLEMODIFY occurs in three steps: **(i)** unlink $M[i]$ from its old transaction record to enable future reuse of that record; **(ii)** link $M[i]$ with the transaction record corresponding to this CS invocation; and **(iii)** commit that invocation, i.e., make it take effect atomically. We now explain these steps.

Step (i): Unlinking. Lines 12–22 in ABORTABLEMODIFY unlink $M[i]$ from its old transaction record stored in txn by line 9. We depict the steps of unlinking in Fig. 10, where inset (a) shows an initial state prior to executing lines 12–22. To maintain Invariant I, lines 12 and 13

10:18 Overrun-Resilient Multiprocessor Real-Time Locking

```

struct {
    old: word
    new: word
    txn: Ptr to txn_record
} data

```

```

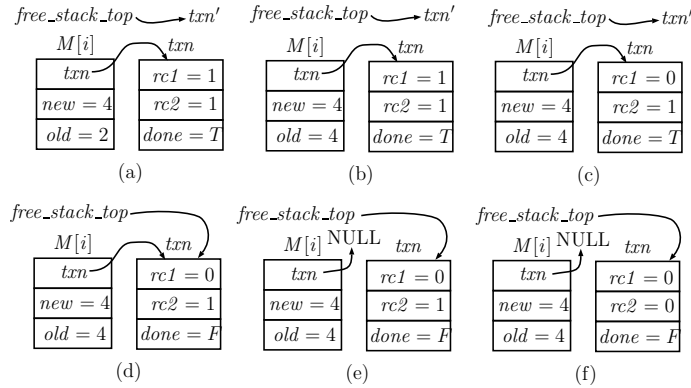
struct {
    rc1: int
    rc2: int
    done: boolean
    next: Ptr to txn_record
} txn_record

```

(a) Data element structure.

(b) Transaction record structure.

■ **Figure 9** Data structures for abortable CS.



■ **Figure 10** Unlinking process. The field *txn.next* is not shown.

copy $M[i].new$ to $M[i].old$ if $txn.done$ is *true*, as shown in Fig. 10(b). Lines 15, 21, and 22 decrement $rc1$ and $rc2$ of txn before and after setting $M[i].txn$ to NULL, respectively, as shown in insets (c)–(f) of Fig. 10. If $rc1$ becomes 0 after line 15, as shown in Fig. 10(c), then lines 17–19 push txn onto $free_stack$ and unset its $done$ field, as shown in Fig. 10(d).

Step (ii): Linking. Lines 24–35 link $M[i]$ to a transaction record. Line 24 assigns new_txn to txn , which is the transaction record corresponding to this invocation of `ABORTABLEMODIFY`. We illustrate the linking process when new_txn is NULL by considering insets (b)–(f) of Fig. 10 in reverse order. Fig. 10(f) shows an initial state after executing line 26 of the linking process when new_txn is NULL. Insets (e), (d), (c), and (b) of Fig. 10 illustrate incrementing $rc2$ (line 27), linking txn to $M[i]$ (line 28), removing txn from $free_stack$ (line 29), and incrementing $rc1$ (line 30), respectively. After this linking process, line 36 sets new_txn to txn to ensure that future loop iterations use this same transaction record. Also, line 37 performs the write operation (from line 4 of `MODIFY`) by updating $M[i].new$.

Step (iii): Committing. The CS is committed by simply setting $new_txn.done$ to *true* in line 39. It is this one-line commit at the end that obviates the need for any undo code.

Why two reference counters? To see why using only one counter is problematic, consider again the unlinking process shown in Fig. 10. If only $rc1$ is used and the CS is aborted after executing the step in Fig. 10(c), then txn is left in an inconsistent state, i.e., its $rc1$ field indicates no structure points to txn yet one does. Using both $rc1$ and $rc2$ enables this “inopportune” CS abort to be detected by simply checking whether $rc1$ is smaller than $rc2$. To fix the inconsistent transaction record in this case, we add a small code sequence to the

OS timer-handling code that deals with CS-budgeting timer expirations. This code completes the remaining steps of unlinking by executing lines 15–21 of `ABORTABLEMODIFY` if $txn.rc1$ is smaller than $txn.rc2$ (with an additional check of $txn = free_stack_top$ in line 15 to prevent inserting txn to $free_stack$ twice). The timer-handling code can access the CS-specific variables involved in these actions via a control page shared with the CS’s task. Note that *the added timer-handling code is the same for all abortable CSs and deriving the EBT of expiring the CS-budgeting timer after adding this code supports Assumption A2*. Similar inopportune CS aborts can affect the linking process, and they are dealt with similarly.

Generalizing to arbitrary shared data structures. For clarity, we presented the idea of an abortable CS via a simple example. However, in an online appendix [41], we present a set of routines for performing necessary actions (reading, writing, linking, unlinking), a set of rules to transform any ordinary CS into an abortable one, and an invariant-based proof that shows that these rules are correct. In our simple example, reading and writing occurred at the granularity of words. However, in the general scheme, any granularity can be assumed.

Have we really eliminated undo code? One could argue that abortable CSs merely intertwine undo-related actions with ordinary CS code. If this is so, do they offer any real advantages over simply following ordinary CS code with potential undo code? The answer is yes. With separate undo code, the “chicken and egg” problem mentioned in Sec. 1 arises: the undo code would have to be budgeted, and to avoid exhausting that budget, an ETB would have to be assumed for it, which could be very costly. Perhaps one could take the same “intertwined view” and inflate the cost of any CS by its undo cost, but then how is the (separate) undo code ever triggered? Presumably, the combined budget would have to be factored into two parts, one for the ordinary CS code and one for the undo code, bringing us back to the chicken-and-egg problem. While abortable CSs are immune from these problems, further research into supporting real-time undo code would certainly be valuable.

7 Experimental Evaluation

To assess the costs and benefits of overrun-resilient locking, we conducted two sets of experiments under LITMUS^{RT} [10, 15] on an eight-core 2.1GHz Intel Xeon Silver processor. To increase timing predictability, we disabled hyperthreading, low CPU power states, and CPU frequency scaling.

Experiment 1. We first assessed the costs of using abortable CSs that satisfy P2.1 vs. using CS ETBs to provision CS execution budgets. As a baseline, we measured the execution times of ordinary (non-abortable) operations on buffers, queues, and binary heaps. To assess the cost of abortable CSs, we compared the baseline to the execution times of corresponding abortable CS implementations. To assess CS ETB budget provisioning (which assumes unrealistically pessimistic conditions), we compared the baseline to the execution times of cacheless runs of the ordinary operations. Two metrics were considered: the *worst-case* (resp., *average-case*) *inflation factor* of an operation is the ratio between the observed maximum (resp., average) execution times of abortable/cacheless CS vs. that of the baseline. As this is not a paper on timing analysis, we note that the ETBs assumed here are provided to give a plausible sense of the pessimism they may entail; we make no claim that they are in fact upper bounds, or if they are, that they cannot be safely tightened.

■ **Table 4** Comparison between abortable and ordinary CSs.

Data structure	Buffer Write	Buffer Read	Queue Enqueue	Queue Dequeue	Heap Insert	Heap Extract
WC Baseline	39.0 ns	33.3 ns	46.7 ns	48.6 ns	89.5 ns	203.8 ns
WC Abortable	161.9 ns	76.2 ns	97.1 ns	252.4 ns	300.9 ns	981.9 ns
WC Cacheless	13.0 μ s	11.0 μ s	22.7 μ s	20.6 μ s	32.3 μ s	15.9 μ s
WC Abortable Inflation	4.14 \times	2.28 \times	2.08 \times	5.19 \times	2.95 \times	4.08 \times
AC Abortable Inflation	6.91 \times	2.55 \times	2.47 \times	5.53 \times	5.88 \times	3.36 \times
WC Cacheless Inflation	332.7 \times	329.8 \times	486.9 \times	424.4 \times	360.7 \times	784.2 \times
AC Cacheless Inflation	93.5 \times	113.5 \times	304.8 \times	200.8 \times	245.6 \times	882.4 \times

For each implementation, we determined the maximum and average duration of each operation (measured using the timestamp counter) through 10,000 trials, running alongside contention-generating tasks that contend for the memory bus. We separately measured the duration of our timing code and subtracted it from our results. For the read/write buffer, we used a one-word buffer with single-word reads and writes. We initialized the queue and heap to contain 1,000 items. Our results, shown in Tbl. 4, support the following observations.

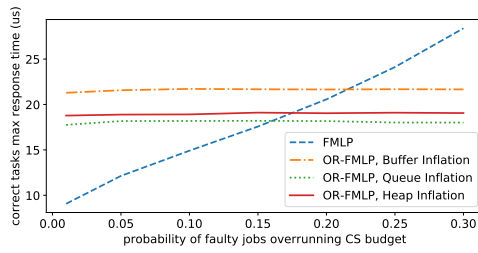
- **Observation 1.** *The worst-case inflation factor of abortable CSs was around two to five.*
- **Observation 2.** *The inflation factors for running cacheless was in the hundreds.*

The extremely high inflation factors for running cacheless were due to both instructions and data being accessed from main memory instead of mainly the L1 cache. While the effects of disabling caches on other processor mechanisms such as branch prediction, pipelining, and prefetching are not well documented, we suspect that these factors also contributed to the slowdown, especially in the case of heaps where branching code is common.

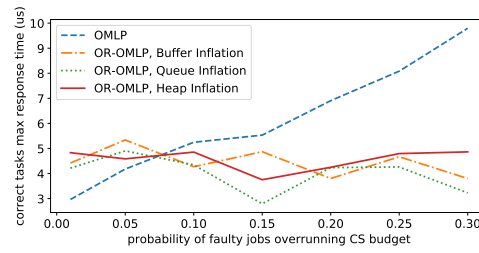
Experiment 2. We assessed the impacts of CS execution budget overruns under the OR-FMLP, OR-OMLP, FMLP, and OMLP by executing a task system consisting of an equal number of synthetic non-overrunning *correct* tasks and overrunning *faulty* tasks. (We did not examine task execution budget overruns because they require application-dependent mitigation; note, however, that common mitigations such as aborting the overrunning job can break the FMLP and OMLP.) Each task τ_i had $(C_i^e, T_i) = (1\text{ms}, 40\text{ms})$. We generated enough tasks so that the sum of all analytical task utilizations (C_i^a/T_i) was $0.8m$.

The task system had a single shared resource for which each job issued a single request. For each synthetic task τ_i , we generated its *CS execution budget* L_i^e and an *actual CS execution time*, denoted L_i , via three steps. When jobs of these synthetic tasks execute CSs, they acquire a lock and spin for the duration of the actual CS execution time. First, for each correct task τ_i , we set L_i^e to 0.2ms and L_i to 0.19ms (which was sufficient to preclude budget overruns due to overheads). Second, for each faulty task τ_i , we set L_i^e to 0.2ms under the FMLP and OMLP. Since abortable CSs require more execution budget, we inflated L_i^e for each faulty task τ_i under the OR-FMLP and OR-OMLP by considering three different *inflation scenarios* based on the data given for buffers, queues, and heaps in Tbl. 4. For each scenario, we inflated each such L_i^e by the worst-case abortable-CS inflation factor of that scenario’s data structure in Tbl. 4. Third, for each faulty task τ_i , we determined L_i by a type-1 Gumbel distribution.⁴ Under the FMLP and OMLP, the mean of this Gumbel distribution was set

⁴ The Gumbel distribution is often used to represent measurement-based probabilistic WCETs [18].



■ **Figure 11** FMLP vs. OR-FMLP results.



■ **Figure 12** OMLP vs. OR-OMLP results.

at 0.05ms. Under the OR-FMLP and OR-OMLP, this mean was inflated according to the average-case abortable-CS inflation factor given in Tbl. 4 for the corresponding scenario. We varied the probability of a job of a faulty task overrunning its CS execution budget from 0.0 to 0.3 with a step size of 0.05. The variance of the Gumbel distribution was determined by this probability value.

For each inflation scenario and CS execution budget overrun probability, we executed the task system for 10 minutes and measured the maximum response time among all correct tasks. Figs. 11 and 12 plot the recorded response times that supports following observations.

► **Observation 3.** *The worst-case response times of correct tasks under the OR-FMLP (resp., OR-OMLP) stayed relatively constant as the overrun probability increased.*

► **Observation 4.** *Cost/benefit tradeoffs are evident in these curves. For example, for buffers in Fig. 12, overrun-resilient protocols increased response times for overrun probabilities less than 0.10 and decreased them for greater probabilities.*

8 Revisiting Assumptions A1 and A3

We now return to Assumptions A1 and A3.

Assumption A1. As we have seen, user-level budgets have a fundamental dependency on the execution times of certain OS code paths. If one defines budgets for those code paths, then what entity would enforce them? The only alternative to budgeting is to require ETBs for these code paths, but this has major implications for real-time OS (RTOS) designs. For example, modern OSs tend to be highly preemptive, but preemptions greatly complicate measurement-based timing analysis. To avoid “chicken and egg” problems, RTOS designs need to be rethought, with enabling reliable timing analysis for critical code paths being a first-class concern. These code paths should be simple and non-preemptive and should have reasonable ETBs. Techniques like cache locking may help in this regard.

Assumption A3. Standard techniques [10] can account for the overheads/delays considered negligible by A3. From a timing-analysis point of view, the overhead of most concern is CPMD, as it is incurred on preemptions, which as noted already, are hard to deal with in timing analysis due to difficulties in predicting cache state. These difficulties have important implications for synchronization and scheduling: CSs that execute on a CPU (as opposed to, e.g., an I/O device) should be non-preemptive, and tasks should either be non-preemptive or only preemptive at certain “preemption points” [13]. While both task-scheduling options have been studied for simple task models [7, 14, 36], they warrant further attention in more complex models relevant today, such those based on processing graphs [4, 40, 45].

9 Related Work

Locking protocols that consider budget overruns (shown in Tbl. 1) have been explored in the past. However, none satisfy properties P1–P3, which define our notion of overrun resiliency.

Satisfying P1. To satisfy P1, a protocol must deal with jobs overrunning their budgets while in a CS. Prior work such as M-BWI and vMPCP satisfy P1 by allowing task budget overruns to occur inside of CSs and account for them analytically. In contrast, SIRAP and M-BROE satisfy P1 using FZs, introduced in [27], to avoid task budget overruns inside of CSs. Tradeoffs between satisfying P1 using FZs and overrun accounting are detailed in [5].

Satisfying P2. M-BWI and vMPCP both require accurate CS WCETs to produce correct overrun accounting. SIRAP and M-BROE also require accurate CS WCETs to correctly provision FZs. Since CS WCETs may exceed CS PETs, protocols that rely on forbidden zones and overrun accounting do not satisfy P2. In the absence of correct CS WCETs, ICSs and protocols such as RRP and RACPwP satisfy both P1 and P2 by aborting CSs when their budgets overrun. To maintain consistent state for shared data structures, ICSs, RRP, and RACPwP use versioning techniques.

Satisfying P3. No prior work considers P3. However, all protocols that satisfy P1 *can* satisfy P3 when ETBs are used to account for corresponding budget-enforcement mechanisms. Unfortunately, all protocols that satisfy P2 use versioning techniques that make copies of modified data. Thus, when CSs only modify data, the ETBs of versioning techniques can be as large as the CS ETBs, nullifying the benefits of ensuring P2. Only ICSs avoid the problem by allowing shared-resource state to remain consistent even when a job is aborted while executing the versioning technique, satisfying P3. However, ICSs are intended for uniprocessors, thus the ICS versioning technique cannot handle concurrent resource accesses.

In this work, we proposed the OR-FMLP and OR-OMLP, which in conjunction with abortable CSs, yield overrun-resilient locking protocols. In fact, prior work on versioning techniques can also be used with the OR-FMLP and OR-OMLP to yield overrun-resilient protocols. However, those prior works have unfortunate tradeoffs as discussed in Sec. 6.

10 Conclusion

We have presented the OR-FMLP and the OR-OMLP, which are overrun-resilient variants of the FMLP and the OMLP, respectively. Both the OR-FMLP and the OR-OMLP utilize FZs and the ability to abort CSs to circumvent problems associated with overrunning task and CS budgets. As our designs of these two protocols suggest, it is better to apply FZs in a coarse-grained way in a spinlock but in a fine-grained way in a suspension-based lock. For both protocols, we have carefully worked out how execution budgets should be defined. To easily apply these protocols to support operations on shared data structures, we have also presented abortable CSs, which enable such operations to be aborted with no undo code.

In future work, we plan to consider aborting CSs in other contexts, such as when using locks to access shared hardware resources. We also intend to investigate other types of locks, such as k -exclusion locks and reader/writer locks, as well as mechanisms for allocating “slack” generated by underrunning CSs to overrunning CSs. Additionally, full budgeting support enables the possibility of freeing system capacity by intentionally under-budgeting certain computations at the expense of aborting work more often; we plan to explore this possibility

as well. This paper also establishes a need for timing-analysis techniques that can guarantee safe ETBs without exorbitant pessimism. The various “chicken and egg” problems we have pointed out warrant scrutiny as well.

References

- 1 A. Alon and A. Morrison. Deterministic abortable mutual exclusion with sublogarithmic adaptive rmr complexity. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*, pages 27–36, 2018.
- 2 J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355, 1998.
- 3 M. Asberg, T. Nolte, and M. Behnam. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 129–140, 2013.
- 4 S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 222–231, 2015.
- 5 M. Behnam, T. Nolte, M. Asberg, and R. Bril. Overrun and skipping in hierarchically scheduled real-time systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 519–526, 2009.
- 6 M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the International Conference on Embedded Software*, pages 279–288, 2007.
- 7 M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 251–260, 2010.
- 8 A. Biondi, G. Buttazzo, and M. Bertogna. Supporting component-based development in partitioned multiprocessor real-time systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 269–280, 2015.
- 9 A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, 2007.
- 10 B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.
- 11 B. Brandenburg. *Multiprocessor Real-Time Locking Protocols*, pages 1–99. Springer, 2020.
- 12 B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60, 2010.
- 13 A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *Advances in Real-Time Systems*, pages 225–248, 1994.
- 14 G. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems. A survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2013.
- 15 J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–126, 2006.
- 16 F. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys*, 52(1):14:1–14:35, 2019.
- 17 R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.
- 18 R. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03:1–03:60, 2019.

- 19 D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, volume 4167, pages 194–208, 2006.
- 20 D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real Time Systems*, 48(6):789–825, 2012.
- 21 K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2003.
- 22 T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.
- 23 T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. *Communications of the ACM*, 51(8):91–100, 2008.
- 24 T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, 2006.
- 25 M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.
- 26 M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.
- 27 P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 149–158, 2002.
- 28 P. Holman and J. Anderson. Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174, 2006.
- 29 P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 295–304, 2003.
- 30 T. Johnson and K. Harathi. Interruptible critical sections. Technical Report TR94-007, University of Florida, 1994.
- 31 H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*, pages 86–95, 2014.
- 32 N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F.D. Smith. Attacking the one-out-of- m multicore problem by combining hardware management with mixed-criticality provisioning. In *Proceedings of the 22nd IEEE Real-Time Embedded Technology and Applications Symposium*, pages 49–160, April 2016.
- 33 H. Lee. Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, pages 364–379, 2010.
- 34 C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys*, 52(3):56:1–56:38, 2019.
- 35 V. Marathe, M. Spear, A. Acharya C. Heriot, D. Eisenstat, W. Scherer, and M. Scott. Lowering the overhead of nonblocking software transactional memory. Technical report, University of Rochester, November 2006.
- 36 M. Nasri, G. Nelissen, and B. Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, volume 106, pages 9:1–9:23, 2018.
- 37 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.
- 38 M. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 31–40, 2002.
- 39 T. Springer, S. Peter, and T. Givargis. Resource synchronization in hierarchically scheduled real-time systems using preemptive critical sections. In *Proceedings of the 17th IEEE International*

- Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 293–300, 2014.
- 40 M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, 2011.
 - 41 Z. Tong, S. Ahmed, and J. Anderson. Overrun-resilient multiprocessor real-time locking (longer version), 2022. URL: <http://jamesanderson.web.unc.edu/papers/>.
 - 42 J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222, 1992.
 - 43 R. Wilhelm. Real time spent on real time (invited talk). In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 1–2, 2020.
 - 44 R. Wisniewski, L. Kontothanassis, and M. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 199–206, 1995.
 - 45 K. Yang, G. Elliott, and J. Anderson. Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23th International Conference on Real-Time Networks and Systems*, pages 77–86, 2015.

Scheduling Offset-Free Systems Under FIFO Priority Protocol

Matheus Ladeira ✉ 

ISAE ENSMA, Chasseneuil, France
University of Poitiers, France

Emmanuel Grolleau ✉ 

ISAE-ENSMA, Chasseneuil, France
University of Poitiers, France

Fabien Bonneval ✉

Ecole Nationale de l'Aviation Civile, Toulouse, France

Gautier Hattenberger ✉ 

Ecole Nationale de l'Aviation Civile, Toulouse, France

Yassine Ouhammou ✉ 

ISAE-ENSMA, Chasseneuil, France
University of Poitiers, France

Yuri Hérouard ✉

ISAE-ENSMA, Chasseneuil, France

Abstract

On UAVs, telemetry messages are often sent following a FIFO schedule, and some messages, depending on the FIFO queue state may suffer long delays, and can even be lost if the FIFO queue is full. Considering the high complexity of the problem of assigning offsets to periodic tasks, we propose a new heuristic, called GCD+, that we compare to the methods of the state of the art, showing that GCD+ significantly outperforms them on synthetic tasks sets. Then we use a real UAV use case, based on Paparazzi autopilot, to show that GCD+ behaves well. The proposed algorithm is meant to be the new Paparazzi's automatic offset assignment method for messages.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Embedded software

Keywords and phrases Scheduling, non-preemptible, heuristics, FIFO, autopilot

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.11

Supplementary Material Source code for an implementation of GCD+ in Python was submitted under LGPL license to artefact evaluation, and can be found in: <https://github.com/lias-laboratory/gcdplus>

Software (ECRTS 2022 Artifact Evaluation approved artifact):

<https://doi.org/10.4230/DARTS.8.1.4>

Funding This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 826610. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Austria, Belgium, Czech Republic, France, Italy, Latvia, Netherlands.

1 Introduction

When conceiving an Unmanned Aerial System (UAS), i.e., a flying drone and every supporting device in the drone's network, the communication between the drone and the Ground Control Station (GCS) or the remote pilot may be critical, since it may determine if the drone is



© Matheus Ladeira, Emmanuel Grolleau, Fabien Bonneval, Gautier Hattenberger, Yassine Ouhammou, and Yuri Hérouard;

licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 11; pp. 11:1–11:19



Leibniz International Proceedings in Informatics

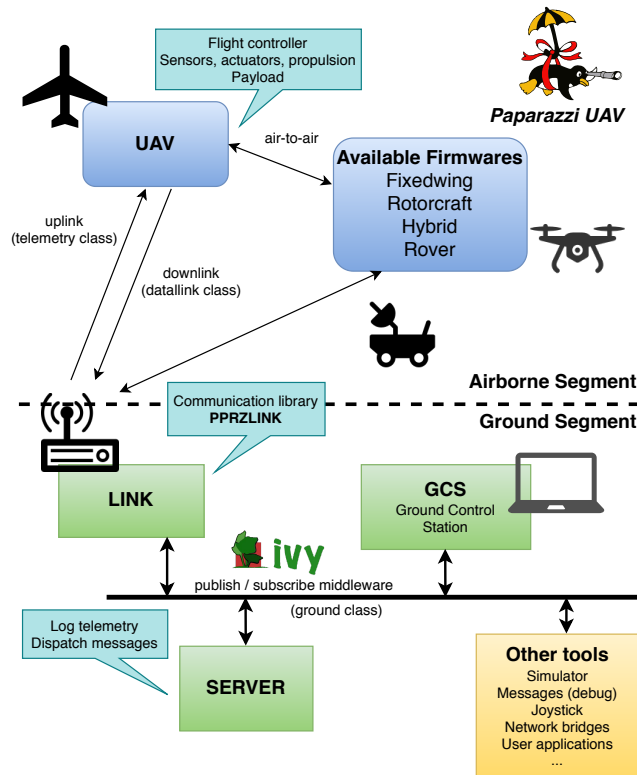
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



11:2 Scheduling Offset-Free Systems Under FIFO Priority Protocol

certifiable or not for a specific mission [15]. Without a reliable communication link and the guarantee that the GCS or the remote pilot can take control of the drone, safety and security of every entity in the surroundings must be insured only by its autonomous functionalities. This requires the development of very complex and, therefore, expensive embedded skills.

Hence, guaranteeing the nominal behaviour of the communication system is an important step in drone development and configuration. Nevertheless, not only a larger communication band can improve the exchange of messages between the different parts of the UAS: the choice of the moments in which these messages are sent, by orchestrating their periods and the moments of the first message transmissions, can play an important role in order to avoid large waiting queues and, consequently, message delays or even losses. Some message delays can make the GCS conclude falsely that the communication is lost and enter in fail-safe mode.



■ **Figure 1** Paparazzi system architecture.

Amongst current solutions for the problem, we can highlight the one used in Paparazzi [3, 11, 16], the open source autopilot developed by ENAC (École Nationale d'Aviation Civile, France) and used in many research-related domains. It is conceived using modular functions for the control algorithm, that are called in a single execution thread, one after the other according to a defined pattern of execution. In parallel, a telemetry thread is responsible for sending telemetry data through a wireless channel to a ground station, as depicted in Figure 1. Each type of message has a defined range of possible sizes and a specific period in which it will be sent. The messages are sent following a First-In-First-Out (FIFO) scheduling policy.

In their current approach, message periods are chosen and adapted so that they yield a very high hyperperiod, i.e., the time it takes for the pattern of message transmission to start repeating (mathematically, it is the Lowest Common Multiple of all message periods). If

every message is called to be sent in the same moment in time, this will happen only once in every hyperperiod. Therefore, by adjusting periods such that their LCM is high enough, the system will have only one simultaneous activation of all the messages in a very long period of time – days, years, millennia, or even further. However, this simultaneous call can still happen, while there are other methods that may prevent it completely.

The problem of message transmission in a single communication link is analogous to the problem of scheduling non-preemptive tasks on a single processor. More specifically, this is equivalent to the problem of finding offsets in an offset-free system of periodic tasks under FIFO scheduling (thus tasks are non-preemptive), which has been studied in the literature.

In this article, we analyse the problem of message and task scheduling, proposing GCD+, a new method to seek for better offsets and to avoid deadline misses. Section 2 explores in depth the motivating case for this development. Section 3 presents a mathematical model for the problem. Section 4 brings to light other heuristics that deal with the same problem. Section 5 describes our contribution: the new heuristic method. Section 6 compares our heuristic to three heuristics found in the literature. Section 7 applies the new method to a real case involving the Paparazzi autopilot, and finally, Section 8 concludes the paper.

2 Motivating Example

Currently, in order to avoid critical events (several simultaneous message calls), the adopted strategy in Paparazzi consists of two methods applied in parallel. The first one is changing message periods in small amounts (from about 1% to 10%), so that the critical case for any two messages only happens once in a very long interval (the hyperperiod, calculated as the LCM of the periods). For example, instead of using the periods 3 and 5 seconds, which will compose an execution pattern that will repeat every 15 seconds (i.e. the hyperperiod is 15 seconds long), one can choose to use periods of 3.1 and 5.1 seconds, which yield a hyperperiod of 158.1 seconds. This technique can help reduce the interference specially considering sets of several messages, which will have a very large hyperperiod, meaning that concurrent calls become less probable. However, it does not guarantee that critical cases will not happen, and it might lead to unexpected long delays that are very hard to reproduce.

The second method, complementary to the first, relies on the use of offsets, so the critical instant does not happen at $t = 0$. The first message to be evaluated is assigned an offset of 0; the second one, an offset equivalent to 10% of its period; the third one, 20% of its period; and so on (and, when the counter reaches 100%, it resets to 0 so that every offset is smaller than the respective period).

Using offsets is a well-known strategy to increase the schedulability of task systems: finding the right combination of offsets for each set of tasks in order to avoid overlapping [7]. The problem consists in choosing offsets in an offset-free system such as to avoid load peaks. Hence, instead of being called at every instant $t = n \cdot T_i$ (where $n \in \mathbb{N}$ and T_i is the period in which τ_i is executed), a task would be called at every instant $t = O_i + n \cdot T_i$ (where O_i is an initial offset given to τ_i). By coordinating the individual offsets of each task according to their periods and to the other periods in the system, it might be possible to completely prevent critical cases or, at least, reduce the maximum possible interference between messages – and in a mathematically predictable way.

The analogy between message and task scheduling holds due to the fact that the transmission of a message happens at a defined rate, such as the processing of individual instructions in a CPU, and the message sizes are analogous to the execution times of the tasks. These, however, must be considered non-preemptible, since once a message starts being sent, the

11:4 Scheduling Offset-Free Systems Under FIFO Priority Protocol

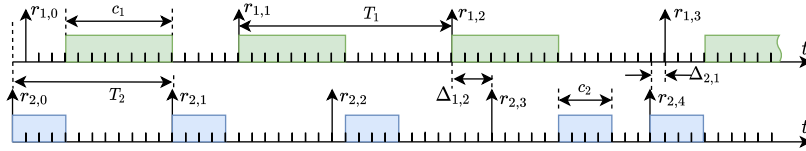
process cannot be interrupted by a new arriving message. Also, due to implementation constraints (namely a ring buffer used to store output messages) the first message to arrive in the transmission queue will be the first to be sent, and therefore they are under a FIFO scheduling policy.

Paparazzi uses a configuration file to describe its telemetry, where each message is characterized by a period and an offset. Therefore, increasing schedulability by changing the offset does not require changing a single line of the C code of the autopilot, saving testing and validation efforts.

In our case, the goal is not only to meet deadlines, but also to minimize the response time of the messages. This goal is related to the freshness of data sent over the datalink. Indeed, the longer a message is kept in a queue before being sent, the less representative of the state of the system it is.

In the following sections, we will analyse the behaviour of task systems (but the analogy between processing non-preemptible tasks and sending telemetry messages holds for every case).

3 Problem Statement



■ **Figure 2** Model of execution of a set of two periodic tasks under FIFO with different offsets. The up arrows represent task releases.

An offset-free task system [7] Φ of n periodic tasks executing in a single processor under FIFO scheduling (and, therefore, non-preemptive), each task τ'_i (such that $i \in [1, n] \subset \mathbb{N}$) being given by a tuple composed of two positive integers – its period T_i and its Worst Case Execution Time (WCET) c_i –, is given by Equation 1.

$$\Phi \triangleq \{\tau'_i \mid \tau'_i = (T_i, c_i) \forall i \in [1, n]\} \quad (1)$$

We are interested in finding a vector of n integers O_i such that the concrete task system Θ , defined by Equation 2, is schedulable and, in order to guarantee data freshness, where the tasks have the smallest starting time. Since the scheduler is non-preemptive, an earlier starting time means a lower response time.

$$\Theta \triangleq \{\tau_i \mid \tau_i = (\tau'_i, O_i) \forall i \in [1, n]\} \quad (2)$$

In order to do that, like in [7, 18, 1], we will make use of Theorem 1. Nevertheless until now, it has been used to guarantee space between releases of pairs of tasks, but in our GCD+ heuristic, we use it to guarantee space between more tasks than two, hence the name of our heuristic.

► **Theorem 1** ([18]). *Given two concrete tasks τ_i and τ_j , the minimum distance between the release of a job of τ_i and the next job of τ_j is given by:*

$$\Delta_{i,j} = (O_j - O_i) \bmod \text{GCD}(T_i, T_j) \quad (3)$$

Using the example depicted in Figure 2 and considering that τ_1 has an offset $O_1 = 1$, a period $T_1 = 16$ and an execution time $c_1 = 8$, and that τ_2 has an offset $O_2 = 0$, a period $T_2 = 12$ and an execution time $c_2 = 4$, the minimum distance between the release of a job of τ_1 and the next release of τ_2 is:

$$\Delta_{1,2} = (O_2 - O_1) \bmod \text{GCD}(T_1, T_2) = (0 - 1) \bmod 4 = 3$$

Similarly, the minimum distance between the release of a job of τ_2 and the next release of a job of τ_1 is:

$$\Delta_{2,1} = (O_1 - O_2) \bmod \text{GCD}(T_1, T_2) = (1 - 0) \bmod 4 = 1$$

By taking Equation 3 and comparing its result to c_i , if $c_i > \Delta_{i,j}$ and jobs use their WCET, it is guaranteed that there will be a delay in the execution of a job of τ_j , and this delay will be of at least $c_i - \Delta_{i,j}$. Therefore, we can define a lower bound $I_{i,j}$ for the largest interference caused by the execution of a job of τ_i on the execution of the next job of τ_j as being the amount of time the execution of τ_i overlaps into the minimum distance from its request to the subsequent request of τ_j . So, we have Equation 4.

$$I_{i,j} \triangleq \begin{cases} 0 & \text{if } c_i \leq \Delta_{i,j}, \text{ or } i = j \\ c_i - \Delta_{i,j} & \text{otherwise} \end{cases} \quad (4)$$

Note that we use a FIFO scheduling policy, and that if a job is released before another one, then the former will be executed before the latter. The interference that a job under analysis can suffer when it is released is therefore the remaining execution time of the job in execution (if any) plus the sum of the worst-execution times of the jobs which are ready at or before the release of the job under analysis. The value of $I_{i,j}$, if positive, shows the lowest bound for the biggest amount of delay that will be caused in the execution of a job of τ_j due to the execution of a job of τ_i . The maximum delay can still be larger than $I_{i,j}$ in case any backlog occurs, as will be seen in a following example.

Using the example in Figure 2, the lower bound for the maximum delay that τ_1 causes in τ_2 is:

$$I_{1,2} = c_1 - \Delta_{1,2} = 8 - 3 = 5$$

And the minimal value for the maximum delay that τ_2 causes in τ_1 is:

$$I_{2,1} = c_2 - \Delta_{2,1} = 4 - 1 = 3$$

It can be seen from Figure 2 that the maximum delays suffered by tasks τ_1 and τ_2 are effectively correspondent to $I_{2,1}$ and $I_{1,2}$, respectively. However, considering the same system but with $c_2 = 6$ instead of 4, $I_{2,1} = 5$, but instead the simulation shows that a maximum delay of 6 happens. This can be explained by the presence of a backlog (an accumulation of preceding delays).

This lower bound can be far from the real maximum delay in the case for more than two tasks in a system, since delays can more easily accumulate. Yet, reducing the lower bound for the maximum interference in task pairs is shown to improve the system's schedulability [7, 8].

► **Proposition 2.** *If $I_{i,j} = 0 \forall i, j$, then there is no delay in the execution of the system, i.e., every task job $\tau_{i,k}$ ends its execution time in the instant $r_{i,k} + c_i \forall i, k$.*

Proof. We prove it by recurrence. If $I_{i,j} = 0 \forall i, j$, the first task job to be requested will certainly finish its execution before the next job request to happen, since no task executes before it. Hence, the next job will not have any initial delays, and since $I_{i,j} = 0 \forall i, j$, it will also finish its execution before the next request, and so on. ◀

The goal of this paper is to present a method to look for the state of null interference discussed in Proposition 2 if this state exists.

4 State of the Art

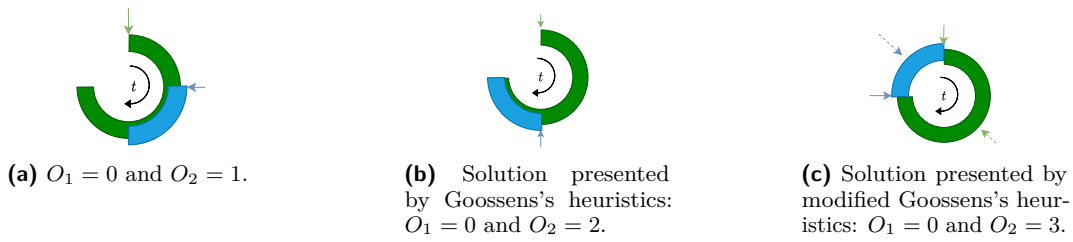
Some methods have been proposed to assign offsets to systems independently of their scheduler (FIFO, Earliest Deadline First, etc.), such as the heuristic proposed in this paper. Given the complexity of the problem, well described by [7], the use of heuristics is a common approach to solving it. In [7], the authors have proposed an algorithm (hereafter called Goossens’s Heuristics) to order every possible task pair in a set according to a decreasing order of the GCD of their periods, so each pair of tasks (starting from the highest GCD) is given a pair of offsets which are apart by half of their GCD. The goal is to try to separate as much as possible the task calls. The complexity of the algorithm is $\mathcal{O}(n^2 \cdot (\log T_{max} + \log n^2))$. We can think of a slight modification to this method by separating the “half instant” (call instant plus half of the execution time) of every task, instead of their calls. This modification is hereafter called Modified Goossens’s Heuristics.

Later, a modification of the previous heuristics is proposed in [8], regarding the priority that determines the order in which task pairs are evaluated, but keeping the principle of separating as much as possible the task calls. Four new priority assignments are evaluated, based on expressions containing the pairwise GCDs. The algorithm complexity is, therefore, considered to be the same.

Also, a method is proposed in [9] (hereafter called CAN Message Heuristics) to look for the longest least-charged time interval in the interval $[0, T_{max})$, and then put the task call in the middle of that interval (starting from the task with the lowest period. Its complexity is said to be $\mathcal{O}(n \cdot T_{max})$, therefore it is pseudo-polynomial.

Paparazzi currently uses a method where each message has an offset correspondent to a multiple of 10% of its period (modulo 100%) [17]. The multiple is defined at the moment the message is added to the set, such that: the first message has an offset correspondent to 0% of its period; the second, 10%; the third, 20%; and so on. In mathematical notation: $O_i = ((i - 1) \bmod 10) \cdot 0.1 \cdot T_i$. Its complexity is, hence, $\mathcal{O}(n)$.

Moreover, some methods were proposed specifically for FIFO schedulers. In [1], the authors explore several properties for FIFO schedulers and derive a sufficient (but not necessary) test to assess the schedulability of a task set (given the tasks’ periods, execution times and offsets). However, they do not propose a method for calculating the tasks’ offsets other than assigning random values, while in [12] the authors do present such a method. Their proposal consists in using one or several offsets per task in order to reproduce, under FIFO, the schedule constructed by another scheduler (Earliest Deadline First, for example). Nevertheless, implementing this approach in the case of the Paparazzi autopilot would require the autopilot’s code to be modified, retested and revalidated, while a single offset per task requires only a specific configuration (already expected by the code).



■ **Figure 3** Representation of the execution of a system around a modular circle of size 4, with $T_1 = 16$, $c_1 = 3$, $T_2 = 12$ and $c_2 = 1$.

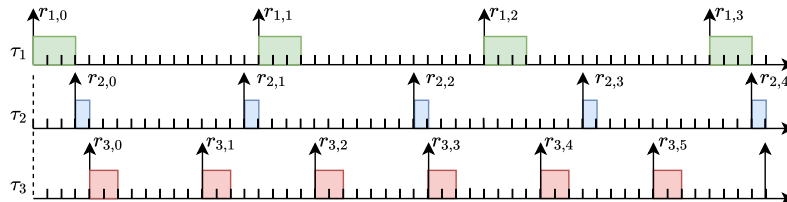
5 Contribution

This paper's contribution relies on the extension of a simple yet very useful technique to analyse pairs of tasks, allowing it to be used on whole sets of tasks. This technique is based on the modular arithmetic around the GCD of their periods, which is based on Theorem 1. Therefore, it needs a “sufficiently large” GCD to function properly, i.e., the GCD of all the periods cannot be smaller than any execution time. Otherwise, this method will give poor results.

If we analyse the modular circle around the $\text{GCD}(T_i, T_j)$, hereafter also referred to as $\text{GCD}_{i,j}$, and we represent the execution of those tasks from their release to their end, we can rapidly see if there will be any overlap in execution, such as in Figure 3a. The overlap in the GCD circle means that there will be an instant during the execution where there will be a task release, but another one will already be in execution. Similarly, if there is no overlap, then there will not be any delay whatsoever if there is no backlog.

The method presented here intends to reduce execution overlaps, but considering the system as a whole instead of pairs of tasks. For this, we take advantage of a property of semi-harmonic task sets, which means their task periods are all multiple of a given value Ω . This can be the case for several task systems, specially when we consider that there are several techniques to adapt task periods in given ranges so that they can approach this semi-harmony [2, 20, 4, 19, 13].

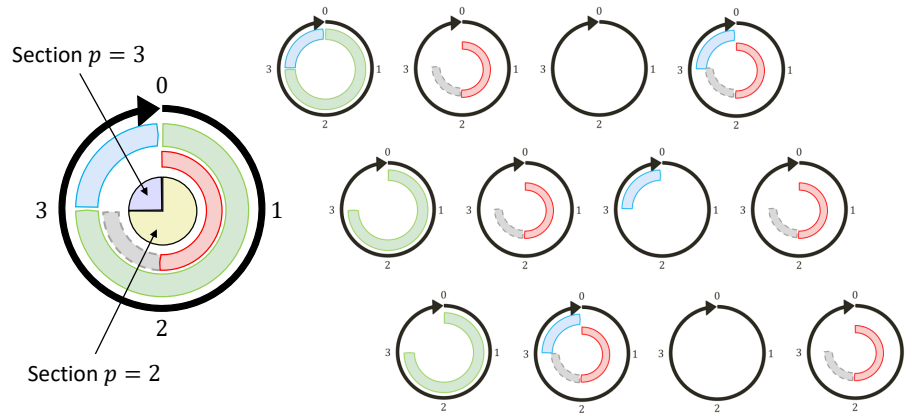
To better understand the method, we can start with an example. Let us suppose we want to assign an offset for a task τ_3 of period $T_3 = 8$ and execution time of $c_3 = 2$ in the system represented by Figure 3c. We might have the impression that adding any task to the system would cause an overlap in execution times. However, even if the circle looks fully occupied, the system still has some free spaces, as seen in Figure 4.



■ **Figure 4** Execution of τ_1 (top, green), τ_2 (middle, blue) and τ_3 (bottom, red).

In fact, when comparing the GCD circle with the real execution of the system, τ_1 is released only once every $\frac{T_1}{\text{GCD}_{1,2}} = 4$ GCDs. On the right side of Figure 5, it is represented in green in GCDs 0, 4 and 8. This means that there is still room to execute, without interference

with the two considered tasks, within the 3 GCDs out of 4 which are not used by τ_1 , using τ_1 reserved slot on the GCD circle. This reserved slot is called a section, and is represented on the left part of Figure 5 as a yellow sector in the central pie chart. Sections are partitions of the GCD cycle that will be allocated to tasks which have to be collocated on the same GCD cycle. For example, since τ_1 and τ_2 have respective periods of 16 and 12, there will be one time interval of size 4 such that τ_1 and τ_2 are both released each LCM of the periods. Now, if we consider τ_3 , of period 8, since $\text{GCD}_{1,3}$ is twice as large as $\text{GCD}_{1,2,3}$, τ_3 can be added to the system using the section allocated to τ_1 because it can always alternate with τ_1 . Since this section is a partition over the GCD cycle, it will not interfere with τ_2 either. This is represented in Figure 5, where task τ_3 is assigned to the second GCD circle, at the beginning of the yellow section, and will then use this section on circles numbered $1 + 2k$ for any natural integer k .



■ **Figure 5** Representation of the execution of τ_1 (green), τ_2 (blue), τ_3 (red) and τ_4 (dashed grey) over the overall GCD circle (left) and the cycles 0 to 11 composing the hyperperiod (right).

Several other tasks could also be added without interfering with the rest of the system, in the unused spaces seen in Figure 5. For example, if we consider τ_2 , the ratio $\frac{T_2}{\text{GCD}_{1,2}} = 3$ allows execution within the 2 GCDs over 3 which are not used by τ_2 . Therefore, we could add, without creating interference, one or several tasks using the section used by τ_2 (represented in purple on Figure 5) two GCDs over three in the same way as τ_3 used τ_1 's unused GCD section. In both cases, we can observe that this addition of tasks without interference is always possible if the period of these additional tasks is a multiple of $\text{GCD}_{1,2}$. As a result, by construction, if $n - 2$ tasks were added to the initial system of two tasks using this method, $\text{GCD}_{1,\dots,n} = \text{GCD}_{1,2}$.

Note that this condition is sufficient but not necessary to add a new task to a system without interference. For example, if we consider the system $\Phi = \{\tau_1 = (T_1 = 6, c_1 = 1), \tau_2 = (T_2 = 10, c_2 = 1)\}$, an additional task $\tau_3 = (T_3 = 15, c_3 = 1)$ can be added without interference while T_3 is not a multiple of $\text{GCD}_{1,2}$. We do not seek a necessary and sufficient condition of non overlap, because we know that finding non overlapping intervals is a hard problem. Indeed, the simultaneous in-congruence problem, which is at the root of this problem, is NP-hard in the strong sense [5].

In the problem tackled by this paper, we do not have the freedom to change periods, but we can retain the main idea by working with the GCD of all the tasks to try to find offsets minimizing the interference. In an attempt to consider the system holistically instead

of a union of independent pairs of tasks, the proposed heuristics of this paper considers the GCD of all the periods. In our context, the time unit being the duration of a bit on the network, and the periods of the telemetry messages often being multiples of tens, hundreds or thousands of milliseconds, we expect the global GCD to be large in general in real applications.

The global GCD, hereafter called Ω , is defined in Equation 5.

$$\Omega = \text{GCD}_{1,2,\dots,n} \quad (5)$$

If two tasks have a GCD between their periods which is greater than Ω , it is possible to arrange them as it was done for τ_3 in the preceding example. On the other hand, if their GCD is exactly Ω , then they must be put in a modular circle around Ω to be analysed, because over the hyperperiod, there will be a time window of size Ω where both tasks will be released.

Our method seeks that sort of arrangement in any given system by checking how many distinct sections (reserved slots, such as the one shared by τ_1 and τ_3 in the example) will need to co-exist in Ω , so that groups of tasks can be scheduled according to their respective sections and that each section will have its own reserved interval of time, without ever overlapping with other sections. The amount of sections will be given after analysing the quotient of each task period by Ω : if the GCD between two quotients is greater than one, it means that their respective tasks can be put in the same section in the mod Ω circle without creating an overlap in execution times. This quotient will be called a task's subperiod as defined in Equation 6.

$$T_{S_i} = \frac{T_i}{\Omega} \quad (6)$$

From the example in Figure 4, $T_{S_1} = 4$, $T_{S_2} = 3$ and $T_{S_3} = 2$. Note that T_{S_1} and T_{S_3} are multiple of 2 (which is why we characterize their section as prime $p = 2$ in Figure 5), and that T_{S_2} is co-prime with the other two. Since $T_{S_2} = 3$, the section assigned to τ_2 is characterized by a prime $p = 3$.

If we divide the whole execution of the system into cycles of Ω units of time such as in the right side of Figure 5, each task can be released periodically every T_{S_i} cycles. So, τ_1 can be released every four cycles, and τ_3 , every two cycles. The algorithm shall be able to schedule τ_3 so that it avoids being released in the same cycle as τ_1 . This can be done by analysing the possible cycles it can be assigned to: since its subperiod is only two, it can only be assigned to cycle zero or cycle one (assigning it to cycle two is the same as assigning it to cycle zero due to its periodicity). For cycle zero, the section for multiples of two is already occupied by τ_1 , but cycle one is free. Then, it can be assigned to start at the beginning of the section of prime two in cycle one.

To sum up, we have three values to choose for each task:

1. The cycle number between 0 and $T_{S_i} - 1$;
2. The section within the cycle, acting as a partition, whose prime has to be a divisor of the subperiod;
3. An offset inside the section, in case two tasks share the same cycle and section (e.g., if we want to add τ_4 with $T_4 = 8$ and $c_4 = 1$ in cycle one, in the section of prime $p = 2$, right after every execution of τ_3 , represented in dashed gray on Figure 5).

For each assignment above there is a partial offset, respectively O_C , O_S and O_I , such that the final offset will be given by Equation 7. Tasks assigned to start at the same cycle will have the same O_C , while those that use the same section (multiples of the same prime) will share the same O_S , and if they share both O_C and O_S they will have to have distinct O_I .

$$O_i = \Omega O_{C_i} + O_{S_i} + O_{I_i} \quad (7)$$

Using our example in Figure 5, τ_1 and τ_2 have $O_C = 0$ since they are executed in the first cycle, while τ_3 has $O_C = 1$. As τ_1 and τ_3 share the same section, they have the same $O_S = 0$, while τ_2 has $O_S = 3$, which is the moment in the GCD circle when their section begins. And, since every task starts in the same time as their respective section, every $O_I = 0$. A new task τ_4 with $T_4 = 8$ and $c_4 = 1$, set to be executed right after τ_3 finishes its execution (as seen in Figure 5), would share the same values of O_C and O_S with τ_3 , but would have $O_I = 2$.

Note that, if we want to add a task τ_5 such that $T_5 = 20$ and, therefore, $T_{S_5} = 5$, independently of the execution time of the task, GCD+ would have to create a new section: those for subperiods multiple of 5. The section will be put to begin its execution right after the one for multiples of 3 (i.e., for τ_5 , $O_C = 4$). It will necessarily result in an interference in every task in the section for multiples of 2, as GCD+ was not able to find a better solution.

Algorithm 1 represents the algorithm to determine the values of each partial offset. First, it calculates Ω , the overall GCD of all the periods. Based on this value, it assigns tasks one by one to the section that will increase the least in size (i.e., for every cycle, the longest it takes for its tasks to finish their execution). By doing this, it can calculate the partial offsets O_{C_i} and O_{I_i} so that the task can start as soon as possible in the section without overlapping with any other. When every task is assigned, it can then calculate each section size and finally apply an offset to each section, so one is released only after the previous has finished its execution. Its complexity is $\mathcal{O}(n \cdot (\log T_{max} + \sqrt{T_{max}} + \frac{\log T_{max}}{\log \log T_{max}} \cdot (\log T_{max} + T_{max})))$ using the Euclid's algorithm for finding GCDs in $\mathcal{O}(\log T_{max})$, a sieve to factor numbers in approximately $\mathcal{O}(\sqrt{T_{max}})$ and considering the number of distinct prime factors of a number to be at most $\frac{\log T_{max}}{\log \log T_{max}}$. Note that the last T_{max} is only a superior bound to T_{S_i} , and the algorithm generally works very far from this bound.

■ **Algorithm 1** GCD+.

Require: $\Phi = \{(T_i, c_i) \mid i, T_i, c_i \in \mathbb{N} \wedge i \in [1, n]\}$
Ensure: $O = \{O_i \mid i, O_i \in \mathbb{N} \wedge i \in [1, n] \wedge O_i < T_i\}$

- 1: $\Omega \leftarrow \text{GCD}(\{\forall T_i\})$
- 2: **for all** $\tau_i \in \Phi$ **do**
- 3: **if** $T_i = \Omega$ **then** assign($\tau_i, 1$)
- 4: **else**
- 5: Find prime factors of T_{S_i} (Equation 6)
- 6: Choose prime p such that assigning τ_i to it increases its section size by the least possible amount
- 7: assign(τ_i, p), defining the cycle O_C and internal offset O_I
- 8: **end if**
- 9: **end for**
- 10: calculate every O_S (section sizes)
- 11: **for all** $\tau_i \in \Phi$ **do** calculate O_i according to Equation 7
- 12: **end for**

6 Performance comparison

6.1 Experimental setup and chosen metrics

Random task sets are generated according to the methodology proposed in [10], to generate an unbiased distribution of utilization factors.

In order to generate representative task periods, there are two available methods in the literature. The first one, used in [1] for generating semi-harmonic task sets, chooses a random integer between defined lower and upper bounds, and multiplies the random number to a defined value, of which every period will be a multiple of. The second one [6], more general than the first, generates task periods from a finite set of possibilities according to a given factor distribution. The latter was chosen due to it being more representative of real-world scenarios.

The method for generating task periods works as follows: a set of prime numbers $\{p_i\}$ is given to the algorithm as an input. Each prime p_i related to a unique vector $\{x_j \in \mathbb{N}\}$ of size n_i . The vector represents the probability distribution that the corresponding exponent will be chosen to compose the final period. In other words, the probability that the factor $p_i^{(j-1)}$ will compose the generated period is $x_j / \sum_k x_k$. The possible values for the periods are limited to $\prod_i p_i^{(n_i-1)}$, and so is the hyperperiod of the system.

The generating set of vectors for each task period is obtained from the factors present in the periods of a real set of messages from Paparazzi¹. This file, in addition to being used in the real telemetry system of rotorcraft equipped with Paparazzi, contains periods that were purposefully tuned to increase the hyperperiod (using 11.1 seconds instead of 10, for example). This causes the overall GCD to be reduced, approaching then a worst-case scenario in our domain of application. Yet, the overall GCD has proven to be sufficiently large.

The generated periods and generated utilization factors are multiplied to obtain the execution times. These are then rounded to the nearest integer. Task sets that have any execution time rounded to 0 are discarded, since they will not have the effective number of tasks that were demanded. Also, only sets for which the GCD of all the periods is greater or equal to the largest execution time will be used. This case is often found in real applications such as Paparazzi, and is needed for our heuristics to work properly.

Then, the algorithms for finding the offsets are used in each task set. After each offset assignment algorithm is run, the respective concrete task sets are simulated over an interval equivalent to two hyperperiods plus the greatest offset and, for every task in every set, its maximum delay η_i is registered (the largest period of time between a task release and the beginning of its execution). This value represents the largest interference a task suffers. The delays can then be put directly in a box plot to be analysed in their brute form, where each task in each set will have its data point indicating the value of its maximum delay, and these data points will be condensed in boxes where each box represents a single algorithm for offset assignment.

However, while a certain absolute value for a delay (for example, 1000 time units) can mean a significant deadline miss (e.g., a task with period 500 with implicit deadline), for others it can mean only an “affordable” amount of 10% of its period (e.g., a task with period 10000 in a system with few and low-utilization tasks). Therefore, we must also evaluate the ratio between each delay and each corresponding task period to better evaluate the obtained results.

¹ https://github.com/paparazzi/paparazzi/blob/master/conf/telemetry/default_rotorcraft.xml

11:12 Scheduling Offset-Free Systems Under FIFO Priority Protocol

If a task was delayed at most by 100 time units in a system in which the shortest task executes in 200 time units, then it performed better than one that had the same maximum delay but in a system where the maximum execution time is 50, indicating that task delays have been chained and, in our case, probably more messages are waiting in the queue. Hence, to look for signs of chaining delays, we also analyse the ratio between the delay and the maximum execution time of other tasks in the set.

To evaluate the response time of each task, the maximum delay summed to the execution time of each task is normalized with respect to each execution time. Values close to 1 will indicate a small relative change to the response time.

A final analysis takes into account the schedulability of each task set. If any implicit deadline was missed, the task set is marked as unschedulable under FIFO. Then, the amount of schedulable sets is counted for each offset-assignment method, and they are compared between each other for each value of the total utilization factor.

In summary, the metrics used in the analysis are in the following list:

1. interference of other tasks η_i
2. interference normalized by period η_i/T_i
3. interference normalized by concurrent tasks' maximal duration $\eta_i/\max_{\{\forall \tau_k \neq \tau_i\}}(c_k)$
4. maximum response time $(\eta_i + c_i)/c_i$
5. schedulability

6.2 Results

The codes of some heuristic methods cited in Section 4, as well as the new contribution were implemented and executed in Python 3.9.6, using a laptop with Ubuntu 18.04.1, Intel Core i7-4710MQ CPU (2.50GHz, 4 cores, 8 threads, 2054MHz), and 16 GB of RAM.

GCD+ is comparatively evaluated using sets of 8 and 16 tasks. The limit on the number of tasks is due to the duration of the simulation used to compute the metrics. The time for each method to yield an offset assignment is measured and, later, the resulting offset assignments are evaluated in a simulation according to the criteria presented before. The simulation stores the maximum delays of each task (the difference between the time it starts its execution and the time it was called). Also, simulations with 1000 sets and then 2000 sets resulted in identical graphs, so we use only 1000 sets as a sufficient sample. The offset calculation time for each group of sets is shown in Table 1 for sets with 70% utilization. Other utilization values did not present significant variations.

■ **Table 1** Average time to assign offsets (results from 1000 filtered sets at 70% utilization).

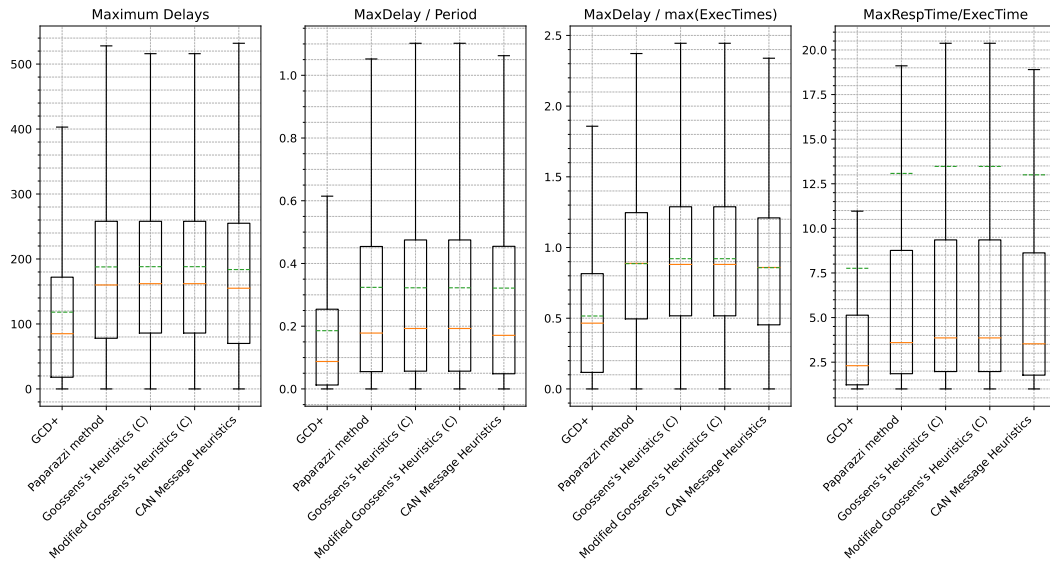
Method	Time - 8 tasks (μ s)	Time - 16 tasks (μ s)
GCD+	54.8	151
Paparazzi method [17]	4.78	9.11
Goossens's method [7]	22.8	74.7
CAN message method [9]	45.7	346

Table 1 shows that the time to calculate offsets is very small from a human perspective for every method. It also shows that every method is scalable, and therefore we can focus solely on their results.

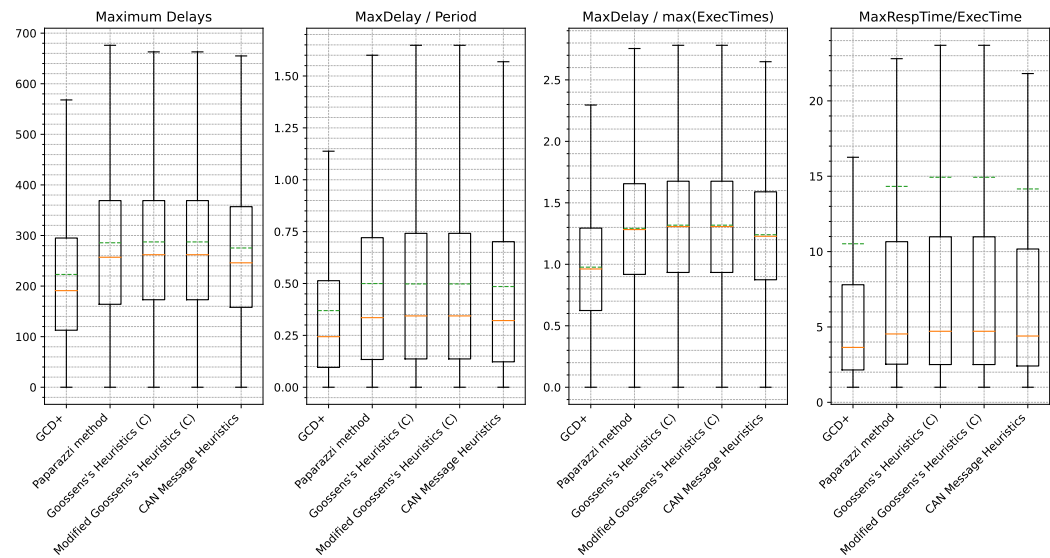
Box-plots are used to represent simulation results, where each heuristic method has its box. In the plots, the median (50th percentile) is represented as an orange continuous bar, the limits of the box are the first quartile (25th percentile) and third quartile (75th percentile),

and whiskers span from a box limit until the furthest value such that its size is, at most, 1.5 times the distance between the first and third quartile. All values beyond the whisker limits are considered outliers and are not represented in order to allow readability. However, the representations showing outliers follow the same tendencies as shown in the following figures. The mean value is also represented, as a dashed green bar.

For sets of 8 tasks, the behaviour of the systems can be seen in Figure 6 for 70% utilization and in Figure 7 for 90% utilization. Similarly, the behaviour for 16 task systems is seen in Figure 8 for 70% utilization and in Figure 9 for 90% utilization. In these figures, the box plot furthest to the left shows the first metric (η_i), the one at its right side shows the second

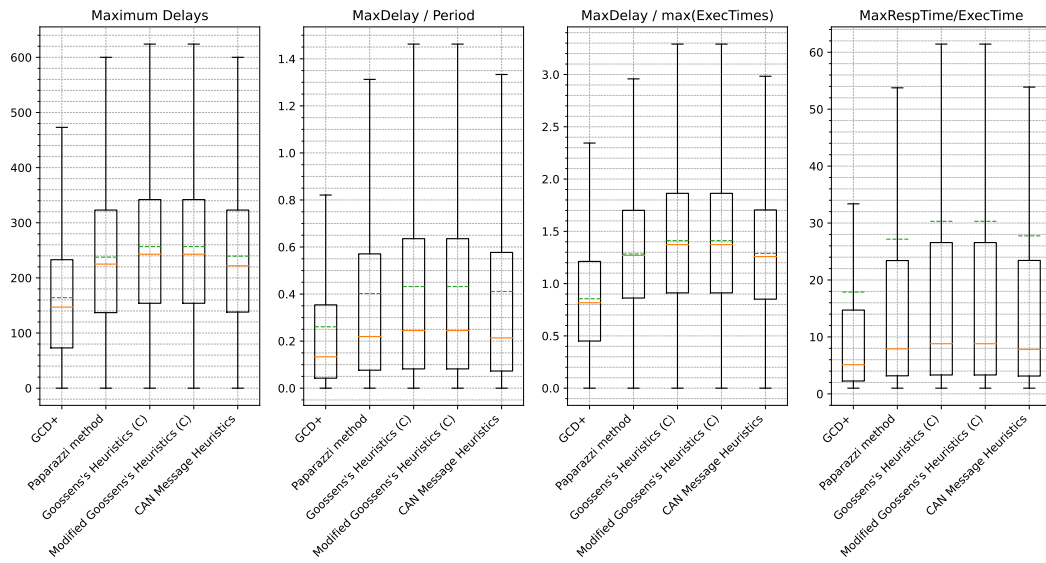


■ **Figure 6** Delays extracted from simulations for 1000 filtered sets of 8 tasks, with $U = 70\%$.

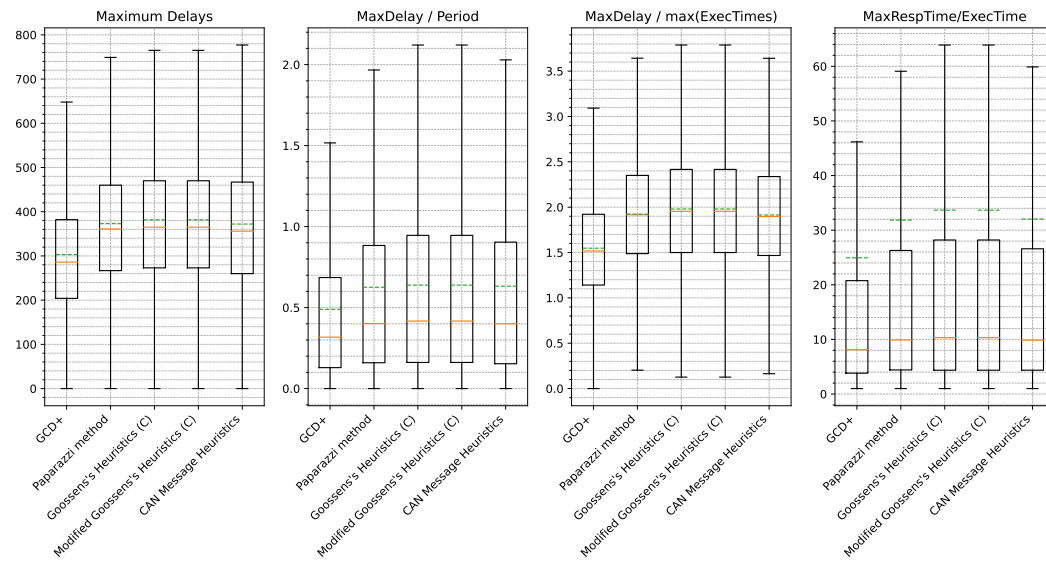


■ **Figure 7** Delays extracted from simulations for 1000 filtered sets of 8 tasks, with $U = 95\%$.

11:14 Scheduling Offset-Free Systems Under FIFO Priority Protocol



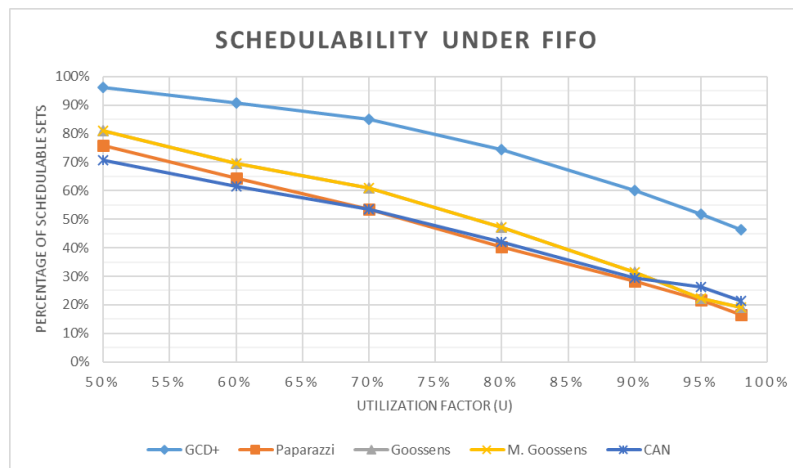
■ **Figure 8** Delays extracted from simulations for 1000 filtered sets of 16 tasks, with $U = 70\%$.



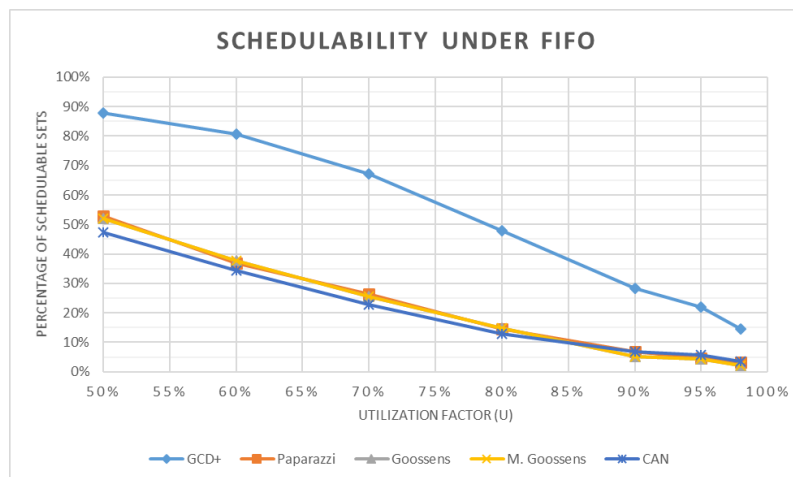
■ **Figure 9** Delays extracted from simulations for 1000 filtered sets of 16 tasks, with $U = 95\%$.

From the box plots, it can be seen that, for GCD+, every quartile (first, second and third) is significantly closer to the ideal value in every situation than those for the other heuristics, as well as the boundaries of the whiskers and the mean values. For utilization factors of 70%, GCD+ was able to keep the delay of the great majority of tasks under 60% and 80% of their periods in the case of sets of 8 and 16 tasks, respectively. Other heuristics, however, show whiskers going beyond the value of 100% in these same situations.

It is also noticeable how every heuristic method has more trouble reducing delays when the task count increases, notably with respect to the ratio between maximum response times and execution times – this ratio tripled when the task count doubled. Increasing the utilization factor is another factor that worsens the maximum delays for every heuristic method.



■ **Figure 10** Schedulability for 1000 filtered sets of 8 tasks, from $U = 50\%$ to 98% .



■ **Figure 11** Schedulability for 1000 filtered sets of 16 tasks, from $U = 50\%$ to 98% .

In order to evaluate the schedulability of these task sets with respect to the utilization factors, an implicit deadline was considered for every task, and if a set has a task that missed a deadline, it is considered unschedulable. Figure 10 and Figure 11 were then obtained for sets of 8 and 16 tasks, respectively. GCD+ shows a significant advantage relative to every other heuristic in every situation that is shown. For sets of 8 tasks, at 80% utilization factor, there is an increase of at least 50% in the number of schedulable sets, while for 90% utilization there are about twice as much schedulable sets when using GCD+. For sets of 16 tasks, the increase is even more pronounced: from about 2x at 50% utilization to about 4x at 90%.

The significant improvement seen in the presented results can be explained by certain limitations of the other methods. Paparazzi method tries to distribute the offsets without taking into account the relations between the messages. The CAN Message heuristic method, when trying to distribute the first calls over the maximum period, ignores the effects over the hyperperiod, during which two tasks that were initially put far away can fall into a critical case. The Goossens's heuristic method, although it considers some interactions between

pairs of tasks during the whole hyperperiod, has a trap: when several tasks have the same period, it will tend to assign the same offset to all of them, intentionally creating a critical case. Furthermore, its random component might lead to undesired critical cases. The fact that it does not take into account the execution times of tasks has no apparent effect in its results, since the Modified Goossens's Heuristic (which does take them into account) had the exact same results as the original method.

7 Case Study

An adapted real-world scenario was set up in order to measure the applicability of GCD+. A telemetry configuration file was used to define the messages that a Paparazzi autopilot implementation sends periodically, such that the telemetry is composed of the messages described in Table 2. The column c indicates the content size of the message, without adding any protocol header or tail.

■ **Table 2** Paparazzi telemetry values.

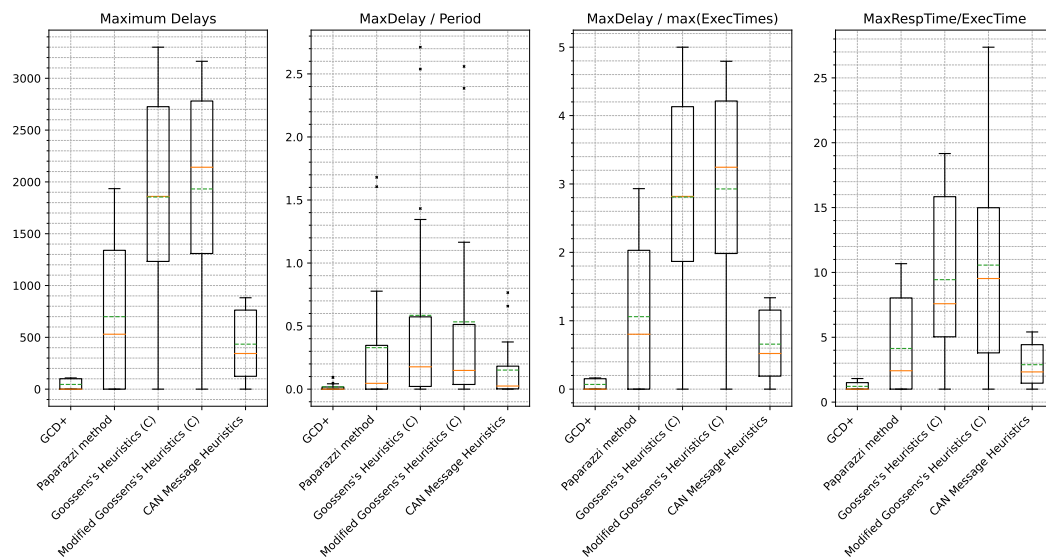
i	ID	T (s)	c (B)
1	ALIVE	2	17
2	ROTORCRAFT_FP	1	58
3	INS_REF	1	32
4	ROTORCRAFT_NAV_STATUS	1	15
5	ENERGY	1	21
6	DATALINK_REPORT	1	11
7	DL_VALUE	0.2	5
8	ROTORCRAFT_STATUS	0.2	20
9	STATE_FILTER_STATUS	0.2	4
10	AIR_DATA	0.2	28
11	INS	0.2	36
12	GPS_INT	0.1	57
13	IMU_GYRO_SCALED	0.04	12
14	IMU_ACCEL_SCALED	0.04	12
15	IMU_ACCEL_RAW	0.02	12
16	IMU_GYRO_RAW	0.02	12

The messages are sent to a UART channel, which is normally connected to an antenna to transmit the data, but for this test and to ignore any effects related to the transmission of radio waves, messages are read directly in the UART channel in our experiment. According to its protocol, every byte has a start and an end bit added to it, which makes it transmit 10 bits per byte. It was configured to transmit 57600 bits per second. In addition, the Paparazzi protocol used (Pprzlink V2.0) adds 8 bytes to each message for header and checksum².

Therefore, to use the time it takes for a bit to be transmitted as a unit of time, i.e., $1/57600$ second, values for periods have to be multiplied by 57600, and the amount of bytes in every message, after the 8 Pprzlink bytes are added, have to be multiplied by 10. With these converted values, we can analyse this case using GCD+. Its results are presented in Figure 12.

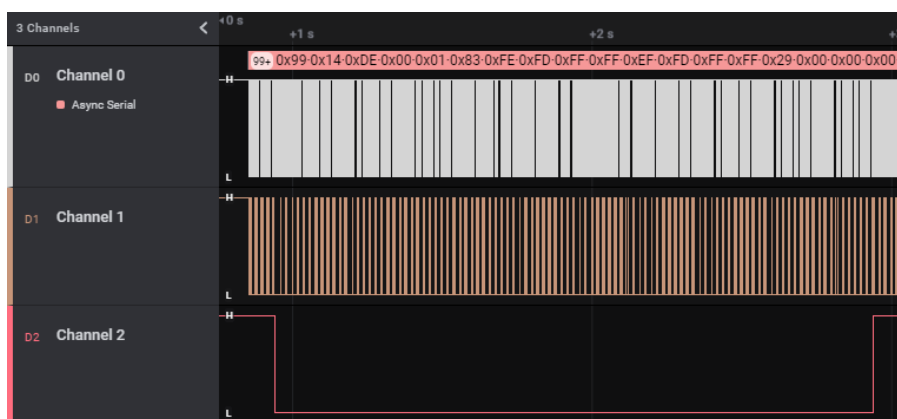
Figure 12 shows that, using GCD+, every message has a delay under 10% of its period, in comparison to 80%, 170% and even 270% in other heuristics. These delays represent at most 20% of the maximum length of other messages, while the third metric shows that there was a significant chaining of interferences for all the other heuristics. Also, deadline misses were registered for the methods Goossens and Paparazzi, while the system was schedulable using offsets from GCD+ and CAN Message heuristics.

² https://wiki.paparazziuav.org/wiki/Messages_Format

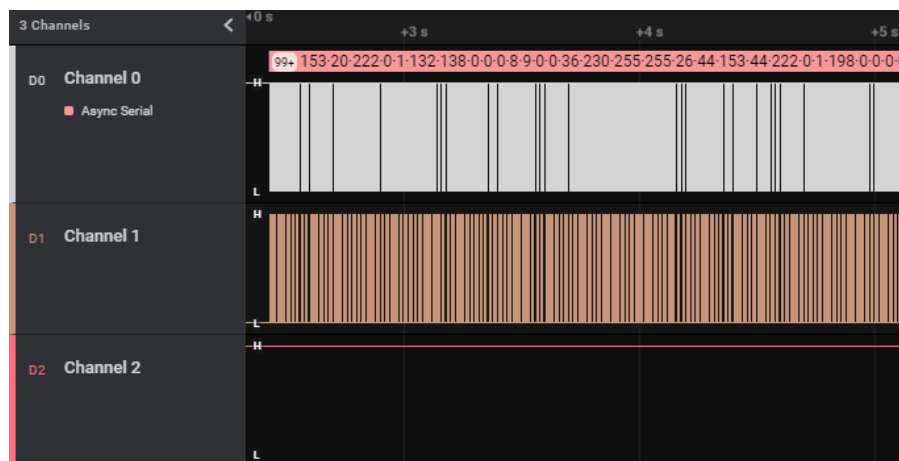


■ **Figure 12** Delays extracted from simulations for the case study.

From the physical analysis, before the implementation of GCD+, some messages were lost due to a full buffer, as it can be seen in Figure 13. The figure is a screen capture from the software Logic 2, for analysing the data acquired from the UART setup. The first line (Channel 0) represents the raw bits sent in the UART port; the second one (Channel 1) goes from a low to a high state every time the autopilot blocks the UART buffer, as it is writing a message to be sent in the channel, and from high to low when it releases the buffer; the third line (Channel 2) goes from high to low or from low to high when the buffer is already full (a lot of messages in the queue) and, hence, the buffer refused to accept a message from the autopilot (i.e., a message loss). It can be seen that there is a message loss at every 2 seconds. This problem was avoided with the offsets given by the heuristics presented in this paper, as it can be seen in Figure 14, where no message is ever lost. The image shows less than 3 seconds of data capturing, but the complete test was made for 20 seconds.



■ **Figure 13** Screen capture from the analysis of a Paparazzi telemetry case with a message loss at every 2 seconds.



■ **Figure 14** Screen capture from the analysis of an improved Paparazzi telemetry case with no more message losses.

This case study confirms what can be seen in simulations. Moreover, the case studies in telemetry usually confirm the fact that the global GCD is large, giving an advantage to GCD+.

8 Conclusion

In this article, we have proposed GCD+ a new method for finding suitable offsets for tasks in execution or messages in transmission under FIFO scheduling. It was compared in simulations to State-of-the-Art methods using a random task set generator and in a real case, presenting better results than the others in both scenarios. In addition, the new method was compared against the one used in Paparazzi in a physical setup of the autopilot, confirming the better result obtained in the simulation. GCD+ proved its efficiency, notably for the cases of a semi-harmonic set where the maximum execution time is not greater than the GCD of all task periods.

GCD+ can be extended in the future to support other cases. For example, GCD+ could be extended to handle multi-periodic precedence constraints, such as described in [14]. It would allow applying this method to the computation of offsets in the case of monolithic periodic tasks integrating a static scheduler, which are central in most UAV autopilots. Also, the integration of GCD+ into the configuration tool of Paparazzi is planned in the near future.

References


- 1 Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. The case for fifo real-time scheduling. Technical report, University of Luxembourg, 2016.
- 2 Chaitanya Belwal and Albert MK Cheng. Generating bounded task periods for experimental schedulability analysis. In *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pages 249–254. IEEE, 2011.
- 3 Pascal Brisset, Antoine Drouin, Michel Gorraz, Pierre-selim Huard, and Jeremy Tyler. The Paparazzi Solution. *HAL*, 2006.
- 4 Vicent Brocal, Patricia Balbastre, Rafael Ballester, and Ismael Ripoll. Task period selection to minimize hyperperiod. In *ETFA2011*, pages 1–4. IEEE, 2011.

- 5 Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.
- 6 Joel Goossens and Christophe Macq. Limitation of the hyper-period in real-time periodic task set generation. In *In Proceedings of the RTS Embedded System (RTS'01)*. Citeseer, 2001.
- 7 Joël Goossens. Scheduling of offset free systems. *Real-Time Systems*, 24(2):239–258, 2003.
- 8 Mathieu Grenier, Joël Goossens, and Nicolas Navet. Near-optimal fixed priority preemptive scheduling of offset free systems. In *14th International Conference on Real-Time and Networks Systems (RTNS'06)*, pages 35–42, 2006.
- 9 Mathieu Grenier, Lionel Havet, and Nicolas Navet. Pushing the Limits of CAN - Scheduling Frames with Offsets Provides a Major Performance Boost. *4th European Congress on Embedded Real Time Software (ERTS 2008)*, 2008.
- 10 David Griffin, Iain Bate, and Robert I Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 76–88. IEEE, 2020.
- 11 Gautier Hattenberger, Murat Bronz, and Michel Gorraz. Using the paparazzi UAV system for scientific research. In *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, page 247, 2014.
- 12 Mitra Nasri, Robert I Davis, and Björn B Brandenburg. Fifo with offsets: High schedulability with low overheads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 271–282. IEEE, 2018.
- 13 Mitra Nasri and Gerhard Fohler. An efficient method for assigning harmonic periods to hard real-time tasks with period ranges. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 149–159. IEEE, 2015.
- 14 Thanh-Dat Nguyen, Yassine Ouhammou, Emmanuel Grolleau, Julien Forget, Claire Pagetti, and Pascal Richard. Design and analysis of semaphore precedence constraints: A model-based approach for deterministic communications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 231–236. IEEE, 2018.
- 15 Réda Nouacer, Mahmoud Hussein, Huascar Espinoza, Yassine Ouhammou, Matheus Ladeira, and Rodrigo Castiñeira. Towards a framework of key technologies for drones. *Microprocessors and Microsystems*, 77:103142, 2020.
- 16 Paparazzi developers. Paparazzi home page. <https://paparazziuav.org>. Accessed: 02-02-2022.
- 17 Paparazzi developers. Paparazzi offset generation source code. https://github.com/paparazzi/paparazzi/blob/master/sw/tools/generators/gen_periodic.ml. Accessed: 02-02-2022.
- 18 Rodolfo Pellizzoni and Giuseppe Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems*, 30(1-2):105–128, 2005. doi:10.1007/s11241-005-0506-x.
- 19 Ismael Ripoll and Rafael Ballester-Ripoll. Period selection for minimal hyperperiod in periodic task systems. *IEEE Transactions on Computers*, 62(9):1813–1822, 2012.
- 20 Jia Xu. A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems. In *Proceedings of 2010 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, pages 288–294. IEEE, 2010.

Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks

Geoffrey Nelissen ✉ 

Eindhoven University of Technology, The Netherlands

Joan Marcè i Igual ✉ 

Eindhoven University of Technology, The Netherlands

Mitra Nasri ✉ 

Eindhoven University of Technology, The Netherlands

Abstract

Gang scheduling has long been adopted by the high-performance computing community as a way to reduce the synchronization overhead between related threads. It allows for several threads to execute in lock steps without suffering from long busy-wait periods or be penalized by large context-switch overheads. When combined with non-preemptive execution, gang scheduling significantly reduces the execution time of threads that work on the same data by decreasing the number of memory transactions required to load or store the data. In this work, we focus on two main types of gang tasks: *rigid* and *moldable*. A moldable gang task has a presumed known minimum and maximum number of cores on which it can be executed at runtime, while a rigid gang task always executes on the same number of cores. This work presents the first response-time analysis for non-preemptive moldable gang tasks. Our analysis is based on the notion of schedule abstraction; a new approach for response-time analysis with the promise of high accuracy. Our experiments on periodic rigid gang tasks show that our analysis is 4.9 times more successful in identifying schedulable tasks than the existing *utilization-based test* for rigid gang tasks.

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

Keywords and phrases schedulability analysis, response time analysis, moldable gang tasks, rigid gang tasks, schedule abstraction graph, multiprocessor, non-preemptive

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.12

Funding *Joan Marcè i Igual*: This work was made with the support of the NWO SAM-FMS project (project number 17931) as a part of the MASCOT program.

Mitra Nasri: This work was made with the support of the EU ECSEL Joint Undertaking under grant agreement no 101007260 (project TRANSACT).

1 Introduction

Gang scheduling is a scheduling approach that groups and executes parallel threads as a “gang”. A gang of threads reserves a set of cores for their execution. The threads have exclusive access to those cores from the moment they start to execute until they complete or get preempted. Since the early 80s, gang scheduling has been adopted by the high-performance computing community [23] as a way to reduce synchronization overheads between related threads, and to optimize the access time to shared data in data-intensive applications [12]. It allows for many threads to execute in lock steps without suffering from long busy-wait periods or be penalized by large context-switch overheads. Furthermore, it reduces the number of memory transactions by allowing the application to load its data only once for all threads rather than once per thread.



© Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri; licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 12; pp. 12:1–12:22



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Gang scheduling is a versatile model and has various applications. It is used to assign resources to applications in data centers, to schedule hardware tasks on *field programmable gate arrays* (FPGAs) [5], and is also used in *graphics processor units* (GPUs) [2] where each kernel thread-block needs a fixed number of cores before its execution starts.

The efficiency of gang scheduling improves significantly when it is combined with non-preemptive execution. It then results in smaller execution times as it eradicates the need to reload data into memory after each preemption. This is usually a crucial factor to optimize the worst-case execution time of a gang task, which often treats large amounts of data. Despite these benefits, to the best of our knowledge, no theoretical analysis exists for obtaining a safe upper bound on the worst-case response time of gang tasks under *non-preemptive* scheduling.

In this work, we provide the first *worst-case response-time* (WCRT) analysis for two generic classes of gang tasks, namely, *rigid* and *moldable*, where tasks are scheduled by a non-preemptive *job-level fixed-priority* (JLFP) scheduling policy (e.g., non-preemptive deadline monotonic (DM), earliest-deadline first (EDF), etc.). A *rigid* gang task requires a fixed number of cores to execute its threads. Thus, a job released by a rigid gang task cannot start its execution until at least the number of cores it requires are available. A *moldable* gang task, however, uses a minimum and a maximum number of cores on which it may be executed. Each job of a moldable task may then be allocated a different number of cores by the scheduler depending on how many cores are available at its start time. As a result, the actual execution time of a job of a moldable task depends on the actual number of cores allocated by the scheduler to that job at runtime. It is worth noting that rigid gang is only a particular case of moldable gang. Yet, due to its potentials to adapt its level of parallelism according to the number of cores available, the analysis of moldable gang tasks is a much more challenging problem than that of rigid gang tasks.

Related work. Gang tasks, or so called “coscheduled threads”, were introduced by Ousterhout et al. [23] in 1982. From the late 80s, it has been known that the optimal scheduling of non-preemptive gang (NPGang) tasks is an NP-complete problem as it requires solving a bin packing problem at the core [6]. Since then, further studies have focused on designing more efficient scheduling algorithms either to improve the average-case response time [32] or to reduce the effect of fragmentation [11], which happens when the number of idle cores is not enough to start executing any of the pending jobs.

Work on real-time rigid gang. Goossens et al. [13] show that preemptive gang scheduling under a JLFP policy is not *sustainable* [7] w.r.t. execution time variation. Since then, two schedulability tests [9,13] and one optimal scheduling policy [15] have been introduced for preemptive rigid gang under FP scheduling. However, neither of the tests nor the optimal scheduling policy are applicable on non-preemptive gang tasks. Recently, Dong and Liu [10] introduced a utilization-based test for non-preemptive sporadic gang tasks. That work is the closest to ours as it considers non-preemptive gang tasks. However, it cannot be used to compute an upper bound on the response-time of the tasks as it only outputs a yes/no answer that indicates whether the task set is schedulable or not. It is also limited to rigid gang tasks and cannot analyze moldable gang tasks.

Work on real-time moldable and malleable gang. Malleable gang tasks are a generalization of moldable gang tasks according to which a job may change its level of parallelism during its execution. Kato et al. [16] and Richard et al. [26] propose sufficient schedulability tests for preemptive moldable gang scheduling under global EDF. Berten et al. [4] propose a greedy scheduling algorithm and its schedulability test for preemptive moldable gangs. Collette et al. [8] present a feasibility test as well as a scheduling algorithm that minimizes the

number of cores required to schedule malleable gang tasks. In their work, they also show that EDF is not an optimal policy for malleable gang. However, these results are again limited to preemptive tasks and have not been extended (and cannot be easily adapted) to non-preemptive scheduling.

Work on bundle scheduling. In 2019, Wasly et al. [30] have extended the classical rigid gang task model and proposed the *bundled task model* (BTM) along with a sufficient schedulability test. Bundle tasks are modelled as a succession of “bundles” with precedence constraints between them. Each bundle is preemptively scheduled and follows a rigid gang model. However, each bundle may request a different number of cores than other bundles, thereby providing a mean to change the parallelism of the task throughout its execution. Nevertheless, similar to prior work, the existing tests for bundle scheduling have been designed for preemptive execution and are not applicable to non-preemptive scheduling.

Work on schedule-abstraction-based analyses. Our response-time analysis for non-preemptive gang tasks is based on the notion of *schedule abstraction*, a new type of response-time analysis that provides highly accurate schedulability results (as shown in [31]). It was first proposed in 2017 [18] and, since then, has been applied to various response-time analysis problems for non-preemptive scheduling on single-core [18,24,25] and multicore platforms [19,20,22], and is also being extended to the analysis of Ethernet TSN [28].

Contributions. In this paper, we propose a new analysis that uses the idea of schedule abstraction [19] to derive the best-case and worst-case response times (BCRT and WCRT) of a set of periodic rigid and/or moldable gang tasks scheduled by a non-preemptive JLFP scheduling policy (detailed in Section 2). To the best of our knowledge, this is the first response-time analysis for non-preemptive moldable (and rigid) gang tasks.

2 System Model

2.1 Platform and Task Model

We assume a platform made of m identical cores on which we execute a set of n non-preemptive periodic moldable gang tasks. We consider periodic tasks since they are present in more than 80% of real-time systems according to the recent survey of Akesson et al. [1] on the industry practice in real-time systems. Each task τ_k ($1 \leq k \leq n$) periodically releases non-preemptive moldable gang *jobs*. Whenever the scheduler dispatches a job J_i , it can chose on how many cores to execute J_i out of a set \mathcal{P}_i (see Section 2.2 for a detailed description of the scheduler). The set \mathcal{P}_i contains all the valid options to parallelize the execution of a job J_i . For each possible number of cores $p \in \mathcal{P}$ that may be allocated to J_i , job J_i will execute for a minimum of $C_i^{\min}(p)$ and a maximum of $C_i^{\max}(p)$ time units before completing its execution. In other words, $C_i^{\min}(p)$ and $C_i^{\max}(p)$ are the best- and worst-case execution times (BCET and WCET) of the job as a function of the number of cores on which it executes. We assume that jobs may experience a bounded release jitter, that may arise from timer inaccuracy, interrupt latency, or queuing delays. That is, a job J_i may be released at any time within an interval $[r_i^{\min}, r_i^{\max}]$ where r_i^{\min} is its *earliest release time* and r_i^{\max} is its *latest release time*.

In sum, each job $J_i \in \mathcal{J}$ is then defined by the tuple $([r_i^{\min}, r_i^{\max}], \bar{C}_i^{\min}, \bar{C}_i^{\max}, \mathcal{P}_i, d_i)$ where r_i^{\min} and r_i^{\max} are the earliest and latest release times of J_i ; d_i is its absolute deadline; \mathcal{P}_i is the set of core counts on which J_i may execute; \bar{C}_i^{\min} and \bar{C}_i^{\max} are vectors such that each entry $C_i^{\min}(p)$ and $C_i^{\max}(p)$ contains the BCET and WCET of J_i on p cores, respectively.

For ease of notation, we define $m_i^{\min} = \min\{p \mid p \in \mathcal{P}_i\}$ and $m_i^{\max} = \max\{p \mid p \in \mathcal{P}_i\}$ as the minimum and maximum number of cores on which job J_i may execute. We also define the function $next_i(p)$ for all values $p < m_i^{\max}$ as a function that returns the smallest number of cores larger than p on which job J_i can execute. That is, $next_i(p) = \min\{k \in \mathcal{P}_i \mid k > p\}$. Without loss of generality, we assume that $1 \leq m_i^{\min} \leq m_i^{\max} \leq m$ for all jobs. That is, job J_i cannot execute on less than one core and cannot request more cores than the number of cores in the platform. We also assume that the values in \bar{C}_i^{\min} and \bar{C}_i^{\max} are monotonically decreasing. That is, a job may execute on more cores only if it decreases its execution time.

As mentioned before, rigid gang is a special form of moldable gang, namely, if $m_i^{\min} = m_i^{\max}$ (and thus $|\mathcal{P}_i| = 1$) for a job J_i , then the job is said to be **rigid**, otherwise, it is **moldable**. If all jobs released by a task are rigid, then the task is rigid.

Our response-time analysis is based on schedule abstraction which must be applied on a finite observation window (or job set). For **periodic tasks** with synchronous releases, constrained deadlines (i.e., deadlines smaller than or equal to periods) and with or without release jitter, the length of the observation window is the task set's hyperperiod H , i.e., the least common multiple of all tasks' periods [14]. Thus, the job set \mathcal{J} must include all jobs released by all tasks during the interval $[0, H)$. Further discussions on how to create job sets for periodic tasks with offsets or arbitrary deadlines are found in [18] and [14].

It is worth noting that despite focusing on periodic tasks, our analysis is applicable to any arbitrary job set as well. This is helpful, for example, to analyze tasks with bursty release patterns or with complex arrival models such as generalized multi-frame tasks [21].

2.2 Scheduler Model

Jobs are scheduled non-preemptively using a work-conserving global job-level fixed priority (JLFP) scheduling algorithm that is assumed to follow the following set of rules:

- ▶ **Rule 1.** A job J_i is considered ready at time t if and only if it is released at or before t , it is not yet completed at t and it is not already executing at time t .
- ▶ **Rule 2.** A job J_i is considered eligible to be dispatched at time t if and only if it is ready at time t and there are at least m_i^{\min} cores available at time t .
- ▶ **Rule 3.** The scheduler is invoked whenever a job is released or a job completes.
- ▶ **Rule 4.** At every invocation of the scheduler, the highest priority eligible job is chosen to be dispatched.
- ▶ **Rule 5.** The dispatched job is assigned a number of cores equal to the maximum value in \mathcal{P}_i that is smaller than or equal to the number of free cores at the time at which it is dispatched (i.e., it always executes on as many cores as possible so as to maximize its parallelism).
- ▶ **Rule 6.** The number of cores allocated to a job cannot change during its execution
- ▶ **Rule 7.** The execution of a job cannot be preempted once it started.
- ▶ **Rule 8.** No core may remain idle as long as there are eligible jobs to be dispatched.

Since we assume a JLFP scheduling algorithm, we use the notations hp_i and lp_i to refer to the set of higher and lower priority jobs than job J_i , respectively.

3 Worst-Case Response-Time Analysis

To check the schedulability of a task set, we compute an upper bound on the worst-case response time (WCRT) of each task by calculating an upper bound on the WCRT of each job of the tasks in the observation window (e.g., hyperperiod). If the WCRT of a task is not larger than its deadline, then the task is deemed *schedulable*.

As already mentioned, we use a *schedule abstraction*-based analysis [18–20] as opposed to, e.g., a *critical-instant*-based analysis [3], in order to compute the WCRT of every job in a job set \mathcal{J} . In this section, we first explain preliminaries on schedule abstraction (Sec. 3.1) and the challenges to build such an analysis for gang tasks (Sec. 3.2). We then elaborate on how we model the *system state* for gang tasks (Sec. 3.3) and provide a *top-down description* of the analysis in Sec. 3.4. We then discuss the details of the algorithm in Sec. 4 and 5

3.1 Preliminaries on Schedule-Abstraction Technique

Motivation. Schedule abstraction [18–20] is a recently developed technique to analyze the best- and worst-case response times of a set of jobs. It efficiently explores the set of “possible” schedules that can be generated by the job set under a given scheduling policy. A recent comparison between the schedule abstraction and an exact response-time analysis in UPPAAL (a generic formal verification tool) on a set of independent non-preemptive periodic tasks scheduled by global fixed-priority scheduling shows that it is more than *three-order-of-magnitude* faster than UPPAAL while having almost 100% accuracy [31], i.e., it is able to detect almost all schedulable task sets. Since then, further improvements using partial-order reduction techniques [24,25] have yet accelerated the analysis by five additional orders-of-magnitude in comparison to [18]. This impressive gain, which now allows to analyze hundreds of tasks non-preemptively scheduled on a single core platform in a matter of seconds, was achieved at the cost of a negligible added pessimism on the WCRT estimation. However, this improved version of the analysis is currently limited to the analysis of single-core platforms and is therefore not directly applicable nor extendable to the analysis of gang scheduling on multiprocessor platforms yet. For this reason, in this section, we focus on the original idea of the schedule-abstraction graph (SAG) as it was presented in [19].

In terms of accuracy, it has been shown that the SAG analysis is far less pessimistic than critical-instant-based analyses when applied on more complex problems such as the response-time analysis of parallel tasks (with directed-acyclic graph dependencies) on multiprocessor platforms [19]. Namely, for platforms with 4 to 16 cores, the schedule-abstraction technique is *2 to 4.3 times more successful* in identifying schedulable task sets than the analysis of [27] based on the critical-instant theory.

Key idea. Assuming a deterministic scheduling policy and no uncertainty in the release and execution times of the jobs, a job set will have only one possible schedule (which can also be obtained by *simulating* the schedule of the job set during the observation window). Under uncertainties, however, multiple schedules could occur. The schedule-abstraction technique combines these schedules whenever they relate to the same job dispatching ordering on the platform. To defer the state-space explosion and hence scale to reasonably large system sizes, the schedule-abstraction technique uses two key ideas: (i) combining (and abstracting) schedules that share the same jobs dispatching ordering and (ii) introducing merging techniques that allow combining partial schedules (or job orderings) whose future can be explored together (e.g., because they are followed by the same future scheduling decisions).

Schedule-abstraction graph (SAG). It abstracts all possible schedules of a given job set \mathcal{J} in the form of a directed graph $\mathcal{G} = \langle V, E \rangle$, where V is the set of vertices (referred to as nodes) and E is the set of edges each of which connects a pair of vertices to each other. A path in the graph \mathcal{G} represents a possible order of scheduling decisions taken by the scheduler. Each node $v \in V$ represents *the set of system states that may result from the scheduling decisions encoded on the paths that reach to v* . A directed edge connecting a node v to a node v' in \mathcal{G} represents a scheduling decision (i.e., dispatching of a job) taken by the scheduler that brings the set of system states represented by v to a subset of the system states represented by v' . A scheduling decision is equivalent to *dispatching* a job on the platform. Hence, each edge in the graph is labeled by a job. Since the existing analyses that use the schedule-abstraction graph are designed for non-preemptive jobs, the best- and worst-case completion time of a job only depends on its start time, and thus on the system state v after which it is dispatched. The earliest and latest completion time of the dispatched job is also recorded on the edge of the graph.

Given the potential uncertainty on the release and execution times of the jobs, a job J_i may appear in different places in the schedule abstraction graph (i.e., after different sequences of scheduling decisions). Therefore, the WCRT of a job is given by the largest completion time recorded on all edges referencing that job in the SAG.

3.2 Challenges in Analyzing Gang Tasks using Schedule Abstraction

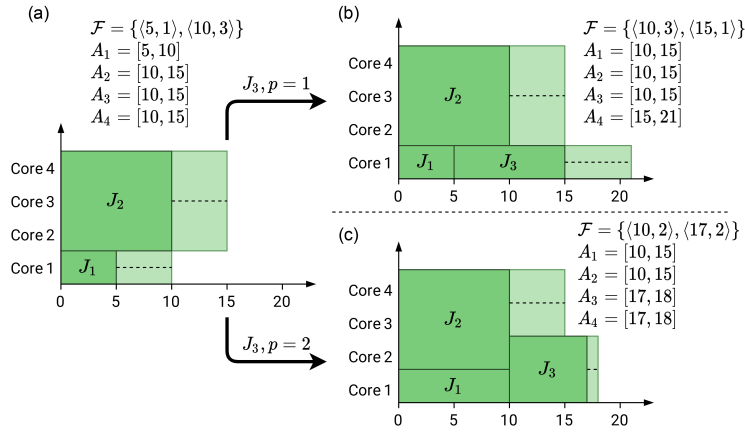
Two of the clear differences between this work and previous works on schedule abstraction [18–20,22] is that: (1) a job of a gang task may need more than one core to start executing, hence cores may remain idle even when there is pending workload, and (2) a single job may release (or free) more than one core simultaneously. These two particularities imply that the time at which different cores start and stop executing workload is somewhat synchronized. Thus, there is a need for system state representation that allows us to keep track of such synchronization between cores in order to correctly and efficiently analyze possible schedules. Furthermore, due to reason (1), the existing rules for analyzing global JLFP scheduling policies are not applicable to the gang scheduling algorithm introduced in Sec. 2.2. This means that new expansion and merge rules must be designed.

3.3 System-State Representation

To keep track of the number of cores claimed by a gang job at the time it is being dispatched on the platform and hence to address challenge (2) in Sec. 3.2, we encode additional information on the edges of the graph. In our new schedule abstraction, an edge between nodes v and v' will not only be labeled with the job J_i that is being dispatched after state v but it also includes the number of cores that are claimed by J_i at the time it has been dispatched.

To address challenge (1) in Sec. 3.2, we develop a new model to encode a system state using four data: (i) the set $\mathcal{S}(v)$ of all jobs that have already been dispatched to reach the system state represented by node v , (ii) the set of all possible instants at which cores may become available to execute new workload, and (iii) the number of cores that might be freed at the exact same time, and (iv) the time at which those cores become available (recall that a gang job executes on p parallel cores and thus releases p cores when it completes).

To encode (ii), we use m intervals. Each interval $A_k(v) = [A_k^{\min}(v), A_k^{\max}(v)]$ (with $1 \leq k \leq m$) encloses all the time instants at which k cores may become available to execute new workload in node v in \mathcal{G} . That is, $A_k^{\min}(v)$ is the time until which there are certainly *less than* k cores available, and $A_k^{\max}(v)$ is the time by which *at least* k cores are certainly



■ **Figure 1** Example of two possible execution scenarios for J_3 and their resulting system states. (a) initial state, (b) J_3 scheduled with $p = 1$, (c) J_3 scheduled with $p = 2$.

available to execute new jobs. We call $A_k^{\min}(v)$ and $A_k^{\max}(v)$ the earliest and latest availability time of k cores for system state v , and we call $A_k(v)$ the availability interval of k cores in state v . In the following, when there is no ambiguity, we do not explicitly write the system state v when referring to A_k , A_k^{\min} and A_k^{\max} .

To encode (iii) and (iv) and thus know how many cores may be freed simultaneously by a single job at what time, we store a set of pairs of values $\mathcal{F} = \{\langle f_1(v), M_1(v) \rangle, \langle f_2(v), M_2(v) \rangle, \dots\}$ such that each pair $F_\ell(v) = \langle f_\ell(v), M_\ell(v) \rangle$ has the following meaning: at least $M_\ell(v)$ cores will be freed by a single job no earlier than time $f_\ell(v)$. By definition, we have that the total number of cores that may be freed is equal to m , i.e., $\sum_{\ell>0} M_\ell(v) = m$, and the earliest time $f_\ell(v)$ at which a group of cores may be freed must also correspond to the earliest time at which some core may become available, i.e., $\forall \ell, \exists k$ s.t. $f_\ell(v) = A_k^{\min}(v)$.

► **Example 1.** Figure (1a) shows a system with $m = 4$ cores where two jobs have been scheduled: J_1 on one core, and J_2 on three cores. J_1 must finish within the interval $[5, 10]$, and J_2 must finish within $[10, 15]$. Therefore, one core becomes possibly available at time 5 and three additional cores become possibly available simultaneously at time 10. Similarly, one core is certainly available at time 10 and three more cores become certainly available at time 15. Thus, $\mathcal{F} = \{\langle 5, 1 \rangle, \langle 10, 3 \rangle\}$, $A_1 = [5, 10]$, $A_2 = [10, 15]$, $A_3 = [10, 15]$ and $A_4 = [10, 15]$.

Now, assume that a job J_3 is released at time 1 with $\mathcal{P}_3 = \{1, 2\}$, $C_3^{\min}(1) = 10$, $C_3^{\min}(2) = 7$, $C_3^{\max}(1) = 11$ and $C_3^{\max}(2) = 8$. Two execution scenarios are possible, hence two new system states are created. If J_1 finishes before J_2 then one core will be freed and J_3 will be scheduled with $p = 1$. This means that J_3 starts executing at the earliest at time 5 and at the latest at time 10. For $p = 1$ we know that the best-case and worst-case execution time of J_3 is 10 and 11, respectively. Therefore, the finish time interval of J_3 is $[15, 21]$. Then, as shown in Figure (1b), three cores become possibly available at time 10 and one additional core becomes possibly available at time 15. Therefore, we have $\mathcal{F} = \{\langle 10, 3 \rangle, \langle 15, 1 \rangle\}$, and the availability intervals become $A_1 = [10, 15]$, $A_2 = [10, 15]$, $A_3 = [10, 15]$ and $A_4 = [15, 20]$.

However, in another execution scenario where J_1 and J_2 finish at the same time, J_3 will be dispatched on $p = 2$ cores. This can only happen at time 10. For $p = 2$, the execution-time interval of J_3 is $[7, 8]$, leading to the finish-time interval $[17, 18]$. Thus, the new availability intervals are $A_1 = [10, 15]$, $A_2 = [10, 15]$, $A_3 = [17, 18]$, $A_4 = [17, 18]$ and $\mathcal{F} = \{\langle 10, 2 \rangle, \langle 17, 2 \rangle\}$ as shown in Figure (1c).

■ **Algorithm 1** Algorithm to generate a schedule abstraction graph.

Input : Job set \mathcal{J}
Outputs: Bounds on the BCRT and WCRT of every job in \mathcal{J} ;

- 1 $\forall J_i \in \mathcal{J}, BCRT_i \leftarrow \infty, WCRT_i \leftarrow 0$;
- 2 initialize \mathcal{G} with a root node v_1 with $\mathcal{S}(v_1) = \emptyset, A_k(v_1) = [0, 0], \forall 1 \leq k \leq m$, and $\mathcal{F}(v_1) = \{(0, m)\}$;
- 3 **while** \exists a leaf node v s.t. $\mathcal{S}(v) \neq \mathcal{J}$ **do**
- 4 $P \leftarrow$ the shortest path from v_1 to a leaf node v ;
- 5 $v \leftarrow$ the leaf vertex of P ;
- 6 **for each** job $J_i \in \mathcal{J} \setminus \mathcal{S}$ **do**
- 7 **for** $\forall p \in \mathcal{P}_i$ **do**
- 8 **if** J_i may be dispatched next on p cores **then**
- 9 Compute the earliest and latest finish time EFT_i^p and LFT_i^p of J_i on p cores;
- 10 $BCRT_i \leftarrow \min\{EFT_i^p - r_i^{min}, BCRT_i\}$;
- 11 $WCRT_i \leftarrow \max\{LFT_i^p - r_i^{min}, WCRT_i\}$;
- 12 Build the next states using Alg. 2;
- 13 Try to merge the new system states with other nodes in \mathcal{G} (Sec. 5);
- 14 **return** $BCRT_i, WCRT_i$ for all $J_i \in \mathcal{J}$;

3.4 Constructing the Schedule Abstraction Graph

The schedule-abstraction graph for a job set \mathcal{J} is built according to Algorithm 1 following a breadth-first strategy. The algorithm starts by building an initial node v_1 representing the state of the system when no job started to execute yet. Therefore, v_1 is initialized with an empty set of scheduled jobs ($\mathcal{S}(v_1) = \emptyset$), with all cores potentially and certainly available at time 0 (i.e., $A_k(v_1) = [0, 0], \forall 1 \leq k \leq m$), and with all m cores being freed simultaneously at time 0 (i.e., $\mathcal{F}(v_1) = \{(0, m)\}$).

Then, for each node in the graph that has not been analyzed yet (Line 3), Algorithm 1 checks which jobs that have not been scheduled yet may be dispatched next by the scheduler and on how many cores they may be executed (Lines 6 to 8). For each such job J_i and number of cores p , the earliest and latest completion times EFT_i^p and LFT_i^p of the job are computed (Line 9). If the computed completion times result in larger (smaller, respectively) worst-case (best-case, respectively) response times for J_i than those computed on other path of the graph (i.e., for other sequences of scheduling decisions), then it updates the recorded values $WCRT_i$ and $BCRT_i$ for their WCRT and/or BCRT (Lines 10 to 11). Finally, Algorithm 1 uses Algorithm 2 presented later in Section 4.3 to build *all* system states that may result from scheduling J_i on p cores in state v (Line 12) and hence expand the graph. Section 4 provides more details and explanations about this *expansion phase*.

To defer a potential state-space, Algorithm 1 tries to merge the newly created nodes with existing ones and hence reduce the number of branches in the graph (Algorithm 1). Section 5 provides more details about this *merge phase*. Finally, the algorithm stops when no more job can be added to any of the leaf nodes, namely, when the set of scheduled jobs in each leaf node is equal to the set of input jobs (i.e., $\mathcal{S}(v) = \mathcal{J}$).

4 Expansion Phase

The expansion phase has three consecutive steps: **(1)** for each job J_i that was not dispatched yet (i.e., $J_i \notin \mathcal{S}(v)$) and for each possible number of cores $p \in \mathcal{P}_i$, check whether J_i may be the next job dispatched by the scheduler on exactly p cores in state v , **(2)** if J_i may be

dispatched next, compute the earliest and latest finish times of J_i , and finally, **(3)** build the new system states resulting from the scheduler dispatching J_i on p cores in state v . We discuss each of those steps in Sections 4.1–4.3.

4.1 Dispatch Condition

To check whether a job J_i may be the next job dispatched by the scheduler on p cores in system state v , we first compute the earliest time $EST_i^p(v)$ at which that job would be starting to execute on p cores if it was the only job left to execute. Then, we compute the latest time $LST_i^p(v)$ at which it must have started *in order to be the first job dispatched by the scheduler* considering all the other pending jobs in the system. It is crucial to understand that $LST_i^p(v)$ is defined under the condition that J_i is going to be the first job being dispatched after the state v . Scenarios where J_i is not the first job to be dispatched after the state v will be automatically explored during future expansions of the graph.

If $LST_i^p(v)$ is larger than or equal to $EST_i^p(v)$, then there exists an execution scenario in which J_i may be the next job dispatched on p cores by the scheduler. Otherwise, if $LST_i^p(v) < EST_i^p(v)$, then either J_i cannot be dispatched on p cores or there will always be *another job* dispatched *before* J_i .

4.1.1 Earliest Start Time

The earliest start time $EST_i^p(v)$ of J_i on p cores ($p \in \mathcal{P}_i$) depends on the following properties:

- (i) by Rule 2, J_i cannot start before it is released (i.e., $EST_i^p(v) \geq r_i^{\min}$);
- (ii) there must be at least p cores available to start to execute J_i on p cores; and
- (iii) by Rule 5, if $p < m_i^{\max}$, then less than $next_i(p)$ cores may be available when J_i is dispatched (otherwise, by Rule 5, it would be dispatched on more than p cores).

Hence, as proven in Lemma 2 below, we can compute $EST_i^p(v)$ as follows

$$EST_i^p(v) = \max\{r_i^{\min}, t_{gang}^p(v)\} \quad (1)$$

$$t_{gang}^p(v) = \begin{cases} A_p^{\min}(v) & \text{if } p = m_i^{\max}, \\ A_p^*(v) & \text{otherwise.} \end{cases} \quad (2)$$

where we define $A_p^*(v)$ as the earliest time at which *at least* p cores but less than $next_i(p)$ cores may become available. Note that $A_p^*(v)$ is different from $A_p^{\min}(v)$ in the sense that $A_p^{\min}(v)$ only ensures that at least p cores are available but does not enforce that there are less than $next_i(p)$ available cores. Section 4.1.2 will explain how to compute $A_p^*(v)$.

► **Lemma 2.** *A job J_i cannot start executing on exactly p cores before time $EST_i^p(v)$.*

Proof. We analyze two cases:

Case 1. If $p = m_i^{\max}$, job J_i cannot start before to be released (i.e., before r_i^{\min}) and cannot start until at least p cores are available (i.e., at $A_p^{\min}(v)$). Thus, J_i cannot start before $\max\{r_i^{\min}, A_p^{\min}(v)\}$, thus proving the claim for the case $p = m_i^{\max}$ in Equation (1).

Case 2. If $p < m_i^{\max}$, again, job J_i cannot start before r_i^{\min} and at least p cores are available. Furthermore, by Rule 5, J_i cannot start executing on p cores if $next_i(p)$ or more cores are available. Thus, J_i cannot start before $\max\{r_i^{\min}, A_p^*(v)\}$. ◀

4.1.2 Computing $A_p^*(v)$

Let $A_k^{exact}(v)$ be the earliest time at which *exactly* k cores may become available. Then, the earliest time at which *at least* p cores but less than $next_i(p)$ cores may become available is given by $A_p^*(v) = \min_k \{A_k^{exact}(v) \mid p \leq k < next_i(p)\}$.

The value of $A_k^{exact}(v)$ can be computed from the information available in $\mathcal{F}(v)$. Specifically, we must find a subset $\mathcal{F}' \subseteq \mathcal{F}(v)$ such that $\sum_{F_\ell \in \mathcal{F}'} M_\ell = k$ and for which the time at which the latest core is freed (i.e., the time given by $\max_{F_\ell \in \mathcal{F}'} \{f_\ell\}$) is minimum. The earliest time $A_k^{exact}(v)$ at which exactly k cores may become available is then equal to the time at which the last core in \mathcal{F}' is freed, i.e., $A_k^{exact}(v) = \max_{F_\ell \in \mathcal{F}'} \{f_\ell\}$.

If there is no subset $\mathcal{F}' \subseteq \mathcal{F}(v)$ such that $\sum_{F_\ell \in \mathcal{F}'} M_\ell = k$, then there is no possibility for exactly k cores to become simultaneously available in system state v , i.e., there will always be more cores or less cores available at any time. In that case, we set $A_k^{exact}(v) = +\infty$.

Note that to avoid computing all combinations of values in $\mathcal{F}(v)$ to compute $A_k^{exact}(v)$, one can use text-book solutions that solve the subset-sum problem. Namely, using dynamic programming [17] for the subset-sum problem, one can compute $A_p^*(v)$ with a complexity $O(s \cdot N)$ where s is the maximum sum to find, and N is the number of elements in the set \mathcal{F} . In our case, both s and the size of \mathcal{F} are upper-bounded by the number of cores m resulting to an $O(m^2)$ complexity.

4.1.3 Latest Start Time

The latest time LST_i^p at which job J_i may start to execute on p cores assuming that it is the next job that is dispatched by the scheduler depends on three factors:

- (i) The time $t_{avail}^p(v)$ at which at least $next_i(p)$ cores become available, since, if $m_i^{\max} > p$, the scheduler would then dispatch J_i on more than p cores (instead of p cores);
- (ii) The time $t_{wc}(v)$ at which another job than J_i certainly becomes eligible for execution, since J_i will not be dispatched *first* if it has not been dispatched before $t_{wc}(v)$;
- (iii) The time $t_{high}^p(v)$ at which a higher-priority job may become eligible, since to be dispatched before any other job, J_i must be dispatched before time $t_{high}^p(v)$.

We explain how to compute bounds on those three time instants.

First, according to Rule 5, if J_i starts to execute on p cores at time LST_i^p , then either p is the maximum number of cores on which J_i may execute, i.e., $p = m_i^{\max}$, or there are no more than p cores available at time LST_i^p . Since $A_{next_i(p)}^{\max}(v)$ denotes the time by which $next_i(p)$ cores will certainly become available, we have that

$$LST_i^p(v) \leq t_{avail}^p(v) \quad (3)$$

where

$$t_{avail}^p(v) = \begin{cases} A_{next_i(p)}^{\max}(v) - 1 & \text{if } p < m_i^{\max}, \\ +\infty & \text{otherwise.} \end{cases} \quad (4)$$

Second, if J_i is the first job dispatched by the scheduler until time LST_i^p , then according to Rules 3 and 8, there must be no other job that was eligible to be dispatched before LST_i^p . Since by Rule 2, a job J_j is eligible only if it is ready and there are at least m_j^{\min} cores available, we must have

$$LST_i^p(v) \leq t_{wc}(v) \quad (5)$$

with

$$t_{wc}(v) = \min_{J_j \notin \mathcal{S}(v)} \{\max\{r_j^{\max}, A_{m_j^{\min}}^{\max}(v)\}\}, \quad (6)$$

where r_j^{\max} is the latest time at which a job J_j that was not scheduled yet (i.e., $J_j \notin \mathcal{S}(v)$) may be released, and $A_{m_j^{\min}}^{\max}(v)$ is the latest time by which the minimum number of cores m_j^{\min} requested by J_j will be available to execute J_j .

Third, according to Rule 4, if job J_i is dispatched at time LST_i^p and it is the first job dispatched by the scheduler in system state v , then J_i must be the highest priority eligible job until time LST_i^p . That is,

$$LST_i^p(v) < t_{high}^p(v), \quad (7)$$

with

$$t_{high}^p(v) = \min_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}}^{\infty} \{t_h^p(J_i, J_j)\}, \quad (8)$$

where $\min_{x \in S}^{\infty} \{x\} = +\infty$ if $S = \emptyset$, otherwise, $\min_{x \in S}^{\infty} \{x\} = \min_{x \in S} \{x\}$, and

$$t_h^p(J_i, J_j) = \begin{cases} r_j^{\max} & \text{if } m_j^{\min} \leq p, \\ \max\{r_j^{\max}, A_{m_j^{\min}}^{\max}\} & \text{otherwise.} \end{cases} \quad (9)$$

► **Lemma 3.** *J_i will not be the first job dispatched in system state v or will not be dispatched on exactly p cores, if it did not start to execute before time $t_{high}^p(v)$ as defined by Equation (8).*

Proof. We prove that a not-yet-dispatched higher-priority job J_j (i.e., $J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}$) will be dispatched before J_i if J_i did not start executing before $t_h^p(J_i, J_j)$. It then directly follows that J_i will not be the first job dispatched on p cores if J_i did not start to execute before $t_{high}^p(v) = \min_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}}^{\infty} \{t_h^p(J_i, J_j)\}$, hence proving the lemma. We consider two cases:

Case 1. If $m_j^{\min} \leq p$, then the higher-priority job J_j requires fewer cores than the number of cores requested by job J_i . Thus, if job J_j is released when J_i become eligible, then according to Rule 2, J_j is also eligible, and because J_j has a higher priority than J_i , the scheduler will dispatch J_j instead of J_i (Rule 4). Therefore, J_i cannot be scheduled before J_j on p cores if it did not start to execute before r_j^{\max} . This proves that J_j will be dispatched before J_i if J_i did not start to execute before $t_h^p(J_i, J_j)$.

Case 2. If $m_j^{\min} > p$, then, according to Rule 2, the higher-priority job J_j becomes eligible when it is released and when m_j^{\min} cores are available. This happens at the latest at time $\max\{r_j^{\max}, A_{m_j^{\min}}^{\max}(v)\}$. Then, because J_j has a higher priority than J_i , the scheduler will dispatch J_j if J_i did not start to execute before $\max\{r_j^{\max}, A_{m_j^{\min}}^{\max}(v)\}$ (Rule 4). Thus, we proved that J_j will be dispatched before J_i if J_i did not start to execute before $t_h^p(J_i, J_j)$. ◀

► **Corollary 4.** *Job J_i cannot be dispatched on p cores and be the first job dispatched in state v later than $LST_i^p(v) = \min\{t_{avail}^p(v), t_{wc}(v), t_{high}(v) - 1\}$.*

Proof. It directly follows from the combination of Equations (3), (5) and (7). ◀

4.1.4 Dispatch Condition

A job J_i may be dispatched on p cores (with $p \in \mathcal{P}_i$) and may be the first job dispatched by the scheduler in a system state v only if the earliest time at which it may be dispatched on p cores is no later than the latest time at which it may be the first job to be dispatched. That is, it must respect the following inequality:

$$EST_i^p(v) \leq LST_i^p(v) \quad (10)$$

► **Theorem 5.** *A job J_i may be dispatched on p cores and be the first job dispatched by the scheduler in system state v only if $EST_i^p(v) < \infty$ and Inequality (10) is respected.*

Proof. It is obvious that the earliest start time $EST_i^p(v)$ of J_i must be smaller than ∞ to ensure that J_i may start to execute in system state v . Hence, we focus on Inequality (10). By contradiction, assume that (i) a job J_i is the first job dispatched by the scheduler in system state v , that (ii) J_i is assigned p core by the scheduler and that (iii) J_i does not respect Inequality (10). Let t_s be the time at which J_i starts executing. By Lemma 2, we have that $t_s \geq EST_i^p(v)$. Thus, by assumption (iii) and the definition of $LST_i^p(v)$ given in Corollary 4, we have $t_s > t_{avail}^p$ or $t_s > t_{wc}$ or $t_s \geq t_{high}$. We analyse each case independently.

- $t_s > t_{avail}^p$. Since by Equation (4), $t_{avail}^p \geq A_{next_i(p)}^{\max}(v) - 1$ and because $t_s > t_{avail}^p$, we have $t_s \geq A_{next_i(p)}^{\max}(v)$. Therefore, at least $next_i(p)$ cores are available at time t_s . Thus, by Rule 5, J_i is dispatched on at least $next_i(p)$ cores. This contradicts the assumption (ii) that J_i is dispatched on p cores.
- $t_s > t_{wc}$. By definition of t_{wc} , a job certainly became eligible to be dispatched by time t_{wc} . Therefore, a job must have been dispatched by the scheduler at or before t_{wc} . This contradicts the assumption (i) that J_i is the first job dispatched by the scheduler and J_i is dispatched at time t_s .
- $t_s \geq t_{high}^p$. By Lemma 3, J_i is not the highest-priority eligible job at time t_s . Thus, by Rule 4, it is not the first job dispatched by the scheduler, hence contradicting the assumption (i).

We thus reached a contradiction in all cases, which proves the claim. ◀

4.2 Job Finish Times

The earliest time at which a job J_i may complete its execution when dispatched on p cores is when it starts at the earliest (i.e., at $EST_i^p(v)$) and executes for its best-case execution time on p cores (i.e., for $C_i^{\min}(p)$). That is,

$$EFT_i^p(v) = EST_i^p(v) + C_i^{\min}(p) \quad (11)$$

Similarly, the latest time at which a job J_i may complete its execution when it is the next job dispatched and it is dispatched on p cores is when it starts as late as possible (i.e., at $LST_i^p(v)$) and it runs for its WCET on p cores (i.e., for $C_i^{\max}(p)$). That is,

$$LFT_i^p(v) = LST_i^p(v) + C_i^{\max}(p) \quad (12)$$

4.3 Building New System States

If job J_i satisfies the dispatch condition for p cores in state v , then there are execution scenarios in which the scheduler may dispatch J_i on p cores in system state v . For each such scenario, we build a new node v' representing the system state resulting from scheduling J_i on p cores. Apart from adding J_i to the set of scheduled jobs $\mathcal{S}(v')$, there are two data structures that must be updated. The set of availability intervals, and the set of earliest simultaneous core releases \mathcal{F} . We discuss these in the following sub-sections.

4.3.1 New Set of Earliest Simultaneous Core Releases \mathcal{F}

Our discussion has two parts. We first cover the case where the number of cores p assigned to J_i is smaller than its maximum parallelism m_i^{\max} , and then cover the case where $p = m_i^{\max}$.

4.3.1.1 $p < m_i^{\max}$

If $p < m_i^{\max}$, then exactly p cores must be available when J_i starts to execute (Rule 5). Yet, any combination of simultaneously released cores that sum to p and are possibly released between the earliest and latest start time of J_i may be used to execute J_i . Because there may be more than one such combination, we first identify every subset $\mathcal{F}_k^{\leq p}$ of elements in $\mathcal{F}(v)$ such that $\sum_{F_\ell \in \mathcal{F}_k^{\leq p}} M_\ell(v) = p$ and $\forall F_\ell \in \mathcal{F}_k^{\leq p}, f_\ell(v) \leq LST_i(v)$. Then, for each subset $\mathcal{F}_k^{\leq p} \subseteq \mathcal{F}(v)$ that meets those conditions, we create a new node v'_k in the graph that represents the system state resulting from dispatching J_i on the specific p cores contained in $\mathcal{F}_k^{\leq p}$. The new set of earliest simultaneous core releases $\mathcal{F}(v'_k)$ in the new state v'_k is then built according to Lemma 6.

► **Lemma 6.** *Let node v'_k result from executing J_i on the p cores in $\mathcal{F}_k^{\leq p}$, then the set of earliest simultaneous core releases is $\mathcal{F}(v'_k) = \{\langle EFT_i^p(v), p \rangle\} \cup \{\mathcal{F}(v) \setminus \mathcal{F}_k^{\leq p}\}$.*

Proof. Since v'_k considers a system state that results from dispatching job J_i on p cores, p cores will be released simultaneously by J_i when it finishes its execution. This happens no earlier than the earliest finish time $EFT_i^p(v)$ of J_i . Therefore, $\mathcal{F}(v'_k) \supseteq \{\langle EFT_i^p(v), p \rangle\}$.

Furthermore, because by assumption J_i executes on the cores in $\mathcal{F}_k^{\leq p}$, the time at which the cores in $\mathcal{F}(v) \setminus \mathcal{F}_k^{\leq p}$ are released is not impacted by the execution of J_i . Thus, $\mathcal{F}(v'_k) \supseteq \{\mathcal{F}(v) \setminus \mathcal{F}_k^{\leq p}\}$. ◀

4.3.1.2 $p = m_i^{\max}$

When the number of cores p assigned to J_i is equal to its maximum parallelism m_i^{\max} , there must be at least p but also potentially more than p cores available when J_i starts to execute. Thus, differently from the case covered above, we identify every subset $\mathcal{F}_k^{\geq p}$ of $\mathcal{F}(v)$ whose elements sum up to *at least* p . That is, $\sum_{F_\ell \in \mathcal{F}_k^{\geq p}} M_\ell(v) \geq p$ and $\forall F_\ell \in \mathcal{F}_k^{\geq p}, f_\ell(v) \leq LST_i(v)$. As before, for each subset $\mathcal{F}_k^{\geq p}$, we create a new node v'_k whose set of earliest simultaneous core releases $\mathcal{F}(v'_k)$ is computed according to Lemmas 7 and 8.

► **Lemma 7.** *If all the cores in $\mathcal{F}_k^{\geq p}$ are released when J_i starts to execute, then J_i starts no earlier than $t_k = \max_{F_\ell \in \mathcal{F}_k^{\geq p}} \{f_\ell\}$.*

Proof. By definition of F_ℓ , the M_ℓ cores modeled by F_ℓ are all released at the earliest at time f_ℓ . Thus, all the cores in $\mathcal{F}_k^{\geq p}$ are available no earlier than $\max_{F_\ell \in \mathcal{F}_k^{\geq p}} \{f_\ell\}$. Since J_i starts when all cores in $\mathcal{F}_k^{\geq p}$ are available, this proves the claim. ◀

► **Lemma 8.** *Let node v'_k result from executing J_i on p of the cores in $\mathcal{F}_k^{\geq p}$, then the set of earliest simultaneous core releases is $\mathcal{F}(v'_k) = \{\langle EFT_i^p(v), p \rangle\} \cup \{\langle t_k, (s-p) \rangle\} \cup \{\mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p}\}$ where s is the number of cores in $\mathcal{F}_k^{\geq p}$, i.e., $s = \sum_{F_\ell \in \mathcal{F}_k^{\geq p}} M_\ell(v)$.*

Proof. Since v'_k considers a system state that results from dispatching job J_i on p cores, p cores will be released simultaneously by J_i when it finishes its execution. This happens no earlier than the earliest finish time $EFT_i^p(v)$ of J_i . Therefore, $\mathcal{F}(v'_k) \supseteq \{\langle EFT_i^p(v), p \rangle\}$.

■ **Algorithm 2** Build all system states resulting from dispatching J_i on p cores in v .

```

1 for  $\forall \mathcal{F}_k^p \subseteq \mathcal{F}(v)$  s.t. conditions of Sec. 4.3.1 are respected do
2   Add a node  $v'_k$  to the sched.-abstraction graph  $\mathcal{G}$ ;
3    $\mathcal{S}(v'_k) \leftarrow \mathcal{S}(v) \cup \{J_i\}$ ;
4   Compute  $PA$  and  $CA$  according to Lemmas 9 and 11;
5   Sort  $PA$  and  $CA$  in non-decreasing order ;
6    $\forall x, 1 \leq x \leq m, A_k(v'_k) = [PA_x, CA_x]$ ;
7   Compute  $\mathcal{F}(v'_k)$  according to Lemmas 6 and 8;
8   Connect  $v$  to  $v'_k$  with an edge;

```

Furthermore, by assumption, all cores in $\mathcal{F}_k^{\geq p}$ are free when J_i starts to execute. Therefore, all $(s - p)$ cores in $\mathcal{F}_k^{\geq p}$ on which J_i does not execute are free from J_i 's start time onward. By Lemma 7, J_i starts no earlier than t_k . Hence, $\mathcal{F}(v'_k) \supseteq \{ \langle t_k, (s - p) \rangle \}$.

Finally, because J_i executes on the cores in $\mathcal{F}_k^{\geq p}$, the time at which the cores in $\mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p}$ are released is not impacted by the execution of J_i . Thus, $\mathcal{F}(v'_k) \supseteq \{ \mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p} \}$. ◀

4.3.2 New Availability Intervals

To construct the availability intervals $A_x(v'_k)$ ($1 \leq x \leq m$) of a system state v'_k reachable from v , we build the set PA of all instants at which each core may *potentially* be available, and the set CA of the latest possible times at which each core will certainly become available after dispatching J_i on p cores in $\mathcal{F}_k^{\leq p}$ or $\mathcal{F}_k^{\geq p}$ (depending on whether $p < m_i^{\max}$ or $p = m_i^{\max}$ as discussed above). We do so using Lemmas 9 and 11.

► **Lemma 9.** *A set of lower bounds on the time instants at which each core may potentially become available to execute new workload in v'_k is given by*

$$PA = \left\{ p \times \{EFT_i^p(v)\} \right\} \cup \left\{ \max(A_x^{\min}(v), t_k) \mid p < x \leq m \right\}$$

where $p \times \{EFT_i^p(v)\}$ means that $EFT_i^p(v)$ appears p times in the set.

Proof. First, since v'_k considers a system state that results from dispatching job J_i on p cores, at least p cores will become available no earlier than the earliest finish time $EFT_i^p(v)$ of J_i . Therefore, PA must contain p times $EFT_i^p(v)$.

Second, by Rule 8, J_i will always be dispatched on the p first cores that become available. Therefore, the earliest time at which the $(m - p)$ remaining cores may become available is the earliest time at which the $(m - p)$ latest cores may become available before dispatching J_i . By definition of the availability intervals, those times are $\{A_x^{\min}(v) \mid p < x \leq m\}$.

Finally, since job J_i is the first job dispatched by the scheduler in state v , and because by Lemma 7, t_k is the earliest time at which J_i is dispatched, cores can start to execute new workload no earlier than t_k in v'_k . Combining the three facts above prove the claim. ◀

► **Corollary 10.** *A lower bound on the time at which x cores are potentially available to execute new workload in v'_k (i.e., $A_x^{\min}(v'_k)$) is given by the x^{th} element in the non-decreasingly ordered set PA .*

Proof. Since PA contains a lower bound on the availability time of every core in state v'_k , the x^{th} element in the ordered set is a lower bound on the availability time of x cores. ◀

► **Lemma 11.** *A set of upper bounds on the time instants at which each core will certainly become available to execute new workload in v'_k is given by*

$$CA = \left\{ p \times \{LFT_i^p(v)\} \right\} \cup \left\{ \max\{A_x^{\max}(v), t_k\} \mid p < x \leq m \right\}.$$

Proof. Since v'_k represents a system state resulting from dispatching job J_i on p cores, there must be at least p cores that will become available to execute new workload no later than the latest finish time of J_i . That is, there must be p values no smaller than $LFT_i^p(v)$ in CA , i.e., $CA \supseteq \left\{ p \times \{LFT_i^p(v)\} \right\}$.

Furthermore, all $(m - p)$ cores that do not execute J_i will be freed no later than the certain availability time of the $(m - p)$ latest cores that become available in the initial system state v (i.e., the system state before dispatching J_i). Those times are given by $\left\{ A_x^{\max}(v) \mid p < x \leq m \right\}$.

Finally, since job J_i is the first job dispatched by the scheduler in v , and because t_k is the earliest time at which J_i is dispatched (Lemma 7), cores can start to execute new workload no earlier than t_k in v'_k . Combining all the above, we prove the lemma. ◀

► **Corollary 12.** *An upper bound on the time at which x cores are certainly available to execute new workload in v'_k (i.e., $A_x^{\max}(v'_k)$) is given by the x^{th} element in the non-decreasingly ordered set CA .*

Proof. Same proof as Corollary 10, replacing PA with CA . ◀

The complete procedure to build the system states resulting from dispatching J_i on p cores in state v is summarized in Algorithm 2.

5 Merge Phase

The merge phase aims at merging a newly created node v_k with a previously existing node v_q (and create a combined state v_z) when they have the same set of scheduled jobs and their core-availability intervals intersect:

► **Rule 9.** *If v_k and v_q are two nodes such that $\mathcal{S}(v_k) = \mathcal{S}(v_q)$ and $\forall x, 1 \leq x \leq m, A_x(v_k) \cap A_x(v_q) \neq \emptyset$, then v_k and v_q are merged into a single state v_z .*

The availability intervals of the merged state v_z are then computed so that they enclose the availability intervals of both states v_k and v_q . That is, $\forall x \mid 1 \leq x \leq m$:

$$A_x(v_z) = \left[\min\{A_x^{\min}(v_k), A_x^{\min}(v_q)\}, \max\{A_x^{\max}(v_k), A_x^{\max}(v_q)\} \right]. \quad (13)$$

This way, all possible combinations of instants at which cores become available in either state v_k or v_q is also possible in v_z .

Additionally, the set of earliest simultaneous core releases $\mathcal{F}(v_z)$ of the merged state is computed using Algorithm 3. In essence, for both initial states v_k and v_q , Algorithm 3 sorts the groups of cores that are simultaneously released in a non-decreasing order with respect to the time at which they are released. It then breaks the groups of simultaneously released cores in smaller ones so that the size of the groups match in both states (lines 3–10), i.e., after the transformation we have $|\mathcal{F}'(v_k)| = |\mathcal{F}'(v_q)|$ and $\forall x, 1 \leq x \leq |\mathcal{F}'(v_k)|, M'_x(v_k) = M'_x(v_q)$. It then keeps the groups of cores that are released the earliest and assign them to $\mathcal{F}(v_z)$ (lines 10–14), i.e., $|\mathcal{F}(v_z)| = |\mathcal{F}'(v_k)| = |\mathcal{F}'(v_q)|$ and $\forall x, 1 \leq x \leq |\mathcal{F}'(v_k)|, M_x(v_z) = M'_x(v_k) = M'_x(v_q)$ and $f_x(v_z) = \min\{f'_x(v_k), f'_x(v_q)\}$.

■ **Algorithm 3** Merge of $\mathcal{F}(v_k)$ and $\mathcal{F}(v_q)$ into $\mathcal{F}(v_z)$.

```

input :  $\mathcal{F}(v_k)$  and  $\mathcal{F}(v_q)$ 
output :  $\mathcal{F}(v_z)$ 
1  $\mathcal{F}'(v_k) = \mathcal{F}'(v_q) = \emptyset$ ;
2 while  $\mathcal{F}(v_k) \neq \emptyset \wedge \mathcal{F}(v_q) \neq \emptyset$  do
3   Extract the pair  $\langle f_K, M_K \rangle$  such that  $f_K$  is the minimum value in  $\mathcal{F}(v_k)$  and, in case of
   tie,  $M_K$  is the minimum among the tying values. For  $\mathcal{F}(v_q)$  extract  $\langle f_Q, M_Q \rangle$  using the
   same rule;
4    $M_{new} \leftarrow \min \{M_K, M_Q\}$ ;
5   Add  $\langle f_K, M_{new} \rangle$  to  $\mathcal{F}'(v_k)$ ;
6   Add  $\langle f_Q, M_{new} \rangle$  to  $\mathcal{F}'(v_q)$ ;
7    $M_K \leftarrow M_K - M_{new}$ ;
8    $M_Q \leftarrow M_Q - M_{new}$ ;
9   Add  $\langle f_K, M_K \rangle$  to  $\mathcal{F}(v_k)$  if  $M_K > 0$  ;
10  Add  $\langle f_Q, M_Q \rangle$  to  $\mathcal{F}(v_q)$  if  $M_Q > 0$  ;
11 forall  $1 \leq x \leq |\mathcal{F}'(v_k)|$  do
12    $f_x(v_z) = \min \{f'_x(v_k), f'_x(v_q)\}$ ;
13    $M_x(v_z) = M'_x(v_k)$ ;
14   Add  $\langle f_x(v_z), M_x(v_z) \rangle$  to  $\mathcal{F}(v_z)$ ;
15 return  $\mathcal{F}(v_z)$ ;
```

We now prove that all simultaneous core release patterns that are possible in one of the two initial states v_k or v_q is also possible in the new merged state v_z .

► **Lemma 13.** *If exactly p cores may be available at time t in either v_k or v_q , then exactly p cores may be available at time t in v_z .*

Proof. Assume that v refers to either v_k or v_q . Each group of cores $F_\ell(v) \in \mathcal{F}(v)$ is subdivided in one or several smaller groups of cores in $\mathcal{F}(v_z)$ (lines 3–10 in Algorithm 3), that is, $\exists \mathcal{F}' \subseteq \mathcal{F}(v_z)$, $\sum_{F_x(v_z) \in \mathcal{F}'} M_x(v_z) = M_\ell(v)$. Furthermore, each group of cores in the subset \mathcal{F}' has an *earliest* release time that is earlier than or at the same time as that of F_ℓ (lines 10–14), i.e., $\forall F_x(v_z) \in \mathcal{F}'$, $f_x(v_z) \leq f_\ell(v)$. Since for every group of cores that can be simultaneously released at a given time t in v there is a set \mathcal{F}' in v_z composed of the same number of cores, each with an earliest release time no later than t , it then holds that the cores in \mathcal{F}' can also be simultaneously released at t . This proves the lemma. ◀

6 Proof of Correctness

Now that the complete algorithm for building the schedule-abstraction graph has been presented, we prove that the analysis covers all possible execution scenarios and hence returns safe bounds on the BCRT and WCRT of each job in the analyzed job set \mathcal{J} .

► **Theorem 14.** *For any possible execution scenario such that J_i executes on p cores and finishes at time t , there is a path $\langle v_1, \dots, v_k \rangle$ in the schedule-abstraction graph such that J_i passes the dispatch condition on p cores in v_k and $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$.*

Proof. Assume that the availability intervals and the set of earliest simultaneous core releases $\mathcal{F}(v_k)$ of state v_k safely model the actual availability times and simultaneous releases of the m cores resulting from the sequence of scheduling decisions encoded in the path $\langle v_1, \dots, v_k \rangle$.

We prove that $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$, that J_i passes the dispatch condition in v_k and that each state v'_k created by Algorithm 2 because of executing J_i on p cores in v_k , correctly models the actual availability times and simultaneous releases of the cores after executing J_i on p cores.

Under the inductive assumption stated above, Lemma 2 and Corollary 4 prove that $EST_i^p(v_k)$ and $LST_i^p(v_k)$ are safe lower- and upper-bounds on the start time of J_i on p cores in v_k , respectively. Furthermore, since gang jobs are non-preemptive, Equations (11) and (12) are safe lower- and upper-bounds on t (i.e., $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$). Moreover, since $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$, it must hold that $EFT_i^p(v_k) \leq LFT_i^p(v_k)$, and thus the condition of Equation (10) is respected. Then, Lemmas 6 and 8 and Corollaries 10 and 12 prove that the simultaneous releases of the cores and their availability is correctly modeled in each newly created state v'_k resulting from scheduling J_i on p cores. Therefore, the inductive assumption is respected for v'_k . Also, according to Lemma 13, potentially merging v'_k with another node in Algorithm 1 maintains the validity of the inductive assumption.

Finally, since all cores are assumed to be free in the initial system state, the inductive assumption (i.e., correct availability intervals and simultaneous core releases) obviously holds for v_1 and thus follows by induction on all the states created by Algorithm 1. ◀

7 Empirical Evaluation

We performed experiments to: (i) evaluate whether the proposed analysis improves schedulability in comparison to the state of the art, (ii) understand the influence of m^{\min} and m^{\max} on schedulability of moldable gang tasks, and (iii) evaluate the runtime of our analysis.

The experiments were conducted by applying Algorithm 1 to the analysis of rigid and moldable gang tasks under a non-preemptive JLFP policy (some experiments use non-preemptive G-EDF and others G-RM as reported in Table 1). We compared our results with the test by Dong and Liu [10] as it is the only existing test for non-preemptive gang tasks. It is worth noting that the test of Dong and Liu considers sporadic tasks while we have performed the analysis on periodic tasks. We, however, decided to keep this comparison since it is currently the only available test that can be applied on periodic gang tasks.

We implemented Algorithm 1 in C++ and performed the analysis on a cluster using AMD Ryzen Threadripper 2920X 12-Core and Intel Core i9-9900K processors. All machines are equipped with 64 GiB of RAM. Roughly, 60% of the experiments ran on the AMD and 40% on the Intel machines. We report the *CPU time* as the runtime of the analysis.

7.1 Experiments on Synthetic Task Sets

We generate periodic task sets using the same established method used in prior studies [15,19,20]. We randomly generated n utilization values with a total sum of $m \times U$, where U is the system utilization and m is the number of cores. This was carried out using Stafford's RandFixSum algorithm [29] (where we ensure that the utilization U_i of each task is in the interval $[0.001, m_i^{\min}]$). To avoid cases where the hyperperiod is impractically large due to incompatible task periods, we choose the period values with a log-uniform distribution in the interval $[10000, 100000]$ with a granularity of 5000 (as in [20]). Additionally, we discard every task set that contains more than 100,000 jobs in its hyperperiod. To allow comparison with the state-of-the-art, release jitter is set to 0. Note that this favourably impacts our analysis runtime since the schedule abstraction graph branches less often for such setting.

■ **Table 1** Specification of the experiments performed.

Experiment	m	n	m_i^{\min}	m_i^{\max}	$\max U_i$	Policy
a-seq-random	8	20	1	$\{1, 2, 3, \dots, m\}$	1	NP G-RM
a-seq-divisor				$\{1, 2, 4, 8\}$		
a-gang-random			$\{1, 2, 3, \dots, m\}$	$\{1, 2, 3, \dots, m\}$	$m \times U$	
a-gang-divisor			$\{1, 2, 4, 8\}$	$\{1, 2, 4, 8\}$		
b	8	20	$m^{\min} = m^{\max}$	$\{1, 2, 4, 6, 8\}$	$m \times U$	NP G-EDF
c	8	8–24	1	$\{1, 2, \dots, 8\}$	1	NP G-RM
d	4	10	1	$\{1, 2, 3, 4\}$	1	NP G-RM
e	8	20	1	$\{1, 2, \dots, 8\}$	1	NP G-RM
f	16	32	1	$\{1, 2, \dots, 16\}$	1	NP G-RM

To evaluate the impact of m_i^{\min} and m_i^{\max} on schedulability, we assign different values for m_i^{\min} and m_i^{\max} for each experiment depending on its purpose, as detailed in Table 1. The $\max U_i$ value specifies the maximum utilization that a single task may have in the specified experiment. The values of m_i^{\min} and m_i^{\max} in experiments *a-seq-random* and *a-gang-random* are selected randomly with a uniform distribution from the set $\{1, 2, \dots, m\}$. In experiments *a-seq-divisor* and *a-gang-divisor*, these values are selected randomly from the set $\{1, 2, 4, 8\}$ which is composed of the divisors of the number of cores $m = 8$, we always ensure that $m_i^{\min} < m_i^{\max}$ when picking random values. Experiment (b) assumes a rigid gang model, thus, $m_i^{\min} = m_i^{\max}$ for all tasks. Finally, in experiments (c)–(f) we show results for the generation methods seq-random, gang-random and when all tasks share the same m_i^{\min} and m_i^{\max} values. In the latter case, $m_i^{\min} = 1$ and m_i^{\max} varies from 1 to 16.

In our experiments, jobs of a task τ_i can execute on any number of cores within $[m_i^{\min}, m_i^{\max}]$. Their BCET and WCET on p cores (with $p \in [m_i^{\min}, m_i^{\max}]$) are set to $\lfloor \frac{U_i \times T_i}{2 \times p} \rfloor$ and $\lfloor \frac{U_i \times T_i}{p} \rfloor$, respectively. Hence, execution time decreases with an increasing number of cores, and BCET is half the WCET.

Figure 2 shows the results of each experiment. For each data point in the plots, we generate 450 random task sets and report the *schedulability ratio* (i.e., the percentage of task sets deemed schedulable by the analysis). Additionally, we report the runtime of the schedulability analysis for each task set tested in experiments (d), (e), and (f) as a function of the number of jobs in their hyperperiod.

7.2 Schedulability Results

Impact of system utilization on rigid gang tasks. As shown in Figure 2b, The SAG analysis clearly outperforms the utilization-based test of Dong and Liu for any value of m^{\max} . For instance, Dong and Liu’s test does not detect any schedulable task set at $U=40\%$ while our analysis confirms that between 95 and 100% (depending on the maximum task parallelism m_i^{\max}) of these task sets are in fact schedulable. More importantly, our analysis identifies **4.9 times** more schedulable task sets than [10] (over all m_i^{\max} values from 1 to 8). This value is computed by taking the ratio between the surface below all schedulability curves obtained with our analysis and the surface below all schedulability curves of Dong and Liu’s test. Another observation is that rigid gang tasks with $m_i^{\max} = 8$ have the highest schedulability in comparison to rigid gangs with $m_i^{\max} < 8$. When $m_i^{\max} = 8$, tasks can only start when all 8 cores are available, hence, the schedulability problem boils down to a schedulability analysis for a uniprocessor platform, where the SAG analysis is highly accurate (see [18]).

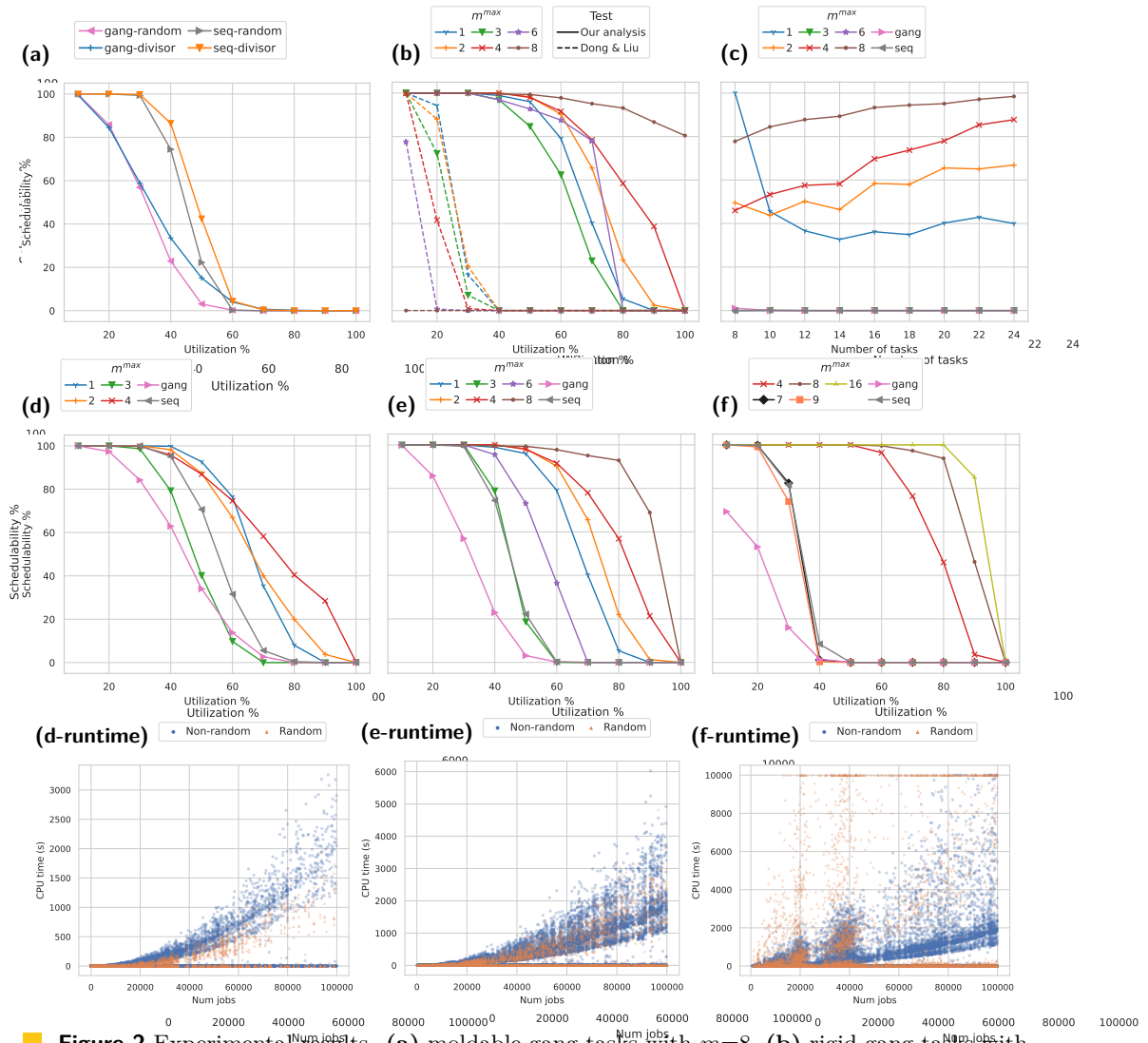


Figure 2 Experimental results. (a) moldable gang tasks with $m=8$, (b) rigid gang tasks with $m=8$, dashed lines are the Dong and Liu’s test [10] and the continuous lines are ours, (c) moldable gang tasks with $m=8$, $U=0.7$ and $m^{\min}=1$, (d) moldable gang tasks with $m=4$ and $m^{\min}=1$, (e) moldable gang tasks with $m=8$ and $m^{\min}=1$, (f) moldable gang tasks with $m=16$ and $m^{\min}=1$.

Impact of utilization and task parallelism on moldable gang tasks. As shown in Figure 2a (see explanation of the curves in experimental setup), task sets with m_i^{\min} set to 1 for all tasks (i.e., curves *a-seq-random* and *a-seq-divisor*) have a higher-schedulability ratio because they can be dispatched as soon as one or more cores become free. If $m_i^{\min} \geq 1$, however, tasks may experience longer blocking (waiting for their minimum number of cores to be freed) and frequent priority inversions with lower-priority tasks “stealing” available cores from higher-priority ones as it can be seen in *a-gang-random* and *a-gang-divisor* curves.

Furthermore, we compared the difference between choosing m^{\max} to be a random value from 1 to m (the number of cores) and be a random value that is a divisor of m , i.e., $m^{\max} \in \{1, 2, 4, 8\}$. In the latter case (see curves *a-seq-divisor* and *a-gang-divisor*), the schedulability ratio slightly improves in comparison to the former. This slight improvement is caused by having a few less scenarios where cores are left idle with pending workload. However, the impact remains rather small.

Impact of the number of tasks and task parallelism on moldable gang tasks. Figure 2c shows the effect of the number of *moldable* gang tasks when $U = 70\%$. When $m^{\max} = 1$, the results are identical to non-preemptive global scheduling since each task can claim only one core. Also, when $m^{\max} = 8$, the scheduler described in Section 2.2 will execute all jobs on $p = 8$ cores, which is equivalent to single-core scheduling. Thus, as the number of tasks increases, the execution time of the tasks decreases and the results become closer to those of single-core preemptive scheduling. That is why the schedulability ratio increases.

Similarly, when m_i^{\max} is set to 2 or 4 for all tasks, then the scheduler of Section 2.2 behaves identically to a non-preemptive global scheduler on 4 and 2 cores respectively, hence explaining the typical tendency witnessed for such systems. From this experiment, we can conclude that a larger maximum task parallelism is beneficial for schedulability. When m_i^{\max} is set to 3, 6 or is randomly chosen for each task, many jobs cannot be dispatched with their maximum number of cores due to m_i^{\max} not being a divisor of the number of cores. Therefore, consistently with what is seen in Figure (2e) discussed later, the schedulability ratio falls to 0.

Impact of the number of cores and task parallelism on moldable gang tasks. Figure 2d shows that in a system with four processors, configuring m^{\max} to 3 causes a significant lower schedulability ratio than with other values. With eight cores (Figure 2e), setting m^{\max} to 3 or 6 also yields lower schedulability ratios (the same is true for $m^{\max} = 5$ and $m^{\max} = 7$, even though we do not show the results here to avoid clutter). The same effect is visible when m^{\max} is chosen randomly for each task. This shows the positive impact of using a same m^{\max} value that is a divisor of the number of cores for all tasks. When it is the case, all the jobs will always be scheduled with $p = m^{\max}$ because, as soon as a job finishes, it frees exactly the same number of cores as the next job needs to execute with m^{\max} cores. This eliminates the problem of some cores being available but not used due to all pending jobs requesting more cores than available. When m^{\max} is not a divisor of m , some jobs may be executing with m^{\max} cores while others may execute with smaller values of p , this causes an imbalance in execution times that leads to more frequent deadline misses.

Runtime of the analysis. Figure 2 shows the runtime of the SAG analysis when all tasks share the same m^{\max} value (blue) and when m^{\max} is assigned randomly (orange) for the experiments (d)–(f). It shows that the runtime is well below 1000 s in a vast majority of experiments, and the worst-case runtime is below 100 minutes for task sets with 20 tasks scheduled on platforms with 8 cores (see Figure 2e-runtime). For 16 cores executing 32 tasks, we start to see experiments timing out when the number of jobs increases and m^{\max} is assigned randomly (note that those are reported as being deemed unschedulable in Figure (2f)).

Comparison with existing tests for global scheduling of sequential tasks. For the case where the schedulability problem can be reduced to an equivalent global JLFP scheduling problem of non-preemptive sequential tasks, we compared our results with those of the SAG-based test of [19], i.e., the most accurate analysis we are aware of for such systems. All task sets that were detected as schedulable by the test of [19] were also deemed schedulable by our analysis. This suggests that our new analysis reduces to that of [19] in the special case where all tasks share a same m^{\max} value that divides the number of cores m . A comparison against other sufficient tests for global JLFP scheduling can be found in [19].

8 Summary and Conclusion

We proposed a new response-time analysis for rigid and moldable gang tasks scheduled under a non-preemptive JLFP scheduling policy. As far as we know, our work is the first effort to provide sound worst-case (and best-case) response time bounds for such systems. Our analysis is based on the notion of schedule abstraction and efficiently explores all possible sequences of scheduling decisions that may happen during the runtime of the system.

Our experiments show that for periodic rigid gang tasks, our analysis is able to identify 4.9 times more schedulable task sets than the state of the art analysis. Moreover, our experiments revealed the importance of choosing proper values for the minimum and maximum parallelism assigned to moldable gang tasks. We observed that assigning the same value m_i^{\max} to all tasks yields the best performance, specially when m_i^{\max} is a divisor of the number of cores.

We plan on extending our analysis to add support for precedence constraints between moldable gang jobs, and on improving the runtime and memory consumption of our analysis by extending the very promising partial order reduction techniques presented in [24,25]. Among other steps, this will require to develop a fast sufficient schedulability test for non-preemptive moldable gang tasks.

References

- 1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–9, 2020.
- 2 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.
- 3 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.
- 4 Vandy Bertin, Pierre Courbin, and Joël Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, pages 9–12, 2011.
- 5 A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.
- 6 Jacek Blazewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, c-35(5):389–393, 1986.
- 7 Felipe Cerqueira, Geoffrey Nelissen, and Björn B Brandenburg. On strong and weak sustainability, with an application to self-suspending real-time tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 26–1, 2018.
- 8 Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.
- 9 Zheng Dong and Cong Liu. Analysis techniques for supporting hard real-time sporadic gang task systems. *Real-Time Systems*, 55(3):641–666, 2019.
- 10 Zheng Dong and Cong Liu. Work-in-progress: Non-preemptive scheduling of sporadic gang tasks on multiprocessors. In *Work-in-Progress of IEEE Real-Time Systems Symposium (WiP-RTSS)*, pages 512–515. IEEE, 2019.
- 11 Dror G. Feitelson. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 89–110, 1996.
- 12 Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

12:22 Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks

- 13 Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 189–196, 2010.
- 14 Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Systems*, 52(6):808–832, 2016.
- 15 Joël Goossens and Pascal Richard. Optimal scheduling of periodic gang tasks. *Leibniz transactions on embedded systems*, 3(1):4:1–4:18, 2016.
- 16 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 459–468, 2009.
- 17 Konstantinos Koiliaris and Chao Xu. Faster Pseudopolynomial Time Algorithms for Subset Sum. *ACM Transactions on Algorithms*, 15(3):1062–1072, 2019.
- 18 Mitra Nasri and Björn B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2017.
- 19 Mitra Nasri, Nelissen Geoffrey, and Björn B. Brandenburg. Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 21:1–21:23, 2019.
- 20 Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106, pages 9:1–9:23, 2018.
- 21 Saranya Natarajan, Mitra Nasri, David Broman, Björn B. Brandenburg, and Geoffrey Nelissen. From code to weakly hard constraints: A pragmatic end-to-end toolchain for timed C. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 167–180, 2019.
- 22 Suhail Nogh, Geoffrey Nelissen, Mitra Nasri, and Björn B. Brandenburg. Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 115–127, 2020.
- 23 John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.
- 24 Sayra Ranjha, Mitra Nasri, and Geoffrey Nelissen. Work-in-progress: Partial-order reduction in reachability-based response-time analyses. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 544–547, 2021.
- 25 Sayra Ranjha, Geoffrey Nelissen, and Mitra Nasri. Partial-order reduction for schedule-abstraction-based response-time analyses of non-preemptive tasks. In *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 121–132, 2022.
- 26 Pascal Richard, Joël Goossens, and Shinpei Kato. Comments on “Gang EDF Schedulability Analysis”, 2017. [arXiv:1705.05798](https://arxiv.org/abs/1705.05798).
- 27 Maria A. Serrano, Alessandra Melani, Sebastian Kehr, Marko Bertogna, and Eduardo Quiñones. An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling. In *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, pages 193–202, 2017.
- 28 Srinidhi Srinivasan, Geoffrey Nelissen, and Reinder J. Bril. Work-in-progress: Analysis of tsn time-aware shapers using schedule abstraction graphs. In *Real-Time Systems Symposium (RTSS)*, pages 508–511, 2021.
- 29 Roger Stafford. Random vectors with fixed sum. Technical report, University of Oxford, 2006. URL: <http://www.mathworks.com/matlabcentral/fileexchange/9700>.
- 30 Saud Wasly and Rodolfo Pellizzoni. Bundled scheduling of parallel real-time tasks. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 130–142, 2019.
- 31 Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 1222–1227, 2019.
- 32 Yanyong Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, 2003.

Response-Time Analysis for Self-Suspending Tasks Under EDF Scheduling

Federico Aromolo

Scuola Superiore Sant'Anna, Pisa, Italy

Alessandro Biondi

Scuola Superiore Sant'Anna, Pisa, Italy

Geoffrey Nelissen

Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract

The self-suspending task model proved to be particularly effective in capturing the timing behavior of real-time systems characterized by complex execution patterns, such as computation offloading to hardware accelerators, inter-core synchronization by means of multiprocessor locking protocols, and highly parallel computation. Most of the existing results for the timing analysis of self-suspending tasks do not support the widely adopted Earliest Deadline First (EDF) scheduling algorithm, being instead primarily focused on fixed-priority scheduling. This paper presents a response-time analysis for constrained-deadline self-suspending tasks scheduled under EDF on a uniprocessor system. The proposed analysis is based on a model transformation from self-suspending sporadic tasks to sporadic tasks with jitter, which can then be analyzed using a state-of-the-art analysis method for EDF scheduling. Experimental results are presented to compare the performance of the proposed technique in terms of schedulability ratio with that of the pessimistic suspension-oblivious approach and with a less general technique for task sets with implicit deadlines.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases Real-Time Systems, Schedulability Analysis, Self-Suspending Tasks, EDF Scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.13

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.8.1.5>

1 Introduction

Modern embedded software systems are characterized by execution behaviors that are becoming increasingly complex. For instance, with the emergence of heterogeneous computing platforms that combine scalar multiprocessors with specialized hardware accelerators such as Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs), the possibility of speeding up compute-intensive operations by offloading some computational activities to the accelerators has become commonplace. Complex execution behaviors also arise in multiprocessing due to inter-core synchronization. For example, this is the case for locking protocols that regulate the access to resources that are shared by tasks running on different processors, or for parallel tasks that dispatch computational activities upon multiple processors, with some of those activities being subject to precedence constraints specified according to a graph-based topology.

Such execution behaviors share the common pattern that some of the computational activities in the system may need to wait for some event to occur before continuing with their execution. In particular, for the case of computation offloading to hardware accelerators,



© Federico Aromolo, Alessandro Biondi, and Geoffrey Nelissen;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 13; pp. 13:1–13:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



the task performing the offloading must wait for the event signaling the completion of the accelerated workload. Instead, in the case of locking protocols, a task requesting access to a shared resource has to wait for the permission to access that resource in accordance to the specifics of the protocol. Finally, in the presence of precedence constraints, a subtask may have to wait for its predecessor subtasks to complete before starting its execution.

Since the delays incurred by a task when waiting for such events to occur may be significant, the typical implementation forces the task to relinquish the processor by having it suspend itself until the expected event occurs, as a way to avoid wasting processor time. The self-suspending task model was introduced to deal with the timing analysis of systems involving tasks that may suspend themselves to wait for an event to occur. This model has been extensively studied during the last decade, and proved to be a particularly effective tool to analyze the complex execution patterns exhibited by modern embedded systems from the point of view of their timing behavior [8].

Despite the effectiveness of EDF in dealing with both uniprocessor and multiprocessor scheduling problems, most of the existing analytical results for self-suspending tasks do not support the popular Earliest Deadline First (EDF) scheduling algorithm, being instead primarily focused on fixed-priority scheduling. In particular, none of the existing works provide a specialized and effective method to bound the response times of constrained-deadline self-suspending tasks in the specific case of uniprocessor systems.

Contributions. This paper presents a response-time analysis method for dynamic self-suspending tasks with constrained deadlines scheduled under EDF on a uniprocessor system. The analysis is based on a model transformation to the sporadic task model with release jitter and on the application of the exact worst-case response time (WCRT) analysis for sporadic tasks with jitter by Spuri [21]. An experimental comparison with the baseline suspension-oblivious approach, which pessimistically treats suspensions as additional computation [8], shows significant improvements in terms of the number of accepted task sets. For the less general case of task sets with implicit deadlines, in which the relative deadline of each task is equal to the minimum inter-arrival time of the task, the proposed approach is also compared with the state-of-the-art suspension-aware response time analysis by Günzel et al. [17]. In this case, the two approaches are shown to provide comparable performance.

Paper structure. The rest of this paper is organized as follows. Section 2 provides an overview of the literature on self-suspending task systems. Section 3 describes the system model and terminology considered in the paper. Section 4 presents the analytical derivation of the proposed approach and the resulting response-time analysis algorithm. The experimental results are reported and discussed in Section 5. Finally, Section 6 concludes the paper and discusses possible avenues for future work.

2 Related work

A comprehensive survey of the literature on self-suspending tasks was recently published by Chen et al. [8]. As discussed in that survey, many of the previous works on the analysis of real-time self-suspending tasks were found to be flawed. In addition to establishing a common framework for the analysis of self-suspending task systems, the survey by Chen et al. [8] aimed at collecting amendments to as many of those flawed works as possible.

Two main models exist for self-suspending tasks. The segmented self-suspending task model considers tasks whose execution behavior is determined by a fixed interleaving sequence of computation and suspension intervals, where each interval is characterized by a specific

maximum length. The dynamic self-suspending task model, on the other hand, only assumes a total maximum execution time and a total maximum suspension time, computed, respectively, across all execution and suspension intervals. In an attempt to reduce the pessimism of response-time analyses for the dynamic self-suspending task model, von der Brüggen et al. [22] introduced the hybrid suspension model, which is similar to the dynamic model but assumes a limit on the maximum number of suspension intervals allowed for each job.

The typical analysis strategies for self-suspending tasks include modeling suspension time as computation, modeling the effect of suspension on other tasks as release jitter, and modeling the effect of suspension as a blocking term in the response-time analysis.

One of the most prominent works on the analysis of dynamic self-suspending tasks under uniprocessor fixed-priority scheduling is the work by Chen et al. [7], which proposed a response-time analysis for the dynamic self-suspending task model with constrained deadlines that dominates all other existing schedulability tests by combining elements of both the jitter-based and the blocking-based analyses. Similarly to the approach in the present paper, the proof for the analysis in [7] is based on a schedule transformation procedure followed by the analysis of the transformed schedule. Later, Günzel et al. [16] generalized the approach of [7] to the case where tasks have arbitrary deadlines and their releases are modeled by arrival curves.

For the case of segmented self-suspending tasks, Nelissen et al. [20] proposed a response-time analysis based on optimization methods for tasks with constrained deadlines scheduled under uniprocessor fixed-priority scheduling. For the case of multiprocessor systems, Liu and Anderson [18] derived the first suspension-aware WCRT analysis for dynamic self-suspending tasks under global scheduling. As discussed in [8], both the fixed-priority analyses from [20] and [18] required later revision due to some incorrect statements that were discovered within the respective proof frameworks.

Concerning the analysis of self-suspending task models under EDF scheduling, Liu and Anderson [18] also proposed a response-time analysis approach for multiprocessor global EDF scheduling of arbitrary-deadline tasks, which also supports soft real-time scheduling by means of tardiness thresholds. The approach by Dong and Liu [10] provides a utilization-based schedulability test for dynamic self-suspending tasks under multiprocessor global EDF scheduling for the case of implicit deadlines, and was later shown to be equivalent to the suspension-oblivious analysis for the case of uniprocessor systems [17]. Günzel et al. [17] provided the first response-time analysis for the dynamic model under EDF, for the case of implicit deadlines. That same work showed that an earlier analysis by Devi [9] that tried to solve the same problem was indeed flawed.

Self-suspending task models see fruitful application in the analysis of hardware-accelerated task systems in the context of heterogeneous computing. The case of hardware acceleration by means of Graphics Processing Units (GPUs) was explored in the works by Dong et al. [11] and Elliot et al. [12]. Biondi et al. [4] applied the segmented suspension model to the analysis of hardware acceleration on Field-Programmable Gate Arrays (FPGAs) embedded in emerging system-on-a-chip platforms.

Numerous works on the analysis of multiprocessor synchronization protocols hinge on self-suspending task models to derive a suitable real-time analysis. In this context, self-suspending task models can capture the behavior of tasks that suspend themselves while waiting to acquire a shared resource protected by a suspension-based locking mechanism. Detailed discussion on these works can be found in the most recent survey on multiprocessor locking protocols by Brandenburg [5].

Relevant applications of self-suspending task models also include the analysis of real-time parallel workloads. Fonseca et al. [14] considered a transformation to the segmented model for the analysis of parallel tasks under multiprocessor partitioned scheduling. The event-driven

delay-induced (EDD) task model was introduced by Aromolo et al. [1] to model parallel topologies that incorporate delays in the concept of precedence constraints, with applications in the analysis of hardware-accelerated systems and of partitioned parallel tasks. The EDD task model was analyzed by means of a transformation to the dynamic self-suspending task model.

Concerning the analysis of non-suspending sporadic tasks with release jitter under uniprocessor EDF scheduling, the work by Spuri [21] generalizes previous results by Baruah et al. [2] to obtain both a feasibility test and an exact WCRT analysis based on the workload analysis approach. These approaches were later revised by George et al. [15] to provide some algorithmic efficiency improvements to the resulting analyses. To date, the technique by Spuri [21], combined with the efficiency enhancements in [15], represents a valid approach to check the feasibility of a set of sporadic tasks with jitter and to obtain their exact WCRTs.

3 System model

We consider a set $\Gamma = \{\tau_1, \dots, \tau_n\}$ of n sporadic self-suspending real-time tasks executing on a single processor and described according to the dynamic self-suspending (DSS) task model [8]. Each task τ_i releases a potentially infinite sequence of jobs $\tau_{i,1}, \dots, \tau_{i,j}, \dots$ and is characterized by a tuple (C_i, S_i, T_i, D_i) , where C_i represents the worst-case execution time (WCET), S_i is the maximum suspension time of each job of τ_i , T_i is the minimum inter-arrival time between the jobs of τ_i , and D_i is the relative deadline of the task, with $D_i \leq T_i$ (constrained deadlines). A job of τ_i may execute for up to C_i time units and may suspend itself at any point in its execution. When suspending, the job yields the processor for the execution of other tasks. The total suspension time of a job of τ_i across all its suspension intervals is upper bounded by S_i time units. The minimum inter-arrival time T_i represents the minimum amount of time separating successive jobs of τ_i .

We assume that job releases occur without jitter, so that each job $\tau_{i,j}$ is released as soon as it arrives. That is, if $a_{i,j}$ and $r_{i,j}$ represent, respectively, the arrival time and the release time of $\tau_{i,j}$, then it holds that $a_{i,j} = r_{i,j}$. Once it has arrived, a job $\tau_{i,j}$ is expected to complete its execution within D_i time units. Let $f_{i,j}$ denote the finishing time of a job $\tau_{i,j}$ of τ_i . We say that $\tau_{i,j}$ meets its deadline if $f_{i,j}$ is no greater than its absolute deadline $d_{i,j} = r_{i,j} + D_i$. The response time of a job $\tau_{i,j}$ is given by $R_{i,j} = f_{i,j} - a_{i,j} = f_{i,j} - r_{i,j}$. The worst-case response time (WCRT) R_i of a task τ_i is defined as the maximum possible response time across the jobs of τ_i . A job that is released and not yet completed is said to be pending.

Tasks in Γ are scheduled on the processor in a preemptive fashion according to the Earliest Deadline First (EDF) algorithm, which belongs to the class of job-level fixed-priority (JLFP) scheduling policies. Under EDF, each job $\tau_{i,j}$ is assigned a fixed priority level according to its absolute deadline $d_{i,j}$, such that a job with an earlier deadline has a higher priority. Ties are broken arbitrarily in case multiple jobs have the same absolute deadline.

4 Analysis

This section shows how to derive a schedulability test for a DSS task set Γ scheduled on a single processor under EDF scheduling, based on the response-time analysis (RTA) approach. In this approach, an upper bound \bar{R}_i on the WCRT is derived for each task $\tau_i \in \Gamma$; then, the task set is deemed schedulable if $\bar{R}_i \leq D_i$ holds for every task $\tau_i \in \Gamma$.

The analysis for each task τ_i consists in deriving a transformation of the task set Γ to a task set Γ'_i of sequential sporadic real-time tasks with jitter, such that the WCRT R_i of τ_i in Γ is upper bounded by the WCRT R'_i of the corresponding task τ'_i in Γ'_i . The task set Γ'_i can then be analyzed according to the analysis by Spuri [21], hence obtaining a suitable upper bound for the response time of the DSS task τ_i .

For simplicity in the presentation, we assume that execution on the processor follows a discrete-time model where a unit of time corresponds to the length of the smallest relevant time scale in the system (e.g., the length of a processor cycle). The task schedule can then be seen as a sequence of time slices, each with a length of one time unit, within which the scheduling decisions are unaltered.

4.1 Sequential sporadic tasks with jitter

Since our proposed analysis is based on the idea of transforming the set of DSS tasks Γ into a set of sequential sporadic tasks with jitter, we introduce the terminology associated to sporadic tasks with jitter. Let $\Gamma' = \{\tau'_1, \dots, \tau'_n\}$ represent a task set of sequential sporadic tasks with release jitter scheduled on a single processor under preemptive EDF. Each sporadic task with jitter τ'_i releases a potentially infinite sequence of jobs $\tau'_{i,1}, \dots, \tau'_{i,j}, \dots$ and is characterized by a tuple (C'_i, J'_i, T'_i, D'_i) , where C'_i represents the worst-case execution time (WCET), J'_i is the maximum release jitter of each job of τ'_i , T'_i is the minimum inter-arrival time between the jobs of τ'_i , and D'_i is the relative deadline of the task, with $D'_i \leq T'_i$. The minimum inter-arrival time T'_i represents the minimum amount of time separating successive arrivals of jobs of τ'_i . The maximum release jitter J'_i is the maximum time a job of τ'_i can spend waiting for release after its arrival. Specifically, letting $a'_{i,j}$ and $r'_{i,j}$ represent, respectively, the arrival time and the release time of a job $\tau'_{i,j}$ of τ'_i , it holds that $r'_{i,j} - a'_{i,j} \leq J'_i$. Once it has arrived, a job $\tau'_{i,j}$ is expected to complete its execution within D'_i time units. Let $f'_{i,j}$ denote the finishing time of a job $\tau'_{i,j}$. The absolute deadline of $\tau'_{i,j}$ is defined as $d'_{i,j} = a'_{i,j} + D'_i$, and is considered respected if $f'_{i,j} \leq d'_{i,j}$. The response time of a job $\tau'_{i,j}$ is given by $R'_{i,j} = f'_{i,j} - a'_{i,j}$. The worst-case response time (WCRT) R'_i of a task τ'_i is defined as the maximum possible response time across the jobs of τ'_i .

4.2 Schedule transformation

By sustainability of self-suspending tasks with respect to their WCETs [6], the WCRT R_i of a task $\tau_i \in \Gamma$ is produced in a schedule σ of Γ in which all jobs $\tau_{j,l}$ of all tasks $\tau_j \in \Gamma$ execute up to their respective WCETs C_j . In the following procedure, we show how to transform the schedule σ in order to obtain a preemptive EDF schedule σ' in which none of the jobs self-suspend and where the response time of at least one of the jobs of τ'_i in σ' is equal to R_i .

- Step 1** Initially set $\sigma' := \sigma$.
- Step 2** Let $\tau_{i,k}$ represent a job of τ_i in σ with response time $R_{i,k} = R_i$. Let $\tau'_{i,k}$ represent the job of σ' corresponding to $\tau_{i,k}$. Remove all jobs in σ' with lower priority than $\tau'_{i,k}$, i.e., all jobs with deadline greater than $d'_{i,k}$.
- Step 3** Replace all suspension intervals of jobs of τ'_i in σ' in which the processor is idle with execution intervals of equivalent length for τ'_i .
- Step 4** Let t_f represent the finishing time of $\tau'_{i,k}$ and t_b represent the earliest time instant in σ' at and after which the processor is continuously busy until t_f , and let $I_B = [t_b, t_f)$ represent the busy interval for job $\tau'_{i,k}$. Identify the set of carry-in jobs C_B for the busy interval I_B as the set of jobs suspended at time $t_b - 1$ and that finish at or after t_b in σ' . Remove all jobs in σ' released before t_b that do not belong to C_B .

- Step 5** Let t_0 represent the earliest release time among the jobs in C_B . Traverse all time slices in σ' within the interval $[t_0, t_b)$, from t_b down to t_0 . For each such time slice T_I in which the processor is idle, if there is at least one time slice before T_I in which the processor is busy executing a job of C_B , let T_E represent the latest of such time slices and \mathcal{J}_E represent the corresponding job, then, move the execution time of \mathcal{J}_E in T_E from T_E to T_I .
- Step 6** Let t'_b represent the (updated) earliest time instant in σ' at and after which the processor is continuously busy until t_f , and let $I'_B = [t'_b, t_f)$ represent the (extended) busy interval for job $\tau'_{i,k}$. For each job \mathcal{J}_C in C_B , if $r'_C < t'_b$, where r'_C represents the release time of \mathcal{J}_C , postpone the release time r'_C of \mathcal{J}_C to t'_b , without modifying the arrival time or the execution pattern of \mathcal{J}_C in σ' . This corresponds to introducing a release jitter of length $t'_b - a'_C$ for \mathcal{J}_C , where a'_C represents the arrival time of \mathcal{J}_C .
- Step 7** Remove all the execution that takes place at or after t_f from σ' .
- Step 8** Traverse the processor time slices in σ' located within I'_B , from t'_b up to t_f . For each such time slice T_L , if a job \mathcal{J}_L which is not one of the highest-priority jobs that are pending in T_L is executing in T_L , let \mathcal{J}_H represent any one of the highest-priority jobs that are pending in T_L , and let T_H represent the earliest time slice after T_L in which the processor is busy executing \mathcal{J}_H , then, move the execution time of \mathcal{J}_H in T_H from T_H to T_L and move the execution time of \mathcal{J}_L in T_L from T_L to T_H .

4.2.1 Transformation example

Figure 1 provides an example that illustrates the schedule transformation procedure. In the provided schedules, upwards dashed arrows, upwards solid arrows, and downwards solid arrows represent, respectively, the arrival time, the release time, and the absolute deadline of a job, while white rectangles and grey rectangles represent, respectively, execution and self-suspension for a job.

Figure 1(a) illustrates an example schedule σ of a set of DSS tasks $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. When applying the transformation procedure for task τ_1 , the transformed schedule σ' is set to be identical to σ in **Step 1**. Assume that the job $\tau_{i,k}$ identified in **Step 2** corresponds to job $\tau_{1,3}$ in the example. Figure 1(b) shows the transformed schedule σ' after **Step 3** of the transformation. Then, the transformed schedule after **Step 6** is provided in Figure 1(c), where the busy interval $I_B = [t_b, t_f)$ and the extended busy interval $I'_B = [t'_b, t_f)$ are highlighted. The set of carry-in jobs C_B for I_B is composed of the first job of τ'_2 and the first job of τ'_3 , and t_0 is set to coincide with the release of the first job of τ'_2 . Note that, in **Step 6**, the release time of the first job of τ'_2 is delayed to coincide with t'_b . Finally, Figure 1(d) provides the resulting transformed schedule σ' , obtained after **Step 8**. Note that the response time $R'_{1,3}$ of $\tau'_{1,3}$ was not altered in the transformation, i.e., $R'_{1,3} = R_{1,3}$.

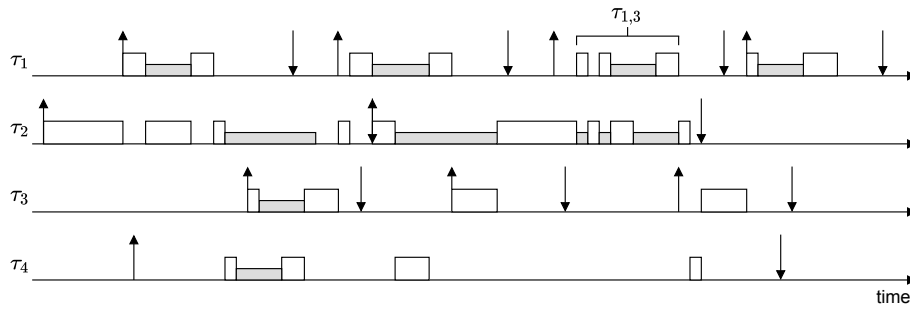
4.2.2 Properties of the transformed schedule

The following properties of σ' can be derived based on the transformation procedure.

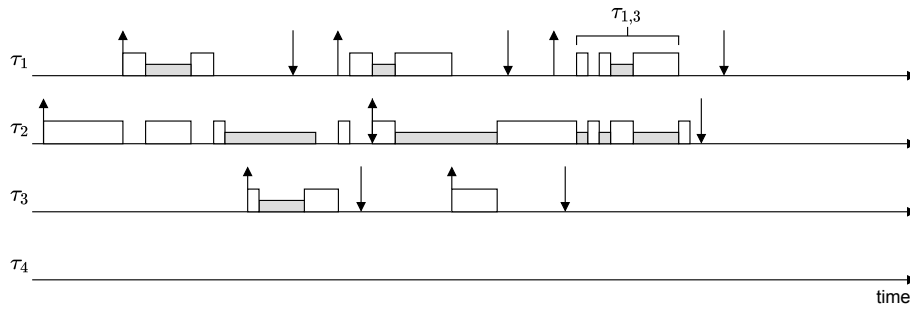
The following lemma establishes that the start of the extended busy interval t'_b happens at or before t_b .

► **Lemma 1.** *In the schedule σ' , the extended busy interval I'_B starts at or before t_b ; i.e., it holds that $t'_b \leq t_b$.*

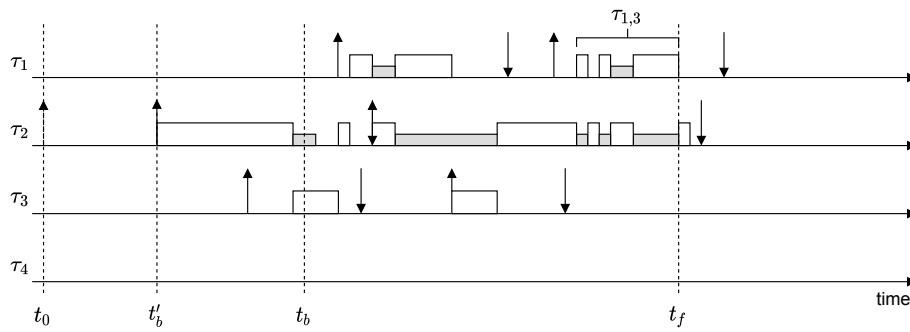
Proof. By definition, at the beginning of **Step 4**, the processor is continuously busy within the busy interval $I_B = [t_b, t_f)$, and, at the beginning of **Step 6**, the processor is continuously busy within the extended busy interval $I'_B = [t'_b, t_f)$. Note that the right end of the intervals



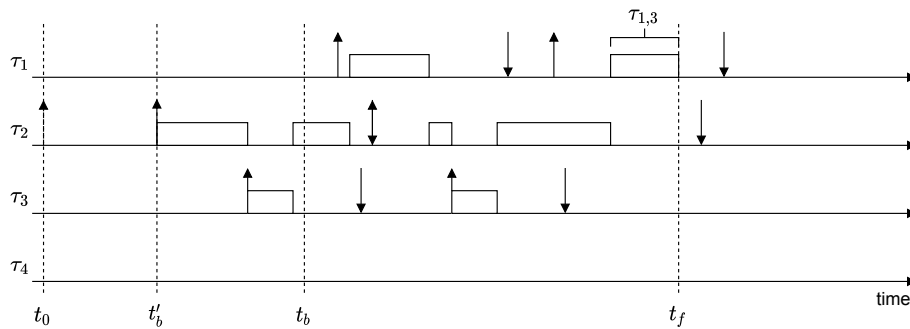
(a) Example schedule σ , equivalent to σ' after **Step 1** of the transformation.



(b) Example schedule σ' after **Step 3** of the transformation.



(c) Example schedule σ' after **Step 6** of the transformation.



(d) Example schedule σ' after **Step 8** of the transformation.

■ **Figure 1** Transformation example.

I_B and I'_B is the same, since it is defined as t_f in both cases. Given these definitions, it is sufficient to show that I'_B cannot be smaller than I_B in order to prove the lemma. In other words, it must be shown that the interval where the processor is continuously busy until t_f at the beginning of **Step 6** cannot be shorter than the interval where the processor is continuously busy until t_f at the beginning of **Step 4**. In **Step 4**, jobs released before t_b and not belonging to C_B are removed from σ' . In order to derive a contradiction, consider one such job \mathcal{J}_j and assume that it executed for at least one time slice within I_B before it was removed from σ' in **Step 4**. By the definition of the busy interval I_B , the processor must have been idle at time instant $t_b - 1$ at the beginning of **Step 4**. Therefore, \mathcal{J}_j is suspended at $t_b - 1$. In fact, if \mathcal{J}_j is not suspended at $t_b - 1$, then, by the work-conserving property of EDF, either \mathcal{J}_j is executing at $t_b - 1$ or a job with higher or equal priority than \mathcal{J}_j is executing at $t_b - 1$. This contradicts the fact that the processor is idle at $t_b - 1$. It follows that \mathcal{J}_j satisfies the definition of carry-in job, since it is suspended at time $t_b - 1$ and it executes at or after t_b . This is in contradiction with the assumption that \mathcal{J}_j does not belong to C_B . As a result, **Step 4** does not alter the execution pattern within $[t_b, t_f)$ in σ' . Finally, note that **Step 5** does not affect the execution pattern within $[t_b, t_f)$ in σ' , since the execution slices from jobs in C_B can only be moved up to $t_b - 1$ in **Step 5**. Therefore, it holds that the busy interval I'_B identified at the beginning of **Step 6** can only be larger than or equal to I_B . ◀

The following lemma shows that the execution within σ' takes place wholly within the extended busy interval I'_B .

► **Lemma 2.** *In the schedule σ' , the processor can be busy only within the extended busy interval I'_B .*

Proof. To obtain the lemma, we prove that (i) the processor cannot be busy at or after t_f and (ii) the processor cannot be busy before t'_b .

(i) In **Step 7**, all the execution that takes place at or after t_f is removed from σ' . Then, note that the execution slice interchanges within **Step 8** can only occur between time slices that were already busy at the end of **Step 7**. As a result, the processor cannot be busy at or after t_f in σ' .

(ii) In **Step 4**, all jobs released before t_b that do not belong to C_B are removed from σ' . Therefore, only jobs in C_B can execute before t_b in σ' after **Step 4**. In **Step 4**, t_0 is defined such that $t_0 \leq t_b$, and the execution slices of jobs in C_B are only moved within the interval $[t_0, t_b)$. Thus, no job of σ' executes before t_0 after **Step 5**. As a result, in case $t'_b = t_0$ in **Step 6**, no execution of jobs in C_B can take place before t'_b . Then, note that the case $t'_b < t_0$ cannot occur, since t'_b is defined in **Step 6** as the earliest time instant at and after which the processor is continuously busy until t_f , while the processor is idle before t_0 at the beginning of **Step 6**. In the following, consider the case in which $t'_b > t_0$. By definition of t'_b , the time slice $T_I = [t'_b - 1, t'_b)$ corresponding to $t'_b - 1$ is necessarily idle. Given that $t'_b > t_0$, and, by Lemma 1, $t'_b \leq t_b$, it holds that $T_I = [t'_b - 1, t'_b) \subseteq [t_0, t_b)$. If time slice T_I in $[t_0, t_b)$ is idle after **Step 5**, then it means there are no execution slices of jobs in C_B before T_I . Thus, no execution slices of jobs in C_B can take place before t'_b after **Step 6**. Finally, note that the execution slice interchanges within **Step 8** can only occur between time slices that were already busy after **Step 5**; therefore, no job of σ' executes before t'_b after **Step 8**. ◀

The following lemma shows that the processor is continuously busy within the extended busy interval I'_B after the transformation procedure.

► **Lemma 3.** *In the schedule σ' , the processor is continuously busy within the extended busy interval I'_B .*

Proof. In **Step 6**, the extended busy interval $I'_B = [t'_b, t_f)$ is identified by construction as an interval in which the processor is continuously busy. Removing the execution of jobs in σ' that takes place at or after t_f in **Step 7** does not alter the execution pattern within I'_B . Then, the execution slice interchanges within **Step 8** can only occur between time slices that were already busy at the beginning of **Step 6**. Therefore, the processor is continuously busy within the extended busy interval I'_B in the transformed schedule σ' . ◀

The following lemma shows that jobs of the task under analysis τ'_i cannot be categorized as carry-in jobs in **Step 4** of the schedule transformation.

► **Lemma 4.** *The set C_B of carry-in jobs for the busy interval I_B does not contain any job of τ'_i .*

Proof. By the definition of the busy interval I_B in **Step 4**, the processor is idle at time instant $t_b - 1$. Then, in order for a job $\tau'_{i,l}$ of τ'_i to belong to the set C_B , it must hold that $\tau'_{i,l}$ is suspended when the processor is idle at time $t_b - 1$ in **Step 4**. This is impossible since, in **Step 3**, all suspension intervals of jobs of τ'_i in which the processor is idle are replaced with execution intervals of equivalent length for that job. ◀

The following lemma shows that the response time of job $\tau_{i,k}$ is preserved after the transformation.

► **Lemma 5.** *The response time of job $\tau'_{i,k}$ in σ' is equal to the response time of $\tau_{i,k}$ in σ , i.e., it holds that $R'_{i,k} = R_{i,k}$.*

Proof. In **Step 2**, all jobs with priority lower than that of $\tau'_{i,k}$ are removed from the schedule σ' . Therefore, $\tau'_{i,k}$ is one of the jobs that share the lowest priority in the schedule σ' . In **Step 3**, additional execution slices of $\tau'_{i,k}$ can only be added before its finishing time $f'_{i,k} = f_{i,k}$. Then, in **Step 4**, only jobs that are released before t_b can be removed from σ' . Note that, by definition, the right end of the busy interval $I_B = [t_b, t_f)$ defined in **Step 4** corresponds to the finishing time $f'_{i,k}$ of $\tau'_{i,k}$, with $f'_{i,k} = f_{i,k}$. In addition, in **Step 3**, all suspension intervals of $\tau'_{i,k}$ in which the processor is idle are replaced with execution intervals of equivalent length for $\tau'_{i,k}$; thus, within the interval $[a'_{i,k}, f'_{i,k})$, the processor is either busy executing $\tau'_{i,k}$ or another job with higher or equal priority than $\tau'_{i,k}$. It follows that the start of the busy period t_b must necessarily occur at or before the arrival time $a'_{i,k}$ of $\tau'_{i,k}$. Therefore, **Step 4** does not affect the execution of $\tau'_{i,k}$. Similarly, **Step 5** only affects the execution of jobs belonging to C_B , which, by definition, are released before t_b . Thus, the execution of $\tau'_{i,k}$ is not affected by **Step 5**. Then, in **Step 7**, removing the execution of jobs in σ' taking place at or after t_f does not affect the finishing time of $\tau'_{i,k}$. Finally, in **Step 8**, execution slices of a job \mathcal{J}_H can only be anticipated to an earlier time slice T_E if \mathcal{J}_H has higher priority than the job executing in T_E . Since $\tau'_{i,k}$ is one of the jobs that share the lowest priority in the schedule σ' , it is not possible that the final execution slice of $\tau'_{i,k}$ is anticipated in **Step 8**. Therefore, the finishing time of $\tau'_{i,k}$ after the transformation is given by $f'_{i,k} = f_{i,k}$. In addition, since arrival times for jobs of σ' are not modified with respect to the corresponding jobs of σ within the transformation procedure, it holds that $a'_{i,k} = a_{i,k}$. It follows that $R'_{i,k} = f'_{i,k} - a'_{i,k} = f_{i,k} - a_{i,k} = R_{i,k}$. ◀

The following lemma shows that jobs in σ' are scheduled in accordance with the preemptive EDF scheduling policy.

13:10 Response-Time Analysis for Self-Suspending Tasks Under EDF Scheduling

► **Lemma 6.** *For any time slice T_σ in the schedule σ' , the processor executes one of the jobs with earliest absolute deadline that are pending in T_σ , if any.*

Proof. The statement trivially holds for any time slice of σ' in which no job is ready for execution. By Lemma 2, the time slices in which the processor is busy are limited to the busy interval I'_B . Concerning the time slices within I'_B , in **Step 8**, for each time slice T_L within I'_B , the schedule σ' is modified such that one of the highest-priority jobs that are pending in T_L is executing in T_L . The statement follows since the priority ordering in **Step 8** is determined according to the EDF scheduling policy. ◀

The following lemma guarantees that the execution of a job in σ' cannot occur before the release time of the corresponding job in σ .

► **Lemma 7.** *Consider a job $\tau'_{j,l}$ of a task τ'_j in σ' . The execution of $\tau'_{j,l}$ that takes place in σ' does not start before $r_{j,l}$.*

Proof. The execution of the job $\tau_{j,l}$ corresponding to job $\tau'_{j,l}$ in σ cannot start before its release time $r_{j,l}$. In the following, we show that no execution slices for job $\tau'_{j,l}$ are added before $r_{j,l}$ within the transformation procedure. In **Step 3**, additional execution time can only be introduced for a job $\tau'_{j,l}$ in a time slice T_S where $\tau'_{j,l}$ is suspended in σ , meaning that $\tau'_{j,l}$ was released before T_S . In **Step 5**, execution slices of jobs in C_B can only be delayed to a later time slice, meaning that no execution slices of $\tau'_{j,l}$ are introduced before $r_{j,l}$. Finally, consider that, in case an execution slice of $\tau'_{j,l}$ is moved as part of **Step 8**, then either one of the following scenarios occurs:

1. An execution slice of $\tau'_{j,l}$ originally occurring at T_H is anticipated to T_L . By construction of **Step 8**, in order for this situation to occur, $\tau'_{j,l}$ must have been pending in T_L , meaning that it was released at or before the start of T_L .
2. An execution slice of $\tau'_{j,l}$ originally occurring at T_L is delayed to T_H , which occurs after T_L by construction. Since job $\tau'_{j,l}$ was executing in T_L , the release time $r_{j,l}$ must have occurred at or before the start of T_H .

As a result, no matter how many times the execution slices of $\tau'_{j,l}$ are moved in **Step 8**, no execution slices of $\tau'_{j,l}$ are introduced before $r_{j,l}$. ◀

The following lemma shows that jobs in σ' do not execute before their release time.

► **Lemma 8.** *Consider a job $\tau'_{j,l}$ of a task τ'_j in σ' . Job $\tau'_{j,l}$ does not execute before its release time $r'_{j,l}$.*

Proof. The release time of a job $\tau'_{j,l}$ in σ' can only be modified in **Step 6** of the transformation. In case $\tau'_{j,l}$ does not belong to C_B , its release time is not modified in **Step 6**, meaning that $r'_{j,l} = r_{j,l}$ in the transformed schedule σ' . By Lemma 7, the execution of a job $\tau'_{j,l}$ in σ' does not start before $r_{j,l}$. Therefore, in case $\tau'_{j,l} \notin C_B$, $\tau'_{j,l}$ does not execute before $r'_{j,l}$ in σ' . Similarly, in case $\tau'_{j,l} \in C_B$ and $r'_{j,l} \geq t'_b$ at the beginning of **Step 6**, the release time of $\tau'_{j,l}$ is not modified, therefore $r'_{j,l} = r_{j,l}$, and, by Lemma 7, the execution of $\tau'_{j,l}$ in σ' does not start before $r_{j,l}$. Finally, in case $\tau'_{j,l} \in C_B$ and $r'_{j,l} < t'_b$ at the beginning of **Step 6**, then the release time of $\tau'_{j,l}$ is set to $r'_{j,l} = t'_b$. By Lemma 2, none of the jobs in σ' execute outside the busy interval $I'_B = [t'_b, t'_f)$, thus $\tau'_{j,l}$ does not execute before $r'_{j,l} = t'_b$. ◀

The following lemma provides an upper bound on the release jitter introduced for the jobs in σ' .

► **Lemma 9.** *Consider a job $\tau'_{j,l}$ of a task τ'_j in σ' . The release jitter of $\tau'_{j,l}$ is upper bounded by $R_j - C_j$.*

Proof. Since arrival times are not altered for jobs of σ' with respect to σ , it holds that $a'_{j,l} = a_{j,l}$. Within the schedule transformation, the release time of jobs in σ' can only be postponed for jobs of C_B (in **Step 6**). Therefore, in case $\tau'_{j,l}$ does not belong to C_B , the release jitter of $\tau'_{j,l}$ is given by $r'_{j,l} - a'_{j,l} = r_{j,l} - a_{j,l} = 0$. In the following, consider the case in which $\tau'_{j,l}$ belongs to C_B . In case the release time $r'_{j,l}$ is not modified in **Step 6**, then the release jitter of $\tau'_{j,l}$ is 0. Otherwise, the release time $r'_{j,l}$ is set to t'_b . Therefore, in the latter case, the resulting release jitter for $\tau'_{j,l}$ is given by $r'_{j,l} - a'_{j,l} = t'_b - a_{j,l}$. By definition of carry-in job, the finishing time of $\tau'_{j,l}$ is greater than or equal to t_b in **Step 4**. Furthermore, because **Step 5** does not alter the schedule σ' at or after t_b , the finishing time of $\tau'_{j,l}$ is unchanged at the end of **Step 5**, i.e., just before the release time $r'_{j,l}$ is postponed in **Step 6** to yield the release jitter of $\tau'_{j,l}$. In addition, note that, by the assumption that job $\tau_{j,l}$ executes for its WCET C_j in σ , and since none of the steps in the transformation up to **Step 6** reduce or increase the total execution time of any job in C_B , $\tau'_{j,l}$ executes for C_j units of time in σ' at the beginning of **Step 6**. Finally, by Lemma 8, the execution of $\tau'_{j,l}$ only takes place at or after $r'_{j,l} = t'_b$ in **Step 6**. It follows that, at the beginning of **Step 6**, $f_{j,l} \geq t'_b + C_j$. As a result, the release jitter introduced for $\tau'_{j,l}$ in **Step 6** can be upper bounded as $t'_b - a_{j,l} \leq f_{j,l} - C_j - a_{j,l} = R_{j,l} - C_j \leq R_j - C_j$. ◀

4.3 Model transformation to enable the response-time analysis

To prove that the schedule σ' resulting from the schedule transformation is suitable to be analyzed as a set of sequential sporadic tasks with jitter, we first define a legal schedule for a set of sequential sporadic tasks with jitter under preemptive EDF scheduling.

► **Definition 10.** *A schedule σ' is considered legal with respect to preemptive EDF scheduling of a set Γ' of sequential sporadic tasks with jitter if the following statements hold for each job $\tau'_{j,l}$ of all tasks τ'_j in σ' :*

- **Property 1.** *The minimum inter-arrival time constraint is satisfied; i.e., if $l > 1$, it holds that $a'_{j,l} \geq a'_{j,l-1} + T'_j$.*
- **Property 2.** *The absolute deadline $d'_{j,l}$ of $\tau'_{j,l}$ is such that $d'_{j,l} = a'_{j,l} + D'_j$.*
- **Property 3.** *The processor does not execute $\tau'_{j,l}$ for more than C'_j units of time.*
- **Property 4.** *The release of $\tau'_{j,l}$ takes place at or after its arrival; i.e., it holds that $r'_{j,l} \geq a'_{j,l}$.*
- **Property 5.** *The processor does not execute $\tau'_{j,l}$ before its release time $r'_{j,l}$.*
- **Property 6.** *The release jitter constraint is satisfied, i.e., it holds that $r'_{j,l} - a'_{j,l} \leq J'_j$.*
- **Property 7.** *For each time slice from the release time $r'_{j,l}$ up to the finishing time $f'_{j,l}$ of $\tau'_{j,l}$, the processor is either busy executing $\tau'_{j,l}$ or another job with absolute deadline smaller or equal than that of $\tau'_{j,l}$.*

The following lemma shows that σ' is a legal preemptive EDF schedule of a set of sequential sporadic tasks with jitter.

► **Lemma 11.** *Consider a task set $\Gamma'_i = \{\tau'_1, \dots, \tau'_n\}$ of sequential sporadic tasks with release jitter, with $\tau'_i = (C_i + S_i, 0, T_i, D_i)$ and $\tau'_j = (C_j, R_j - C_j, T_j, D_j)$ for all $\tau'_j \neq \tau'_i$. The transformed schedule σ' is a legal schedule of Γ'_i under preemptive EDF scheduling.*

Proof. In the following, we show that σ' complies with Definition 10 with respect to task set Γ'_i . First, note that the arrival times and the absolute deadlines of the jobs in σ' are kept equal to the arrival times and absolute deadlines of the corresponding jobs in σ . Therefore,

Property 1 and Property 2 in Definition 10 hold for each job in σ' . Then, consider that the schedule transformation procedure does not increment the cumulative execution time of jobs of τ'_j in σ' such that $\tau'_j \neq \tau'_i$, thus the amount of execution for such jobs is within C_j . In addition, the execution time of jobs of τ'_i is not incremented by more than S_i (in **Step 3**). Therefore, Property 3 holds for each job in σ' . Within the schedule transformation, the release times of jobs in σ' can only be modified for jobs of C_B (in **Step 6**). By Lemma 4, jobs of τ'_i do not belong to the set C_B ; therefore, it holds that $r'_{i,l} = r_{i,l}$ for each job $\tau'_{i,l}$ of τ'_i . As a result, $r'_{i,l} = a'_{i,l}$ holds for each job $\tau'_{i,l}$ of τ'_i . Then, consider a job $\tau'_{j,l}$ that belongs to C_B . If at the beginning of **Step 6** $r'_{j,l} < t'_b$, then the release time of $\tau'_{j,l}$ is postponed to t'_b ; therefore, it holds that $r'_{j,l} \geq a'_{j,l}$ in the transformed schedule σ' . Otherwise, the release time of $\tau'_{j,l}$ is not modified, i.e., $r'_{j,l} = a'_{j,l}$. Thus, Property 4 holds for all jobs in σ' . Property 5 and Property 6 follow directly from Lemma 8 and Lemma 9, respectively, for each job in σ' . Finally, by Lemma 6, for any time slice T_σ in the schedule σ' , the processor executes one of the jobs with earliest absolute deadline that are pending in T_σ , if any. Therefore, Property 7 holds for each job in σ' . ◀

The following theorem shows how to obtain a task set Γ'_i of sequential sporadic tasks with jitter that can be analyzed in order to obtain a WCRT upper bound for task τ_i .

► **Theorem 12.** *Given a task set Γ of DSS tasks, the WCRT R_i of a task $\tau_i \in \Gamma$ is upper bounded by the WCRT R'_i of a sequential sporadic task τ'_i in Γ'_i , where $\Gamma'_i = \{\tau'_1, \dots, \tau'_n\}$ is a set of sequential sporadic tasks with release jitter, with $\tau'_i = (C_i + S_i, 0, T_i, D_i)$ and $\tau'_j = (C_j, R_j - C_j, T_j, D_j)$ for all $\tau'_j \neq \tau'_i$.*

Proof. By Lemma 5, the response time of job $\tau'_{i,k}$ in σ' is equal to the response time of $\tau_{i,k}$ in σ , which is in turn equivalent to the WCRT R_i of the task under analysis τ_i ; i.e., it holds that $R'_{i,k} = R_{i,k} = R_i$. In addition, since by Lemma 11 σ' represents a legal schedule of Γ'_i under preemptive EDF scheduling, it holds that $R'_{i,k} \leq R'_i$. It follows that $R_i \leq R'_i$. ◀

4.4 Response-time analysis algorithm

The schedulability of a task set Γ of DSS tasks can be verified by means of the iterative approach described in Algorithm 1. The algorithm produces a vector of WCRT upper bounds $\bar{\mathbf{R}} = [\bar{R}_1, \dots, \bar{R}_n]$ for all tasks in Γ using Theorem 12 (MODELTRANSFORMATION at line 10), starting from the initial condition in which \bar{R}_i is set to D_i for each $\tau_i \in \Gamma$. Then, the algorithm iteratively applies Theorem 12 to each task $\tau_i \in \Gamma$ in order to obtain the WCRT R'_i of task τ'_i by means of the response-time analysis approach by Spuri [21] (JITTERANALYSIS at line 11). At each iteration, the value of \bar{R}_i in $\bar{\mathbf{R}}$ is set to the newly obtained R'_i in case $R'_i < \bar{R}_i$, and the task set is deemed schedulable if $R'_i \leq D_i$ holds for each $\tau_i \in \Gamma$. Otherwise, the iterative loop continues until either the task set is deemed schedulable or the vector $\bar{\mathbf{R}}$ is not updated within the iteration, in which case the task set is deemed not schedulable.

The use of this iterative algorithm is supported by the following reasoning. Assume that the behavior of the preemptive EDF scheduler is altered by means of a run-time mechanism \mathcal{M} that forcibly terminates all jobs at the time of their absolute deadline. With mechanism \mathcal{M} in place, the relative deadline D_i represents a valid upper bound on the WCRT R_i of a task τ_i in Γ . Then, if in a given iteration of the algorithm the WCRT upper bound R'_i obtained for each task τ_i is found to be lower than or equal to the corresponding deadline D_i , then all the possible jobs of all tasks in Γ will terminate within their absolute deadline. In this situation, the mechanism \mathcal{M} does not need to prevent execution for any job, therefore its presence is irrelevant and the resulting behavior is equivalent to standard preemptive

EDF scheduling, wherein \mathcal{M} is not deployed. Otherwise, if in the same iteration at least one of the WCRT upper bounds R'_i for a task τ_i is found to be greater than the corresponding relative deadline D_i , then the task set cannot be deemed schedulable at that iteration. Then, if at least one of the WCRT upper bounds in $\bar{\mathbf{R}}$ was updated, the algorithm proceeds to the next iteration to potentially reduce the WCRT upper bounds of the other tasks in Γ . This reasoning was also adopted in [19] to obtain a similar iterative approach for the derivation of WCRT upper bounds in systems where tasks synchronize their access to shared resources. Note that, by construction, in a given iteration beyond the first, each of the values in $\bar{\mathbf{R}}$ is less or equal than the corresponding value at the previous iteration, and that the algorithm terminates as soon as none of the values in $\bar{\mathbf{R}}$ are updated after an iteration.

When applying Theorem 12 within the algorithm to analyze a task $\tau_i \in \Gamma$ (at line 10), note that the exact value of the WCRT R_j of each task $\tau_j \neq \tau_i$ is not known; therefore, when constructing Γ'_i , $\bar{R}_j - C_j$ must be used as an upper bound of the jitter of τ'_j in place of $J'_j = R_j - C_j$. To ensure that the algorithm remains consistent with this substitution, consider a generic iteration of the algorithm. In case \bar{R}_j was never updated during the previous iterations of the algorithm, then $\bar{R}_j = D_j$, and, assuming \mathcal{M} is active, $D_j = \bar{R}_j \geq R_j$. Instead, if \bar{R}_j was updated in at least one of the previous iterations of the algorithm, then \bar{R}_j must have been set to $\bar{R}_j = R'_j$, where R'_j was obtained by analyzing τ_j by means of task set Γ'_j in Theorem 12 with respect to previous values of the WCRT upper bounds in $\bar{\mathbf{R}}$. In this case, by Theorem 12, it holds that $R_j \leq R'_j$, with R'_j computed with respect to Γ'_j ; thus, $R_j \leq \bar{R}_j$. As a result, $\bar{R}_j \geq R_j$ holds for both cases. Therefore, since $\bar{R}_j - C_j \geq R_j - C_j$, and since increasing the maximum jitter parameter for a task in a task set of sporadic tasks with jitter cannot reduce the WCRT of tasks in that task set, $\bar{R}_j - C_j$ can be used as a safe upper bound on the jitter J'_j of τ'_j for the analysis of τ_i .

Note that the response-time analysis by Spuri [21] can only be applied to systems that are not overloaded, i.e., to those systems for which the system utilization factor $U' = \sum_{\tau'_i \in \Gamma'} \frac{C'_i}{T'_i}$ does not exceed one. Therefore, this condition must be verified in Algorithm 1 before a task set Γ'_i can be analyzed. Given a task set Γ'_i generated to analyze a task τ_i in Γ , this precondition on the system utilization is satisfied if $\frac{S_i}{T_i} + \sum_{\tau_j \in \Gamma} \frac{C_j}{T_j} \leq 1$. This is because the WCET C'_i of τ'_i is incremented by S_i with respect to the original task $\tau_i \in \Gamma$, while the WCET C'_j of tasks $\tau'_j \neq \tau'_i$ is kept equal to C_j . However, note that the resulting utilization factor for each task set Γ'_i to be analyzed in Algorithm 1 is independent of the values of the jitter J'_i , i.e., it is independent of the values of the elements of $\bar{\mathbf{R}}$. Therefore, it is sufficient to check if $\frac{S_i}{T_i} + \sum_{\tau_j \in \Gamma} \frac{C_j}{T_j} \leq 1$ holds for each task under analysis $\tau_i \in \Gamma$ before starting the iterative refinement of the WCRT upper bounds $\bar{\mathbf{R}}$ in Algorithm 1 (NECESSARYCONDITIONS at line 2).

Finally, note that the iterative loop does not need to terminate immediately in case the task set is deemed to be schedulable (i.e., when $R'_i \leq D_i$ holds for each $\tau_i \in \Gamma$). In fact, it is possible to obtain tighter WCRT upper bounds by performing additional iterations with the updated vector $\bar{\mathbf{R}}$.

5 Experimental results

This section presents the results of an experimental evaluation of the proposed response-time analysis approach. For the case of constrained deadlines, we propose a comparison with the suspension-oblivious approach. Then, for the case of implicit deadlines, where the relative deadline of each task is equal to its minimum inter-arrival time, the proposed approach is also compared with the response-time analysis technique by Günzel et al. [17].

■ **Algorithm 1** Schedulability analysis for a task set Γ .

```

1: procedure SCHEDULABILITYTEST( $\Gamma$ )
2:   if NECESSARYCONDITIONS( $\Gamma$ ) = FALSE then
3:     return FALSE
4:   end if
5:    $\forall \tau_i \in \Gamma, \bar{R}_i \leftarrow D_i$ 
6:   update  $\leftarrow$  TRUE
7:   while update = TRUE do
8:     update  $\leftarrow$  FALSE
9:     for all  $\tau_i \in \Gamma$  do
10:       $\Gamma'_i \leftarrow$  MODELTRANSFORMATION( $\Gamma, i, \bar{\mathbf{R}}$ )
11:       $R'_i \leftarrow$  JITTERANALYSIS( $\Gamma'_i, i$ )
12:      if  $R'_i < \bar{R}_i$  then
13:         $\bar{R}_i \leftarrow R'_i$ 
14:        update  $\leftarrow$  TRUE
15:      end if
16:    end for
17:    if  $\forall \tau_i \in \Gamma, R'_i \leq D_i$  then
18:      return TRUE
19:    end if
20:  end while
21:  return FALSE
22: end procedure

```

5.1 Experimental setup

The proposed experiments are based on the analysis of randomly generated task sets. The task set generator used in the experiments was instrumented as follows. The number of tasks generated for each task set is set to a fixed number n . For each task set, the UUniFast algorithm by Bini and Buttazzo [3] was used to generate a set of utilization values U_i , such that $U = \sum_{\tau_i \in \tau_i} U_i$, where U is a generation parameter representing the system utilization, which is varied within the experiments. For each task τ_i in the randomly generated task set, the minimum inter-arrival time was selected from a discrete log-uniform distribution in the range $[T_{min}, T_{max}]$, where T_{min} and T_{max} are generation parameters representing the minimum and the maximum possible value of T_i , as suggested by Emberson et al. [13]. The WCET of τ_i was then set to $C_i = U_i \cdot T_i$. The maximum suspension time S_i of τ_i was selected from a discrete uniform distribution in the range $[(T_i - C_i) \cdot \beta_{min}, (T_i - C_i) \cdot \beta_{max}]$, where β_{min} and β_{max} are generation parameters such that $\beta_{min} \in [0, 1]$ and $\beta_{max} \in [0, 1]$. The relative deadline D_i of τ_i was selected from a discrete uniform distribution in the range $[C_i + (T_i - C_i) \cdot \alpha, T_i]$, where α is a generation parameter such that $\alpha \in [0, 1]$ for the experiments with constrained deadlines, so that $D_i \leq T_i$, and is instead equal to 1 for the experiments with implicit deadlines, so that $D_i = T_i$. In the experiments, the system utilization U is varied from 0 to 1 in increments of 0.05. For each value of U , 1000 task sets were tested using the approaches under evaluation. The performance metric for the experiments is the schedulability ratio with respect to a specific system utilization U ; i.e., the ratio of task sets deemed schedulable by a specific analysis approach over the number of task sets generated for the system utilization point U .

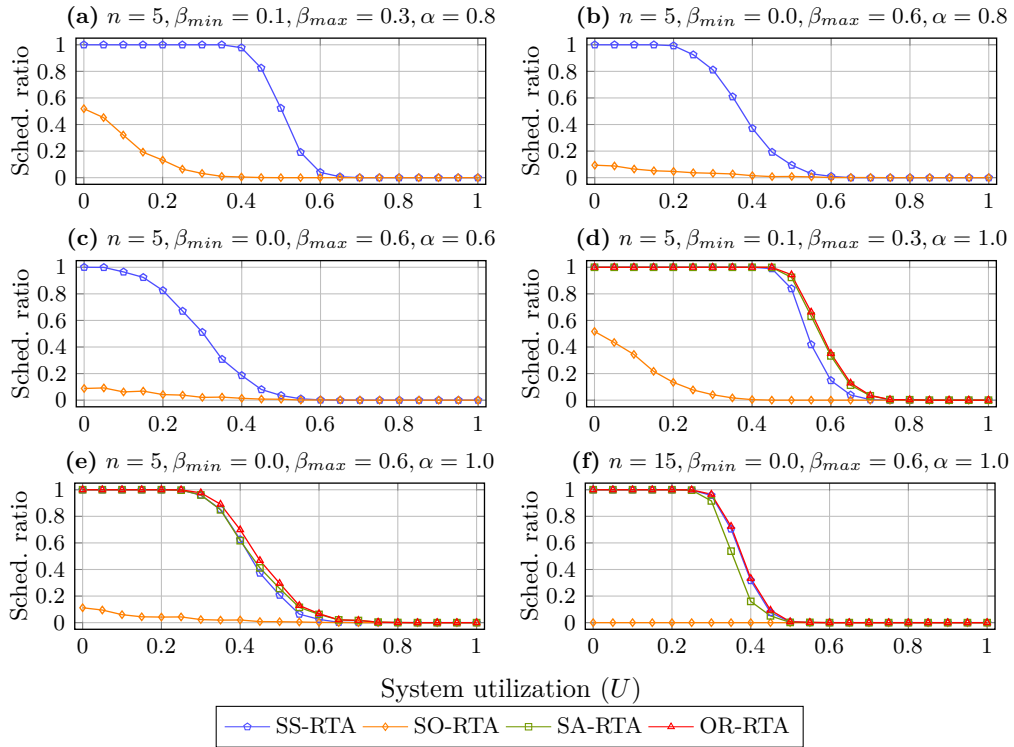
5.2 Results with constrained deadlines

For the case of constrained deadlines, the performance of the proposed approach (**SS-RTA**) in terms of schedulability ratio is compared with that of the suspension-oblivious RTA approach (**SO-RTA**). In the **SO-RTA** approach, suspensions are regarded as additional computation time, and the resulting task set is evaluated with the EDF analysis by Spuri [21]. In particular, the task set analyzed in **SO-RTA** is constructed as $\Gamma' = \{\tau'_1, \dots, \tau'_n\}$, where $\tau'_i = (C_i + S_i, 0, T_i, D_i)$ for each $\tau_i \in \Gamma$. Figures 2(a)-(c) report the results of the experiments with constrained deadlines. In these experiments, the values of T_{min} and T_{max} were set to 100 and 1000, respectively. Figure 2(a) shows that the proposed approach outperforms the suspension-oblivious approach by a significant margin, even with moderate amounts of suspension. When the parameter β_{max} is increased to 0.6 (Figure 2(b)), the schedulability ratio obtained with the suspension-oblivious analysis approaches 0, even for low values of utilization. As shown in Figure 2(c), the proposed approach retains significant schedulability ratios even with the shorter deadlines introduced by generating task sets with shorter relative deadlines (i.e., with a smaller value of α).

5.3 Results with implicit deadlines

The results of the experiments on implicit deadlines are provided in Figures 2(d)-(f). In these experiments, the proposed approach (**SS-RTA**) is compared with the suspension-oblivious approach (**SO-RTA**) and the state-of-the-art RTA for implicit deadlines by Günzel et al. [17] (**SA-RTA**). In this case, for the **SO-RTA** approach, it is sufficient to inflate the WCETs of each task by the maximum suspension time and to check whether the utilization of the resulting task set is less than or equal to 1. In addition, the performance of the schedulability test obtained with the logic OR of **SS-RTA** and **SA-RTA**, which deems a task set under analysis schedulable in case at least one of **SS-RTA** and **SA-RTA** deems the task set schedulable, is reported in the experiments (**OR-RTA**). The values of T_{min} and T_{max} are again set to 100 and 1000, respectively. Figure 2(d) shows that, when relatively small values of β_{min} and β_{max} are applied, **SA-RTA** outperforms **SS-RTA** by a slight margin. Nonetheless, it should be noted that the combination of the two approaches (**OR-RTA**) provides some improvement over using **SA-RTA** by itself. This means that the two approaches are not comparable, in the sense that there exist task sets that are deemed schedulable by **SS-RTA** and that are deemed not schedulable by **SA-RTA**, and vice-versa. The gap between the two approaches is reduced with larger values for the maximum suspension time of the generated tasks, i.e., when β_{max} is increased to 0.6 (Figure 2(e)). Finally, Figure 2(f) shows that the performance of the proposed approach surpasses the performance of **SA-RTA** when more tasks are included in each task set ($n = 15$).

Overall, these experiments show that the performance levels of the proposed approach **SS-RTA** and of the state-of-the-art approach **SA-RTA** are comparable, and that neither of the methods dominates the other. In fact, the strongest performance is obtained with the combined approach **OR-RTA**, which theoretically dominates both approaches and provides slight empirical improvements under specific system configurations. It should be noted that the main advantage of **SS-RTA** over **SA-RTA** is that it allows evaluating task sets with constrained deadlines in addition to task sets with implicit deadlines. Finally, the experiments show that both approaches vastly outperform the basic suspension-oblivious approach, which is only capable of accepting a very limited number of task sets under the evaluated scenarios.



■ **Figure 2** Comparison of the proposed RTA approach with state-of-the-art techniques in terms of the schedulability ratio obtained with different system configurations.

6 Conclusions and future work

This paper presented a response-time analysis for dynamic self-suspending tasks under preemptive EDF scheduling with constrained deadlines. The analysis is based on a model transformation to sporadic tasks with release jitter and on the subsequent application of a state-of-the-art analysis for the target task model. Experiments on randomly generated task sets compared the performance of the proposed approach in terms of schedulability ratio in the case of both implicit and constrained deadlines. The proposed approach significantly outperformed the baseline suspension-oblivious analysis in all the evaluated scenarios. Then, the approach was shown to provide comparable performance with the state-of-the-art response-time analysis for implicit deadlines by Günzel et al. [16]. Most importantly, the schedulability test which combines the two analyses was shown to outperform both techniques, meaning that the proposed approach does not dominate the response-time analysis by Günzel et al. [16] and vice-versa. Future work should consider leveraging the insights from both techniques to obtain a unifying analysis which can provide tighter WCRT upper bounds for the analysis of constrained-deadline self-suspending task systems under EDF. In addition, the proposed approach can be applied to the analysis of the EDD task model [1] towards the derivation of a response-time analysis for parallel tasks scheduled by the partitioned EDF algorithm and the analysis of hardware acceleration patterns under EDF scheduling.

References

- 1 Federico Aromolo, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Event-driven delay-induced tasks: Model, analysis, and applications. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*, pages 53–65. IEEE, 2021.

- 2 Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.
- 3 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
- 4 Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, pages 1–12. IEEE, 2016.
- 5 Björn B. Brandenburg. Multiprocessor real-time locking protocols. In Yu-Chu Tian and David Charles Levy, editors, *Handbook of Real-Time Computing*, pages 1–99. Springer, 2020.
- 6 Felipe Cerqueira, Geoffrey Nelissen, and Björn B. Brandenburg. On strong and weak sustainability, with an application to self-suspending real-time tasks. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, pages 26:1–26:21, 2018.
- 7 Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, pages 61–71. IEEE, 2016.
- 8 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, and Georg von der Brüggen. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.
- 9 UmaMaheswari C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 23–30. IEEE, 2003.
- 10 Zheng Dong and Cong Liu. Closing the loop for the selective conversion approach: A utilization-based test for hard real-time suspending task systems. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, pages 339–350. IEEE, 2016.
- 11 Zheng Dong, Cong Liu, Soroush Bateni, Kuan-Hsun Chen, Jian-Jia Chen, Georg von der Brüggen, and Junjie Shi. Shared-resource-centric limited preemptive scheduling: A comprehensive study of suspension-based partitioning approaches. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018)*, pages 164–176. IEEE, 2018.
- 12 Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS 2013)*, pages 33–44. IEEE, 2013.
- 13 Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.
- 14 José Fonseca, Geoffrey Nelissen, Vincent Nélis, and Luís Miguel Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES 2016)*, pages 1–10. IEEE, 2016.
- 15 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Research Report RR-2966, INRIA, France, 1996.
- 16 Mario Günzel, Niklas Ueter, and Jian-Jia Chen. Suspension-aware fixed-priority schedulability test with arbitrary deadlines and arrival curves. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium (RTSS 2021)*, pages 418–430. IEEE, 2021.
- 17 Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. Suspension-aware earliest-deadline-first scheduling analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4205–4216, 2020.
- 18 Cong Liu and James H. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 271–281. IEEE, 2013.

13:18 Response-Time Analysis for Self-Suspending Tasks Under EDF Scheduling

- 19 Geoffrey Nelissen and Alessandro Biondi. The SRP resource sharing protocol for self-suspending tasks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, pages 361–372. IEEE, 2018.
- 20 Geoffrey Nelissen, José Fonseca, Gurulingesh Raravi, and Vincent Nélis. Timing analysis of fixed priority self-suspending sporadic tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 80–89. IEEE, 2015.
- 21 Marco Spuri. Analysis of deadline scheduled real-time systems. Research Report RR-2772, INRIA, France, 1996.
- 22 Georg von der Brüggen, Wen-Hung Huang, and Jian-Jia Chen. Hybrid self-suspension models in real-time embedded systems. In *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017)*, pages 1–9. IEEE, 2017.

An Approach to Formally Specifying the Behaviour of Mixed-Criticality Systems

Alan Burns ✉

University of York, UK

Cliff B. Jones ✉

Newcastle University, Newcastle upon Tyne, UK

Abstract

This paper proposes a formal framework for describing the relationship between a criticality-aware scheduler and a set of application tasks that are assigned different criticality levels. The exposition employs a series of examples starting with scheduling simple jobs and then moving on to mixed-criticality robust and resilient tasks. The proposed formalism extends the rely-guarantee approach, which facilitates formal reasoning about the functional behaviour of concurrent systems, to address *real-time* properties.

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

Keywords and phrases real-time, scheduling, mixed criticality, rely/guaranteed conditions

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.14

Funding This research has been supported in part by EPSRC (UK) grants, STRATA and MCCps and by Leverhulme Trust grant RPG-2019-020.

Acknowledgements The authors acknowledge useful suggestions made by Iain Bate, Sanjoy Baruah and Ian Hayes.

1 Introduction

Since Vestal published his seminal paper in 2007 [61], there have been a wealth of models and protocols published [16, 17] on the topic of Mixed Criticality Systems (MCS). One of the aims of this wide ranging set of techniques is to improve the survivability of systems by providing a variety of degraded behaviours that can take effect if the system experiences overrunning execution times.

Inevitably these techniques require significant support from the underlying operating system. Unfortunately commercially-available, general-purpose, RTOSs do not provide this support. Hence, in order to utilise many of the more advanced scheduling ideas that are to be found in the MCS literature, it is necessary to develop the code for a bespoke scheduler as part of the application. Programming languages such as Ada [11] do provide the primitives necessary for this software to be developed but to deliver a reliable MCS scheduler the MCS protocols and models must be precisely specified. Research papers that focus on the algorithmic properties of protocols tend to give, at best, informal descriptions of the actual required run-time behaviour of the required scheduler.

The objective of the research described in this paper is *to develop a framework for formally specifying and reasoning about timing correctness properties of mixed-criticality systems*. The following paragraphs explain this objective in greater detail. In general, correctness in safety-critical systems can be considered from two perspectives: (i) (pre-run-time) verification, and (ii) (run-time) survivability.



© Alan Burns and Cliff B. Jones;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio; Article No. 14; pp. 14:1–14:23



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

14:2 An Approach to Formally Specifying the Behaviour of Mixed-Criticality Systems

Pre-run-time *verification* of a safety-critical system involves verifying, prior to deployment, that the run-time behaviour of the system will be consistent with expectations. Verification assumptions are made regarding the kinds of circumstances that will be encountered by the system during run-time and guarantees are used to specify the required runtime behaviour of the system (provided that the assumptions hold).

In contrast, *survivability* addresses expectations of system behaviour in the event that the assumptions fail to hold fully (in which case a fault or error is said to have occurred during run-time). Survivability may further be considered to comprise two notions: robustness and resilience [14]. Informally, the robustness of a system is a measure of the degree of fault it can tolerate without compromising the quality of service it offers; resilience refers to the degree of fault for which it can provide a degraded, yet acceptable, criticality-aware quality of service.

The contribution of this paper is to develop a framework for the formal specification of MCS; we define a formal approach that:

- Demonstrates that the Rely/Guarantee approach (see Section 2) can be extended to cover temporal properties (see Section 3) of concurrent systems (in addition to their functionality).
- Precisely specifies the required behaviour of a run-time scheduler (in normal and degraded modes of operation).
- Enables proofs to be developed and discharged that employ the contract(s) between the jobs and tasks comprising an application, and the scheduler.
- Enables, with additional specifications of the functional elements of the scheduler, the code of the scheduler to be produced as a refinement of these specifications.
- Enables the scheduler to be replaced or modified by verifying that a new version satisfies the original specification.
- Identifies the assumptions that the analysis (scheduling and execution time) makes such that the result of the analysis confirms that the system will meet its timing requirements.
- Enables the many approaches to resilience and robustness to be compared – this requires the formal framework to be sufficiently expressive to capture the semantics of the various schemes that have been proposed.

This initial description of our approach focusses on the specification aspects; future work will address verification. We do however demonstrate where proof can be used to ensure that, whenever a degraded mode must be entered, its prerequisites are ensured by the guaranteed conditions of the mode that has just been abandoned. We also make explicit the proof obligations on the offline scheduling analysis that must be applied to the application prior to deployment.

We explain the elements of the framework via a series of related, increasingly challenging, examples. The initial examples are sufficiently straightforward that, arguably, a full formal specification is not required; however the later examples do show the value of precise specifications. The examples illustrate the approach with at most two criticality levels, this helps to explain the framework, but again the full value of a formal approach comes when the system has increased complexity as happens when there are three or more criticality levels.

In this paper an MCS is assumed to consist of a finite set of jobs/tasks and a single specific Scheduler. Rely and guarantee conditions capture the run-time relationship between the Scheduler and the jobs/tasks, yielding a specification of the necessary behaviours/properties of the Scheduler. Note that this process does not delve into the internal structure of the Scheduler: the scheduling-theoretic issues of how it meets its specification (if indeed it can) is *not* the focus of this work. Rather, in this paper we are only seeking to provide a clear and

intuitive explanation of the formalism. The history of formal methods (such as Hoare Logic) leads us to believe that methods can be developed for showing that specific MC-scheduling algorithms can satisfy (or not) the proof obligations that arise from the Rely/Guarantee (R/G) specifications. Related work in this area includes PROSA which addresses mechanised verification of results from scheduling analysis [21, 10]. (Mechanisation of R/G reasoning is on-going [29, 22]).

Organisation. The paper is organised as follows. After an introduction to R/G conditions (Section 2), the basic properties of the proposed framework are developed in Section 3 via a focus on *jobs* – this allows the approach to be motivated and explained. Mixed-criticality jobs are then covered in Section 4 including the introduction of fault-tolerance via *modes* of operation each with their own R/G conditions. Extensions of the same ideas to *tasks* are then given in Sections 5 and 6. Conclusions are in Section 7.

2 Introduction to Rely/Guarantee conditions

Hoare’s “Axiomatic Approach” provides the basis of a development method for sequential programs. Although [32] employed post conditions of single states, subsequent development methods such as VDM [39], B [1] and Event-B [2] use relational post conditions that define acceptable final states with respect to their initial values. Crucially, there is a relatively obvious notion of compositionality for sequential programs where a specification can be replaced by anything that satisfies its pre/post condition specification.

Finding compositional development methods for the development of concurrent programs proved to be difficult precisely because of the “interference” that comes with (shared-variable) concurrency. One approach is to record and reason about interference using rely and guarantee conditions [37, 38] (a more algebraic presentation of the ideas is covered in [31]). The details and proof obligations of the R/G approach are not the main issue in the current paper. The basic idea is straightforward: just as pre conditions define a subset of possible starting states on which a component is expected to operate, *rely conditions* record interference that the specified component must tolerate; and, just as post conditions abstract from algorithms to achieve the transition from initial to final state, *guarantee conditions* are relations that define the maximum interference that the component may inflict on its environment. It is important to remember that pre and rely conditions are assumptions that a developer is invited to make; in contrast, guarantee and post conditions are obligations on the code to be created. A guarantee condition needs to be satisfied (only) as long as the corresponding rely condition is respected. Stating this negatively, if the environment makes a transition that does not satisfy the rely condition, the developed code is free from further obligations.

The R/G idea targeted the design of concurrent programs where the R/G conditions provide a way of decomposing designs. Papers such as [30, 42, 19] show that the R/G idea can be used to tackle the design of fault-tolerant CPS by using rely conditions to describe assumptions about physical system components. Where the physical components exhibit continuous change, the rely conditions record assumptions about the rate of such changes. This work also showed how layered R/G conditions can assist in addressing fault tolerance; *resilience* is represented by hierarchically related R/Gs – strong rely conditions address full functionality, weaker rely conditions are matched with lesser guarantees (perhaps only the safety-critical aspects), even weaker rely conditions might only guarantee safe fail-stop behaviour. *These properties of related R/G conditions are central to the framework developed in this paper.*

3 Job-based system model

This section focuses on a system comprising a set of jobs, \mathcal{J} , that are managed by a Scheduler (denoted by the symbol S). A representative job, $j \in \mathcal{J}$, has a relative deadline of D_j , arrives (and is released for execution) at time a_j and thus has an absolute deadline at time $d_j = a_j + D_j$. Let f_j denote the time at which it completes (finishes) its execution.¹ The set $act(\mathcal{J}, t)$ is the subset of \mathcal{J} containing the jobs that are active at time t , i.e.

$$j \in act(\mathcal{J}, t) \Leftrightarrow j \in \mathcal{J} \wedge (a_j \leq t < f_j)$$

A job that is immediately terminated on arrival (as required in specific circumstances by some MCS protocols) has $f_j = a_j$; it is deemed never to be active and to have missed its deadline.

We assume a discrete time model in which all job parameters are given as non-negative rational numbers with arbitrary precision. Time is an external physical phenomenon: the Scheduler has no control over the passage of time.

The specification of each job, j , consists of its pre-condition, P_j , post-condition Q_j , rely condition R_j and guarantee condition G_j . In this paper each of these conditions is expressed as a predicate over the system state. For an actual system these conditions will capture both the functional and timing behaviour of the job; here we focus only on the temporal properties. This requires that system states are indexed by time² and that the rely and guarantee conditions directly reference time. We write $R_S(t)/G_S(t)$ for the Scheduler and $R_j(t)/G_j(t)$ for jobs.

Properties that should remain true as time progresses are normally classed as *invariants* but here are represented as rely or guarantee conditions. This is because the jobs (and Scheduler) must take action in order to maintain correct behaviour – a job will miss its deadline if it is not scheduled appropriately.

The primary concern for each job is its execution time; and hence we define, for each job j , $e_j(t)$ which is the amount of execution time the job has consumed up to time t . There are obvious properties (axioms) for e :

$$\forall j \in \mathcal{J}, t \bullet e_j(t) \leq WCET_j \tag{1}$$

where WCET is the worst-case execution time of the job;

$$\forall j \in \mathcal{J}, t_1, t_2, t_1 < t_2 \bullet e_j(t_2) - e_j(t_1) \leq t_2 - t_1 \tag{2}$$

no job can execute faster than “real time”;

$$\forall j \in \mathcal{J}, t_1, t_2, t_1 < t_2 \bullet e_j(t_1) \leq e_j(t_2) \tag{3}$$

a job cannot “lose” execution time; and

$$\forall j \in \mathcal{J} \bullet \left(\forall t \leq a_j \bullet e_j(t) = 0 \wedge \forall t \geq f_j \bullet e_j(t) = e_j(f) \right) \tag{4}$$

a job cannot execute before it arrives or after it has finished.

¹ A job that is yet to finish has $f = \infty$; a job that is permanently suspended but never terminated retains this value.

² A slightly different approach to handling the progress of time was taken in [40]. In that paper a distinction is made between an abstract notion of *Time* and the *ClockValues* stored in a computer.

In this section the scheduler is deemed to exist for the entire life-time of the system, it is therefore specified by a single rely condition $R_S(t)$ and a single guarantee condition $G_S(t)$.

The following derivations first illustrate the basic approach with a set of single criticality jobs. Note that the role of the formal framework is to represent precisely the relationship between the Scheduler and the client jobs in a range of degraded and partial behaviours. It is not a model of a particular scheduler's run-time behaviour; rather it is a specification of the required properties of any scheduler (and its schedulability test) that is being proposed for the particular problem under investigation.

A key feature of mixed-criticality models is that they allow a system to degrade gracefully when faults occur. This leads to the Scheduler's run-time behaviour having different modes of operation. In each mode, different R and G conditions for the jobs and scheduler are defined, as is the transition between R/G contracts.

We start by considering a finite set of jobs that each have the same criticality; there is no degraded behaviour and hence only a single mode of operation. A job j is characterised by its Worst-Case Execution Time, $WCET_j$ (this is a value that will not be known with certainty) and C_j an estimate of $WCET_j$. The timely execution of a job *relies* on this estimate of $WCET$ being valid, and the Scheduler can only meet its obligations with a *reliance* of each job executing for no more than C_j . If these rely conditions hold, a *valid* Scheduler *guarantees* to manage the processing capacity so as to ensure that all jobs complete by their deadlines regardless of when the jobs arrive; each job *guarantees* to execute, when active, for no more than C_j .

Note that the value C_j plays a number of roles: the job relies on its environment behaving according to whatever model or measuring process was used to derive C_j , but the job also has a contract with the scheduler not to execute for more than C_j . The scheduler is assumed to have used some form of analysis to verify (offline usually) that, if all jobs respect their guarantee conditions, then it will be able to provide the necessary capacity to each job. Hence the job can rely upon being allowed to execute for up to C_j before its deadline.

With all four axioms ((1)-(4) above) in force, the rely and guarantee conditions of any valid Scheduler are as follows:

$$R_S(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet e_j(t) \leq C_j$$

$$G_S(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet t + (C_j - e_j(t)) \leq d_j$$

The Scheduler relies on all jobs executing within their estimated $WCET$ and guarantees to provide sufficient resource, following a defined policy, to ensure that each job always has sufficient space to complete before its deadline (i.e. that $t + (C_j - e_j(t)) \leq d_j$).³ The Scheduler's guarantee is an *obligation* that must be achieved by its code – i.e. the Scheduler's offline schedulability test must ensure this property. The conditions $R_S(t)$ and $G_S(t)$ are defined to refer only to jobs that are active at time t .

In order to satisfy G_S , the Scheduler must manage the dispatching of jobs in an appropriate manner. If necessary it will allocate to each job up to C_j execution time. It follows that if $WCET_j \leq C_j$ then each job will terminate by its deadline (i.e. $f_j \leq d_j$).

The R and G conditions of each active job are therefore:

$$R_j(t) \stackrel{\text{def}}{=} WCET_j \leq C_j \wedge t + (C_j - e_j(t)) \leq d_j$$

$$G_j(t) \stackrel{\text{def}}{=} e_j(t) \leq C_j$$

³ An alternative formulation [12] to the one presented here is for the Scheduler to guarantee a budget (of at least C for each job), and for each job to rely on this budget. Example specifications and further investigations indicated that the method defined in the current paper is the more realistic and effective.

At run-time, the job does not need to be aware of its deadline or current execution time; although more expressive and flexible behaviours may require this. Once a job (j) terminates the R_j and G_j conditions no longer apply.

The constraints imposed upon execution time are represented as guarantees and not post-conditions for a number of reasons:

1. post-conditions are, by definition, required to hold upon termination, but a failure may lead to the job not terminating;
2. to add fault tolerance (i.e. to cope with jobs whose estimated execution times are not respected) we will need to know the point in time at which a rely condition fails to hold (and hence a guarantee condition no longer has to hold); and
3. deadlines may change (or be removed) during the execution of the job (see later examples).

The semantics of rely/guarantee conditions is that guarantees are required to be met as long as the rely conditions are satisfied. If a job overruns and breaks its guarantee that $e_j(t) \leq C_j$ there must be a rely condition “at fault”. For this reason, we explicitly include $\text{WCET}_j \leq C_j$ in the rely condition: in an environment where this assumption does not hold, a job is not obliged to guarantee its temporal properties.

If the environment (hardware platform including the influence of concurrently executing jobs, preemption effects on cache etc.) behaves such that the WCET estimate of some job k is exceeded, then this job may execute for more than C_k , thus breaking its guarantee condition. As a consequence the rely condition for the Scheduler would not be satisfied and hence it would be under no obligation to provide the necessary capacity to every job – some jobs may still be active at their deadlines. This takes us to the topic of survivability and how MCS supports graceful degradation.

4 Mixed-criticality jobs

To illustrate how a level of resilience can be added, two criticality levels are considered: *HI*-crit and *LO*-crit; with \mathcal{J}_L a set of *LO*-crit jobs, \mathcal{J}_H a set of *HI*-crit jobs, and $\mathcal{J} = \mathcal{J}_L \cup \mathcal{J}_H$. Job h is a representative *HI*-crit job; l is a representative *LO*-crit job; j continues to represent any job. So, for example, $R_h(t)$ is the rely condition for any *HI*-crit job, $h \in \mathcal{J}_H$. With Mixed-Criticality jobs there are two estimates of C_j : $C_j(L)$ and $C_j(H)$; with $C_j(L) \leq C_j(H)$ [61].

It is initially assumed that the system is either in the Normal mode, in which case all jobs should meet their deadlines, or in the *HI*-crit mode in which only the *HI*-crit jobs are guaranteed to meet their deadlines. For the Normal (N) mode the (R , G) conditions are as above except that $C_j(L)$ replaces C_j in R_j, G_j, R_S and G_S :

$$R_S^N(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet e_j(t) \leq C_j(L)$$

$$G_S^N(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet t + (C_j(L) - e_j(t)) \leq d_j$$

$$R_j^N(t) \stackrel{\text{def}}{=} \text{WCET}_j \leq C_j(L) \wedge t + (C_j(L) - e_j(t)) \leq d_j$$

$$G_j^N(t) \stackrel{\text{def}}{=} e_j(t) \leq C_j(L)$$

The rely and guarantee conditions for the N mode are therefore:

$$R^N(t) = R_S^N(t) \wedge \bigwedge_{j \in \mathcal{J}} R_j^N(t)$$

$$G^N(t) = G_S^N(t) \wedge \bigwedge_{j \in \mathcal{J}} G_j^N(t)$$

Most of these rely and guarantee conditions are mutually supportive in the sense that they “cancel out” when looking at the whole system. The only rely condition that depends on external compliance is:

$$\forall j \in \mathcal{J} \bullet \text{WCET}_j \leq C_j(L)$$

4.1 Adding resilience to *HI*-crit jobs

Considering *HI*-crit jobs ($h \in \mathcal{J}_{\mathcal{H}}$) and their rely condition:

$$R_h^N(t) \stackrel{\text{def}}{=} \text{WCET}_h \leq C_h(L) \wedge t + (C_h(L) - e_h(t)) \leq d_h$$

We want to give a higher (safer) bound on WCET, so we consider a more conservative value ($C_h(H)$), where $C_h(H) > C_h(L)$. Now for all *HI*-crit jobs (h) we have a new *HI*-crit mode (H) and:

$$R_h^H(t) \stackrel{\text{def}}{=} \text{WCET}_h \leq C_h(H) \wedge t + (C_h(H) - e_h(t)) \leq d_h$$

$$G_h^H(t) \stackrel{\text{def}}{=} e_h(t) \leq C_h(H)$$

The Scheduler’s definition for mode H is

$$R_S^H(t) \stackrel{\text{def}}{=} \forall h \in \text{act}(\mathcal{J}_{\mathcal{H}}, t) \bullet e_h(t) \leq C_h(H) \wedge \forall l \in \text{act}(\mathcal{J}_{\mathcal{L}}, t) \bullet e_l(t) \leq C_l(L)$$

$$G_S^H(t) \stackrel{\text{def}}{=} \forall h \in \text{act}(\mathcal{J}_{\mathcal{H}}, t) \bullet t + (C_h(H) - e_h(t)) \leq d_h$$

In this *HI*-crit mode there is no obligation to provide any level of service to the lower criticality jobs or indeed to prevent these jobs from using resources (perhaps at a background priority in a priority-based scheduler). Hence:

$$R_l^H(t) \stackrel{\text{def}}{=} \text{WCET}_l \leq C_l(L)$$

$$G_l^H(t) \stackrel{\text{def}}{=} e_l(t) \leq C_l(L)$$

The above specification is, however, not sufficient for many of the protocols advocated for mixed-criticality scheduling. The standard “mixed-criticality” mechanism for being able to add more capacity to the *HI*-crit jobs is to take computation time away from the *LO*-crit jobs. Or, more precisely, to no longer execute these jobs. This further adds to the guarantees of the Scheduler.

To facilitate this functionality it is necessary to know the time at which R_S^N became false (i.e. when an active *HI*-crit job has first executed for $C(L)$ without terminating). We refer to this as mode N ’s *deviation time*, η^N ; defined by the following property:

$$\exists \eta^N, h \in \text{act}(\mathcal{J}_{\mathcal{H}}, \eta^N) \bullet e_h(\eta^N) \geq C_h(L) \wedge \forall t, t < \eta^N, g \in \text{act}(\mathcal{J}_{\mathcal{H}}, t) \bullet e_g(t) < C_g(L)$$

At the deviation time R_S^N becomes false, mode N is left and, simultaneously⁴, mode H is entered. The rely and guarantee conditions $R^H(t)$ and $G^H(t)$ apply for $t \geq \eta^N$.

⁴ The notion of simultaneous is taken from the Timebands [18] framework that allows instantaneous actions to be defined at one time band (granularity) but implemented by an activity at a finer time band.

We assume here the extreme Vestal behaviour of not executing LO -crit jobs again after η^N . This leads to a full specification for the guarantee condition for the Scheduler:

$$G_S^H(t) \stackrel{\text{def}}{=} \forall h \in \text{act}(\mathcal{J}_H, t) \bullet t + (C_h(H) - e_h(t)) \leq d_h \wedge \forall l \in \text{act}(\mathcal{J}_L, t) \bullet e_l(t) = e_l(\eta^N)$$

with a simplified rely condition as the Scheduler no longer relies on the behaviour of LO -crit jobs as it guarantees that they do not execute:

$$R_S^H(t) \stackrel{\text{def}}{=} \forall h \in \text{act}(\mathcal{J}_H, t) \bullet e_h(t) \leq C_h(H)$$

and therefore:

$$R^H(t) = R_S^H(t) \wedge \bigwedge_{l \in \mathcal{J}_L} R_l^H(t) \bigwedge_{h \in \mathcal{J}_H} R_h^H(t)$$

$$G^H(t) = G_S^H(t) \wedge \bigwedge_{l \in \mathcal{J}_L} G_l^H(t) \bigwedge_{h \in \mathcal{J}_H} G_h^H(t)$$

This strategy of pausing all LO -crit jobs is not an option that the Scheduler *could* choose, but a requirement that is part of the specification of the job's behaviour – and hence must be explicitly contained in G_S^H .

With this specification the LO -crit jobs are suspended; but they may execute later in another mode (perhaps after their deadlines). To abort these and future LO -crit jobs, rather than preempt them indefinitely, the Scheduler could (if specified to do so) enforce termination:

$$\forall t, t > \eta^N \bullet \text{act}(\mathcal{J}_L, t) = \emptyset$$

4.2 Transitioning from mode N to mode H

The specification above requires a movement from mode N to mode H . To provide useful fault tolerance, it must be true that, whenever the rely condition for N fails to be satisfied, the corresponding rely condition for H is satisfied (and remains so) i.e. at time η^N when $R^N(\eta^N)$ no longer pertains: $R^H(\eta^N)$ is satisfied. If $R^H(\eta^N)$ is true then the guarantee condition, $G^H(t)$, is delivered for all $t > \eta^N$, and as a consequence $R^H(t)$ must hold.

In general a mode change could involve modes with unrelated functionality and hence the truth of the rely condition in the new mode would need to be asserted independently of the rely condition in the old mode. This is identical to what is required at system startup where the rely condition of the initial mode must be established. In this work, however, we require a more constrained relationship between the modes:

► **Definition 1.** *Mode B is a weakened form of mode A if*

1. *for all times (t) before η^A when $R^A(t)$ is true then $R^B(t)$ is true (i.e. $R^A(t) \Rightarrow R^B(t)$); and*
 2. *at time η^A when some aspect of $R^A(\eta^A)$ is no longer true $R^B(\eta^A)$ remains true.*
- As $R^B(\eta^A)$ is true, it followed that $G^B(t)$ is true for all $t > \eta^A$.

Counter Example. We require that mode H is a weakening of mode N . Consider the first element of the definition of weakening: in two of the three rely conditions, this is indeed the case as:

$$R_S^N(t) \Rightarrow R_S^H(t); \quad R_l^N(t) \Rightarrow R_l^H(t)$$

but $R_h^N(t)$ does not have a simple relationship to $R_h^H(t)$. The first conjunct is a weakening of the “external” rely condition as $\text{WCET}_h \leq C_h(L) \Rightarrow \text{WCET}_h \leq C_h(H)$. The second conjunct is, however, a strengthening; hence modes N and H do not have the required hierarchical relationship – H is not a weakened form of N .

A *Modified Definition of Mode N* (N^*). In order to assert that mode H is a weakened form of the initial mode it is necessary to constrain the behaviour of the Scheduler further in the Normal mode. It must do more than simply guarantee to provide for all jobs $C(L)$ before the deadline d , it must also reserve sufficient slack so that, at any time a switch can be made, it is possible to guarantee $C(H)$ before d .

It follows that, for a HI -crit jobs, h , to be schedulable in both N^* and H modes, there exists a virtual deadline v_h with

$$v_h \leq d_h - (C_h(H) - C_h(L))$$

that is defined (and confirmed) by the applicable scheduling analysis, such that: if the Scheduler in mode N^* guarantees $C(L)$ by v , then the Scheduler in mode H will be able to guarantee $C(H)$ by d .⁵ To accommodate this constraint the guarantee condition of the Scheduler in mode N^* must be modified to:

$$G_S^{N^*}(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet t + (C_j(L) - e_j(t)) \leq v_j$$

and the Rely conditions of HI -crit jobs becomes

$$R_h^{N^*}(t) \stackrel{\text{def}}{=} \text{WCET}_h \leq C_h(L) \wedge t + (C_h(L) - e_h(t)) \leq v_h$$

For LO -crit jobs (l) $v_l = d_l$ and hence $G_S^{N^*}$ has not changed for these jobs. For HI -crit jobs (h) there is a proof obligation on the scheduling analysis to demonstrate:

$$\forall t, h \in \text{act}(\mathcal{J}_H, t) \bullet G_S^{N^*}(t) \Rightarrow t + (C_h(H) - e_h(t)) \leq d_h \quad (5)$$

Such an obligation could be verified using mechanised proof tools such as PROSA [21, 10].

► **Lemma 2.** *Mode H is a weakening of mode N^* .*

Proof. As noted above $\forall t : R_S^N(t) \Rightarrow R_S^H(t)$ and $R_l^N(t) \Rightarrow R_l^H(t)$. The modification to N^* does not effect these rely conditions. Also $\text{WCET}_h \leq C_h(L) \Rightarrow \text{WCET}_h \leq C_h(H)$ (as $C_h(H) \geq C_h(L)$). Finally $t + (C_h(L) - e_h(t)) \leq v_h \Rightarrow t + (C_h(H) - e_h(t)) \leq d_h$ as $v_h \leq d_h - (C_h(H) - C_h(L))$.

The second step is to show that, at time η^{N^*} (when $R^{N^*}(\eta^{N^*})$ fails), $R^H(\eta^{N^*})$ remains true. Condition $R^{N^*}(\eta^{N^*})$ is false because the $WCET$, for some HI -crit job k , is not bounded by $C_k(L)$. Moreover η^{N^*} is the first time instant at which R^{N^*} is false. Hence at time η^{N^*} , $R_k^{N^*}(\eta^{N^*})$ is false, but $R_k^H(\eta^{N^*})$ is true as $C_k(H) > C_k(L)$.⁶ ◀

This weakening property and the proof obligation represented by eqn (5) are therefore sufficient to ensure that, whenever the Normal mode must be abandoned, the HI -crit mode can be entered and will deliver its guaranteed behaviour. The final point to note about the transition from N^* to H is that the Guarantee conditions are also weakened. The system moves from guaranteeing all job deadlines to just guaranteeing the HI -crit ones. Hence $G^{N^*}(t) \Rightarrow G^H(t)$.

⁵ This virtual deadline is used directly in the EDF-based scheduling scheme EDF-VD [5] and in fixed-priority scheduling is equivalent to the worst-case (maximum) computed response time of the HI -crit job in the Normal mode [6]. Note whatever scheduling protocol is employed at run-time there is an implicit (if not explicit) virtual deadline in the Normal mode. If this were not the case then there would be insufficient spare capacity in the Normal mode to satisfy the extra demand of the HI -crit mode.

⁶ Strictly, we require $C_k(H) > C_k(L) + \delta$ where δ is the minimum time step that the system can undertake in its discrete model of time.

4.3 Postponing the deviation time

As noted in the introduction, the main focus of this paper is to motivate and define a formal framework for the specification of mixed criticality systems. In this section we are able to give an example of how this framework can be utilised.

A system is considered to degrade at deviation time η^{N^*} which is defined, above, as the first time that a *HI*-crit job executes beyond its $C(L)$ constraint. But if this deviation time could be postponed then the dynamics of the system may alleviate the need to make the mode change – the *LO*-crit jobs could continue to meet their deadlines. Possible favourable dynamic behaviours include sporadic jobs not arriving at their maximum rate, and other jobs executing for less than their maximum $C(L)$ bound. To explore the possibility of delaying the deviation time consider again the specification of the N^* mode:

$$R_S^{N^*}(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet e_j(t) \leq C_j(L)$$

$$G_S^{N^*}(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet t + (C_j(L) - e_j(t)) \leq v_j$$

$$R_j^{N^*}(t) \stackrel{\text{def}}{=} \text{WCET}_j \leq C_j(L) \wedge t + (C_j(L) - e_j(t)) \leq v_j$$

$$G_j^{N^*}(t) \stackrel{\text{def}}{=} e_j(t) \leq C_j(L)$$

where $v_j = d_j$ for *LO*-crit jobs and $v_l \leq d_l - (C_l(H) - C_l(L))$ for *HI*-crit jobs.

If all jobs behave according to this R/G specification then all virtual deadlines will be met. This implies there is a weakened form of behaviour (which we denote as mode \hat{N}^*):

$$R_S^{\hat{N}^*}(t) \stackrel{\text{def}}{=} \forall j \in \text{act}(\mathcal{J}, t) \bullet t \leq v_j$$

$$G_j^{\hat{N}^*}(t) \stackrel{\text{def}}{=} t \leq v_j$$

with $G_S^{\hat{N}^*} = G_S^{N^*}$ and $R_j^{\hat{N}^*} = R_j^{N^*}$.

From the definition of the virtual deadline we have $R_S^{N^*} \Rightarrow \hat{R}_S^{N^*}$ and $G_j^{N^*} \Rightarrow \hat{G}_j^{N^*}$.

The deviation time (when $\hat{R}_S^{N^*}$ becomes false for the first time) is now when a *HI*-crit job is still executing at its virtual deadline. And this time is likely to be significantly later than that provided by the earlier definition. Note also that this alternative definition of the deviation time for the normal mode changes what needs to be monitored – from execution time to elapsed time. This is likely to reduce the runtime overheads of the MCS scheduler.

Again it is straightforward to prove that mode *H* is a weakening of (the modified) mode \hat{N}^* , and the proof obligation on the offline scheduling analysis (eqn (5)) must again be used to validate the v values assigned to each *HI*-crit job. Recent scheduling results [8] have demonstrated that for fixed priority-based scheduling and AMC-rtb analysis the same v values are valid for the original definition of deviation time and the one derived in this section. That paper also demonstrated the benefits in terms of run-time performance that is gained from postponing the mode change.

The proposed framework allowed this new protocol to be easily defined and verified. Further properties can be proven (such as the above definition of deviation time being the latest possible). In this introductory paper, however, priority is given to extending the framework to task-based systems.

5 Task-based system model

The above treatment of mixed-criticality jobs has demonstrated that the proposed specification framework has sufficient expressive power to capture the properties commonly required of job-based systems. The scheduling literature typically describes jobs as being organised within tasks – in this section we extend the study to cope with tasks.

A real-time system is deemed to consist of a set of tasks. A single execution of the code of a task is a job. So a task gives rise to a sequence of jobs. The scheduler determines the order in which jobs from different tasks are executed. With a task-based model there is an assumption that the duration of the system is unbounded. This means that any specification framework must cater for the return of the system from any degraded mode back to the initial mode for the system (and to allow these mode changes to occur numerous times). We assume that each task k delivers a potentially unbounded sequence of jobs, k^1, k^2 etc, with job k^m having arrival time a_k^m and completion time f_k^m . This sequence is not “reset” as new modes are entered; it continues to extend indefinitely.

This treatment focuses on issues related to execution time and mixed criticality. It does not directly address the rely and guarantee conditions related to when and how a task is released for execution. For example, time-triggered tasks require their job releases to be guaranteed by some Dispatcher; and event-triggered tasks rely on their releasing events obeying some minimum separation requirement. These issues are covered here by each task guaranteeing that its jobs do not arrive too early – a rely condition for the Scheduler.

The system is again assumed to be defined over two criticality levels, *LO*-crit and *HI*-crit, and to have two modes of behaviour: N^* and H . We however drop, for ease of presentation, the superscript from N in the following. To define a general model, each of the defining temporal parameters of each task (D, T, C, V) has an L and a H value.

We again make use of sets: \mathcal{T} is the set of all tasks, \mathcal{T}_L the set of *LO*-crit tasks, and \mathcal{T}_H the set of *HI*-crit tasks, and $\mathcal{T} = \mathcal{T}_L \cup \mathcal{T}_H$. The axioms defined in Section 3 still apply.

At any time t , each task k has a single *current job*. We let $c(t)$ be the index of this job (for ease of presentation, we just use c for this index as the t value is always implied). Hence the current job of task k is denoted by k^c . This job may have finished, but the next job of this task has not yet arrived ($f_k^c \leq t < a_k^{c+1}$). In task models that allow a job to arrive before the previous job of the same task has finished (i.e. tasks with $D > T$), the “current” job is the one that arrived first.

We modify the definition of “active” to cater for tasks; a task is active if its current job has not yet terminated:

$$k \in act(\mathcal{T}, t) \Leftrightarrow k \in \mathcal{T} \wedge (a_k^c \leq t < f_k^c)$$

In each of the criticality modes the relative parameters (V_k and D_k) are added to the arrival time a_k^c to give the absolute values: v_k^c , $d_k^c(L)$ and $d_k^c(H)$.

5.1 Vestal-inspired example

This section specifies the required behaviour of the system (Scheduler and tasks) for a typical model inspired by the Vestal approach [61]. The properties of this model are, briefly:

- System starts in the mode N in which all jobs of all tasks execute for no more than $C(L)$ and all job deadlines are met.
- All *LO*-crit tasks are assumed (or constrained) to execute for no more than $C(L)$.
- All *HI*-crit tasks are assumed (or constrained) to execute for no more than $C(H)$.
- If any, or indeed all, *HI*-crit tasks execute for more than $C(L)$ then:

14:12 An Approach to Formally Specifying the Behaviour of Mixed-Criticality Systems

- all *HI*-crit tasks must still meet their deadlines;
- all *LO*-crit tasks have their periods and deadlines increased, but must still meet their deadlines.
- If there is an idle instant then the system must return to the Normal mode of operation.

This extension of the Vestal model is often referred to as the *elastic task model* [20] in which the periods and deadlines of *LO*-crit tasks are extended from $T_l(L)$ (and $D_l(L)$) to $T_l(H)$ (and $D_l(H)$), but are still guaranteed.

The major difference when moving from jobs to tasks is that each task, like the Scheduler, exists for the full duration of the time spent in each mode. Although individual jobs terminate, the task does not (in the model being utilised here). So $R_k(t)$ and $G_k(t)$ are the rely and guarantee conditions of task k , but they refer to the job that is current (and possibly active) at time t .

For the Vestal-inspired model outlined above we have, for all *LO*-crit tasks, $l \in \mathcal{T}_L$, $C_l(L) = C_l(H)$, $T_l(H) > T_l(L)$, $D_l(H) > D_l(L)$ and $V_l = D_l(L)$ and for all *HI*-crit tasks, $h \in \mathcal{T}_H$, $C_h(L) < C_h(H)$, $T_h(L) = T_h(H)$, $D_h(L) = D_h(H)$ and $V_h < D_h(L) - (C_h(H) - C_H(L))$.

The conditions for the normal mode N are:

$$R_S^N(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet e_k^c(t) \leq C_k(L) \wedge (c > 1 \Rightarrow a_k^c - a_k^{c-1} \geq T_k(L))$$

$$G_S^N(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet t + (C_k(L) - e_k^c(t)) \leq v_k^c$$

$$R_k^N(t) \stackrel{\text{def}}{=} \text{WCET}_k \leq C_k(L) \wedge k \in \text{act}(\mathcal{T}, t) \Rightarrow t + (C_k(L) - e_k^c(t)) \leq v_k^c(L)$$

$$G_k^N(t) \stackrel{\text{def}}{=} e_k^c(t) \leq C_k(L) \wedge (c > 1 \Rightarrow a_k^c - a_k^{c-1} \geq T_k(L))$$

R_S^N contains the separation condition: if the current job is not the first instantiation of the task then it must arrive at least $T_k(L)$ after the previous job.

In the *HI*-crit mode, H , we have a similar formulation but with different parameters:

$$R_S^H(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet e_k^c(t) \leq C_k(H) \wedge (c > 1 \Rightarrow a_k^c - a_k^{c-1} \geq T_k(H))$$

$$G_S^H(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet t + (C_k(H) - e_k^c(t)) \leq d_k^c(H)$$

$$R_k^H(t) \stackrel{\text{def}}{=} \text{WCET}_k \leq C_k(H) \wedge k \in \text{act}(\mathcal{T}, t) \Rightarrow t + (C_k(H) - e_k^c(t)) \leq d_k^c(H)$$

$$G_k^H(t) \stackrel{\text{def}}{=} e_k^c(t) \leq C_k(H) \wedge (c > 1 \Rightarrow a_k^c - a_k^{c-1} \geq T_k(H))$$

These two formulations can easily be combined into a single specification that is a function of the mode (N or H) but are separated here to improve readability.

5.2 Transitioning from N to H

In this and the following section we consider the movement between modes; from Normal, N , to the *HI*-crit mode, H , and then the return to the Normal mode. In a long-lived task-based system there may be many such transitions between N and H . Each time a mode is entered, we consider this to be a new *occurrence* of the mode and therefore there is a new *occurrence* of the Scheduler for that mode. A move from N to H involves one occurrence of the N -mode Scheduler terminating and, instantaneously, a new occurrence of the H -mode

Scheduler starting its execution⁷. A natural linkage between Scheduler occurrences is for the post-condition of one mode, say $A (Q_S^A)$, to ensure the pre-condition of the follow-on mode, $B (P_S^B)$, with $Q_S^A \Rightarrow P_S^B$.

We note that the two mode changes contained within this task-based two-level mixed criticality system are of a quite different nature. The movement from N to H is forced, as N must be left. But the transition from H back to N is one of preference – both modes are acceptable, but the functional behaviour of the system is enhanced by being in the N mode.

In Section 4.2 we noted that as mode N is left at time η^N , due to $R^N(\eta^N)$ being false, we must prove that $R^H(\eta^N)$ is true. This involves two steps. First, at any time $t < \eta^N$, $R^N(t) \Rightarrow R^H(t)$. Second, at time η^N , when $R^N(\eta^N)$ is broken, $R^H(\eta^N)$ remains true.

Following the approach in Section 4.2, the task model has again made use of a virtual deadline for HI -crit jobs; from this we derive the proof obligation:

$$\forall t, h \in \text{act}(\mathcal{T}_H, t) \bullet G_S^N(t) \Rightarrow t + (C_h(H) - e_h^c(t)) \leq d_h^c(H) \quad (6)$$

Counter Example. With this Vestal-inspired example, the periods of the LO -crit tasks are expanded when the H mode is entered. It is therefore **not** true that $a_l^c - a_l^{c-1} \geq T_l(L) \Rightarrow a_l^c - a_l^{c-1} \geq T_l(H)$ as $T_l(H) > T_l(L)$. Hence R_S^N does **not** imply R_S^H .

A Modified Definition of Mode $H (H^)$.* We must again modify the specification. However on this occasion rather than strengthen the rely condition in mode N we weaken the definition of the rely condition for the Scheduler in the HI -crit mode:

$$R_S^{H^*}(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet e_k^c(t) \leq C_k(H) \wedge (c > 1 \wedge t > \eta^N \Rightarrow a_k^c - a_k^{c-1} \geq T_k(H))$$

Note the addition of $t > \eta^N$, the constraint on the arrival times of jobs in the new mode only applies strictly *after* η^N . The Guarantee condition of mode H^* is unchanged ($G^{H^*}(t) = G^H(t)$) and for the tasks: $R_k^{H^*}(t) = R_k^H(t)$, and $G_k^{H^*}(t) = G_k^H(t)$.

► **Lemma 3.** *Mode H^* is a weakening of mode N .*

Proof. First, $\forall t < \eta^N$: For LO -crit tasks: $C_l(H) = C_l(L)$ and $v_l^c = d_l^c$ hence $R_l^N = R_l^H$ (so $R_l^N \Rightarrow R_l^H$). For HI -crit tasks: $\text{WCET}_h \leq C_h(L) \Rightarrow \text{WCET}_h \leq C_h(H)$ (as $C_h(H) \geq C_h(L)$); and $t + (C_h(L) - e_h^c(t)) \leq v_h^c \Rightarrow t + (C_h(H) - e_h^c(t)) \leq d_h^c$ as $v_h^c \leq d_h^c - (C_h(H) - C_h(L))$. Hence $R_h^N \Rightarrow R_h^{H^*}$. For the Scheduler, the first conjunct is appropriate as $e_k^c(t) \leq C_k(L) \Rightarrow e_k^c(t) \leq C_k(H)$, the second conjunct does not apply as $t < \eta^N$.

The second step (showing R^{H^*} is true at time η^N) follows the proof of Lemma 2; noting again that the second conjunct of $R_S^{H^*}(t)$ does not apply when $t = \eta^N$. ◀

As $R^{H^*}(\eta^N)$ is true, it follows that $G^{H^*}(t)$ is true for all $t > \eta^N$ and hence $R^{H^*}(t)$ is true for all $t \geq \eta^N$ as long as all task execution times are bounded by $C_k(H)$.

The proof obligations on the necessary scheduling analysis must allow for all LO -crit generated jobs to arrive at the time of the mode change. One of the advantages of this more formal specification of the Scheduler's behaviour is that it helps identify this constraint explicitly. We note that many examples of published scheduling algorithms for mixed-criticality systems (for example [15]) do allow LO -crit jobs to arrive (and subsequently execute) at the time of the mode change even if that would not be allowed in the new mode.

⁷ An implementation may utilise a single Scheduler that modifies its behaviour depending upon which mode is current. Nevertheless, from a modelling point of view we consider each occurrence of the Scheduler to be a distinct execution.

However this property is often hidden within the analysis (by the use of a “floor plus one” rather than a “ceiling” representation of job arrivals). Within our formal framework the property is explicit.

To summarise, in order to prove that R^H is true whenever a forced mode change can occur, we note three distinct situations:

1. Conjunctions within R^H are weakened forms of those in R^N and remain true.
2. Conjunctions in R^N must be strengthened so that they then imply the corresponding conjunctions in R^H .
3. Conjunctions in R^H must be weakened so that they are implied by the corresponding conjunctions in R^L .

The above example makes use of all three strategies.

5.3 Transitioning from H to N

As long as the execution times of the HI -crit tasks are bounded by their $C(H)$ estimates, the system will stay in the H mode. All the rely conditions will remain true. However it is desirable to return to the Normal mode if possible as this mode provides a better level of service – i.e. LO -crit tasks will be able to occur more often and have shorter deadlines.

Once the over-running HI -crit job that caused the transition to mode H has terminated, there is the possibility that all new jobs can be released with their LO -crit parameters and, if they all execute for no more than $C(L)$, all deadlines can be met. But we know that any scheduling scheme can only guarantee deadlines if there is bounded (indeed often zero) residual work in the system at the time the Normal mode is (re-)activated [7]. It is therefore scheduler specific as to when the system is “safe” to return to the Normal N mode of operation.

May/must constraints [19] are useful here. If the system is idle (there are no jobs to execute), it is usual to state that the scheduler *must* return the system to the Normal mode, but it *may* make this change earlier if a proof obligation has shown that such a transition is safe.

In terms of the framework presented in this paper a switch back to N mode is allowed only when the scheduling obligations (as represented by G_S^N) of that mode can be satisfied by the current Scheduler. If these obligations are satisfied, the move from H to N can be sanctioned by an appropriate pre-condition on the Normal mode. An example of one such pre-condition is the commonly used protocol that the Normal mode can only be (re-)entered at time t if there are no active jobs at time t (other than ones that arrive at time t):

$$P_S^N(t) \stackrel{\text{def}}{=} k \in \text{act}(\mathcal{T}, t) \Rightarrow a_k^c = t$$

The Scheduler for the Normal mode can therefore assume this property and it is the responsibility of the Scheduler in the HI -crit mode to enforce it whenever it invokes a mode change back to Normal. In other words this is a post-condition for the Scheduler in mode H :

$$Q_S^H(t) \stackrel{\text{def}}{=} k \in \text{act}(\mathcal{T}, t) \Rightarrow a_k^c = t$$

6 Robustness and resilience

Here we extend the treatment for tasks to show how we can more systematically specify levels of robustness and resilience for mixed-criticality systems, the motivation here being to develop a means of quantifying robustness and resilience. The first step in this process is to specify the various schemes being proposed.

Informal definitions of robustness and resilience are provided in [14] – i.e. the robustness of a system is a measure of the level of faults it can tolerate without compromising the quality of service it offers; resilience, by contrast, refers to the level of faults for which it can provide degraded yet acceptable (e.g. safe) quality of service. It is noted in [14] that there are a number of standard responses in the fault tolerance literature for systems that suffer transient faults (equating to one or more concurrent job failures in this work):

1. Fail (Fully) Operational – all tasks/jobs execute correctly (i.e. meet their deadlines).
2. Fail Robust – some tasks are allowed to skip a job but all non-skipped jobs execute correctly and complete by their deadlines; the quality of service at all criticality levels is unaffected by job skipping. Many periodic control tasks have this property [62]; there is sufficient inertia in the physical system to allow the occasional control signal to be missed.
3. Fail Resilient – some lower criticality tasks are given reduced service such as having their periods/deadlines extended, priorities dropped and/or their execution budgets reduced; if the budget is reduced to zero then this is equivalent to subsequent jobs of the task being abandoned.
4. Fail Safe/Restart – where the level of failure exceeds what Fail Resilient bounds can accommodate, more extreme responses are required including rebooting or system shut-down (if the application has a fail-safe state). If a fail-safe state cannot be achieved then the system may need to rely on best-effort tactics that have no guarantees. This is, of course, the last resort to achieving survivability.

6.1 Failure modes

The framework developed above has been extended to include a number of more complex behaviours that arise from supporting robust and resilient behaviour. In this section we briefly outline a set of possible failure modes.

Fail operational – FO. A *HI*-crit job experiences a fault if it executes for more than $C(L)$. One measure of Fail Operational is therefore the number of such job failures that can be accommodated while still meeting all task deadlines. However, if a job from a *HI*-crit task executes for more than $C(L)$, we still assume that the $C(H)$ bound remains operational.

One criticism of those models derived from Vestal [61] is that they usually assume that any overrun of $C(L)$ results in an execution time of $C(H)$. In practice this is very unlikely to occur, a minor overrun is more likely. We therefore introduce a parameter, C_O , that represents a unit of overrun (for all jobs). Fail Operational is a measure of how many such overruns can be accommodated. Let O denote this number over all the tasks. A *HI*-crit job that executes for more than $C_h(L)$ but less than $C_h(L) + C_O$ has an O value of 1. In general, a task has an O value of n if its overrun is between $(n - 1) * C_O$ and $n * C_O$.

The metric for Fail Operational is therefore the maximum O value allowed (F_O) in a defined interval, I_O . This interval could be of a fixed length (and would usually be much greater than the maximum task period). Alternatively it could be the interval from the current time back to when there was an idle moment, m , defined by:

$$\exists m, m < t \bullet \left(\forall k \in act(\mathcal{T}, m) \bullet a_k^c = m \right) \wedge \\ \forall n, m < n < t \bullet \left(\exists k \bullet k \in act(\mathcal{T}, n) \wedge a_k^c < n \right)$$

so the only active tasks at time m are those that released a job at that time, and there are active tasks that have not just been released for all times between m and t . Note m must exist as system startup (time 0) matches the definition of m as the only active tasks are those released at time 0. We note that m is a function of t , hence $m(t)$ in the following.

14:16 An Approach to Formally Specifying the Behaviour of Mixed-Criticality Systems

To compute O at time t , we need to know how many overruns each job has experienced. This can be computed as follows:

$$O = \sum_{\forall h \in \mathcal{T}_{\mathcal{H},s} \bullet t > a_h^s \geq m(t)} \left\lceil \frac{e_h^s(t) - C_h(L)}{C_O} \right\rceil_0$$

where $\lceil \cdot \rceil_0$ constrains the ceiling function to return a value no less than 0.

If this value is greater than 1 but no greater than F_O then the system mode should be Fail Operational (FO) with all tasks meeting their deadlines. It follows that the rely and guarantee conditions for the Scheduler are as follows. Remember that for LO -crit tasks $C(H) = C(L)$:

$$R_S^{FO}(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet e_k^c(t) \leq C_k(H) \wedge (a_k^c = t \wedge c > 1) \Rightarrow a_k^c - a_k^{c-1} \geq T_k(L) \wedge \sum_{\forall h \in \mathcal{T}_{\mathcal{H},s} \bullet t > a_h^s \geq m(t)} \left\lceil \frac{e_h^k(t) - C_h(L)}{C_O} \right\rceil_0 \leq F_O$$

$$G_S^{FO}(t) \stackrel{\text{def}}{=} \forall k \in \text{act}(\mathcal{T}, t) \bullet t + C_k(L) - e_k^c(t) \leq v_k^c \wedge \forall h \in \mathcal{T}_{\mathcal{H}} \bullet a_h^c \geq m(t) \wedge e_h^c(t) > C_h(L) \Rightarrow t + C_k(H) - e_k^c(t) \leq d_k^c(H)$$

As there are no overruns in the normal mode we can deduce that $R_S^N \Rightarrow R_S^{FO}$.

Note this formulation is structurally different from that given earlier for a pure Vestal-like model. What the Scheduler must rely on is a property of the whole set of HI -crit tasks, not a specific property of each individual task. The Scheduler can therefore guarantee $C_h(H)$ (by the task's deadline) to any HI -crit tasks that overrun. But this guarantee is subject to the rely condition remaining true (i.e. there is a bound on the number and extent of these overruns).

The specification of the HI - and LO -crit tasks in the normal mode, and for most tasks in the FO mode, is simply

$$R_k^{FO}(t) \stackrel{\text{def}}{=} \text{WCET}_k \leq C_k(L) \wedge k \in \text{act}(\mathcal{T}, t) \Rightarrow t + C_k(L) - e_k^c(t) \leq v_k^c$$

$$G_k^{FO}(t) \stackrel{\text{def}}{=} e_k^c(t) \leq C_k(L) \wedge (a_k^c = t \wedge c > 1) \Rightarrow a_k^c - a_k^{c-1} \geq T_k(L)$$

But for the tasks that overrun, they experience a mode change that moves the system to a variant of FO :

$$R_h^{FO^*}(t) \stackrel{\text{def}}{=} \text{WCET}_h \leq C_h(H) \wedge h \in \text{act}(\mathcal{T}_{\mathcal{H}}, t) \Rightarrow t + C_h(H) - e_h^c(t) \leq d_h^c(H)$$

$$G_h^{FO^*}(t) \stackrel{\text{def}}{=} e_h^c(t) \leq C_h(H) \wedge (a_h^c = t \wedge c > 1) \Rightarrow a_h^c - a_h^{c-1} \geq T_h(H)$$

For the non overrunning tasks and the Scheduler $R^{FO^*} = R^{FO}$, and $G^{FO^*} = G^{FO}$.

A small number of tasks experiencing this change will not cause the Scheduler to change mode, unless its rely condition is invalidated. The proof obligation (6) will again ensure that $R_h^{FO^*}$ is a weakening of R_h^N and R_h^{FO} .

In summary, a system stays in the normal mode until a single HI -crit task executes for more than $C(L)$. The system then moves to mode FO with the overrunning task behaving according to mode FO^* . Further HI -crit tasks may overrun and move to mode FO^* . Eventually either an idle instant occurs and the system will return to the normal mode N , or the FO count is breached and R_S^{FO} is invalidated. The system will now fail unless there is a further degraded mode it can transition to; such a mode is considered next.

Fail robust – FR. A *robust task* is one that can safely drop one non-started job in a defined time interval. Each task (be it *HI-crit* or *LO-crit*), as part of its definition, has a robustness parameter, w . If a task has successfully completed the execution of w consecutive jobs then the Scheduler can drop the next job (before it has been given any execution time). As such jobs should only be dropped if they have to be, this requires a new mode: *FR* (Fail Robust). This mode will only be entered if the rely condition of the Scheduler in mode *FO* becomes false (i.e there are more than F_O overruns). Within *FR* F_R overruns will be tolerated (with $F_R > F_O$); i.e.

$$\sum_{\forall h \in \mathcal{T}_{\mathcal{H},s} \bullet t > a_h^s \geq m(t)} \left[\frac{e_h^s(t) - C_h(L)}{C_O} \right]_0 \leq F_R$$

We introduce a predicate, $req_k(t)$ (short for required) that returns true if the current job of task k at time t must be executed. Tasks that require all their jobs to execute are assigned, for ease of presentation, $w = 0$. The conditions for the current job (k^c) of task k to be required are: (1) $w_k = 0$, or (2) the task has not yet executed w_k jobs, i.e. $c \leq w$, or (3) one of the previous w_k jobs (before c) had a zero execution time – this is an indication that the job was dropped. This leads to the following definition:

$$req_k(t) \stackrel{\text{def}}{=} w_k = 0 \vee c \leq w_k \vee \exists s, s \in c - w_k..c - 1 \bullet e_k^s(f_k^s) = 0$$

In other words, $req_j(t)$ is false only when the last w_j jobs of τ_j (i.e. $j_j^{c-1}, j_j^{c-2}, \dots, j_j^{c-w_j}$) have completed successfully. A non robust task is always “required” (in that its current job must always complete). The R/G conditions can again be easily derived for the Fail Robust mode:

$$R_S^{FR}(t) \stackrel{\text{def}}{=} \forall k \in act(\mathcal{T}, t) \bullet e_k^c(t) \leq C_k(H) \wedge (a_k^c = t \wedge c > 1) \Rightarrow a_k^c - a_k^{c-1} \geq T_k(L) \wedge \sum_{\forall h \in \mathcal{T}_{\mathcal{H},s} \bullet t > a_h^s \geq m(t)} \left[\frac{e_h^s(t) - C_h(L)}{C_O} \right]_0 \leq F_R$$

Note this is a weakening of the rely condition as $R_S^{FO} \Rightarrow R_S^{FR}$ which follows from $F_R > F_O$ i.e. more overruns can be tolerated in the Fail Robust mode.

We can now complete the full specification. The Scheduler only guarantees execution time to those jobs that are required; moreover, if a job is not required the Scheduler ensures it does not execute.

$$G_S^{FR}(t) \stackrel{\text{def}}{=} \forall k \in act(\mathcal{T}, t) \bullet req_k(t) \Rightarrow t + C_k(L) - e_k^c(t) \leq v_k^c \wedge$$

$$\forall k \in \mathcal{T} \bullet a_k^c = t \wedge \neg req_k(t) \Rightarrow f_k^c = t \wedge$$

$$\forall h \in \mathcal{T}_{\mathcal{H}} \bullet a_h^c \geq m(t) \wedge e_h^c(t) > C_h(L) \Rightarrow t + C_k(H) - e_k^c(t) \leq d_k^c(H)$$

The tasks only need execution time if they are required; their guarantee conditions remain true even if the current job does not execute.

$$R_k^{FR}(t) \stackrel{\text{def}}{=} WCET_k \leq C_k(L) \wedge k \in act(\mathcal{T}, t) \wedge req_k(t) \Rightarrow t + C_k(L) - e_k^c(t) \leq v_k^c$$

$$G_k^{FR}(t) \stackrel{\text{def}}{=} e_k^c(t) \leq C_k(L) \wedge (a_k^c = t \wedge c > 1) \Rightarrow a_k^c - a_k^{c-1} \geq T_k(L)$$

As with mode *FO*, an individual *HI-crit* task can fail (rely condition becomes invalid, false) leading to a weakened specification:

$$R_h^{FR^*}(t) \stackrel{\text{def}}{=} \text{WCET}_h \leq C_h(H) \wedge h \in \text{act}(\mathcal{T}_{\mathcal{H}}, t) \Rightarrow t + C_h(H) - e_h^c(t) \leq d_h^c(H)$$

$$G_h^{FR^*}(t) \stackrel{\text{def}}{=} e_h^c(t) \leq C_h(H) \wedge (a_h^c = t \wedge c > 1) \Rightarrow a_h^c - a_h^{c-1} \geq T_h(H)$$

Note the *req_k* condition has been removed from the rely condition as the current job must be required to execute as it has a non-zero execution time (i.e. a value that exceeded $C(L)$); also this is another weakening of the rely condition.

Again with this specification the Scheduler must rely on a property of the whole set of *HI*-crit tasks, not a specific property of each individual task.

Fail resilient (graceful degradation) – GD. Once the count of job failures becomes greater than F_R , the *FR* mode must be abandoned as the rely condition of the Scheduler becomes false. To add resilience, a number of different general strategies for graceful degradation have been discussed in the literature [55, 45, 54]. Some strategies are hierarchical, in that they form a natural progression of increasingly severe forms of degradation that are invoked by increasingly severe forms of failure. Others take the form of alternative approaches.

All strategies are defined by their level of fault tolerance (the maximum O count they can deal with) and their impact on *LO*-crit tasks. Example strategies include:

1. Increasing the periods and deadlines of *LO*-crit tasks [60, 59, 36, 58, 57, 53, 25], called *task stretching*, the *elastic task model* or *multi-rate* (also see Section 5.1)
2. Imposing only a weakly-hard constraint on the *LO*-crit tasks [24, 51]
3. Decreasing the computation times of the *LO*-crit tasks [13, 4], perhaps by utilising an imprecise mixed-criticality (IMC) model [50, 52, 49, 33] or budget control [26, 27]
4. Moving some *LO*-crit tasks to a different processor that has not experienced a criticality mode change [63, 64, 35, 3].
5. Abandoning *LO*-crit work in a disciplined sequence [23, 34, 28, 56, 46, 47].

Some example strategies have already been described in the paper. Of course the specific set of schemes that may be applicable will depend on the details of the application. Nevertheless, any collection of approaches can be (partially) ordered using preferences and the strengths/weaknesses of the rely conditions of the Scheduler.

In general, the full set of modes forms a lattice with the Normal N mode at the top, and the Fail Safe (*FS*) mode at the bottom (see below). Preferences are assigned to reflect the structure of this lattice (N is the most preferred mode, *FS* the least). The least preferred resilient mode is the one that represents the total abandonment of all *LO*-crit jobs. We define this to be the backstop mode (*BM*). In the following *BM* is entered after the failure of *GD*:

$$R_S^{BM}(t) \stackrel{\text{def}}{=} \forall h \in \text{act}(\mathcal{T}_{\mathcal{H}}, t) \bullet e_h^c(t) \leq C_h(H) \wedge (a_h^c = t \wedge c > 1) \Rightarrow a_h^c - a_h^{c-1} \geq T_h(H)$$

$$G_S^{BM}(t) \stackrel{\text{def}}{=} \forall h \in \text{act}(\mathcal{T}_{\mathcal{H}}, t) \bullet t + C_h(H) - e_h^c(t) \leq d_h^c \wedge \forall l \in \text{act}(\mathcal{T}_{\mathcal{L}}, t) \bullet e_h^c(t) = e_h^c(\eta^{GD})$$

where again η^{GD} is the time this mode is entered (i.e. when some graceful degradation mode, *GD* must be abandoned). Now no active *LO*-crit jobs execute.

$$R_h^{BM}(t) \stackrel{\text{def}}{=} \text{WCET}_h \leq C_h(H) \wedge h \in \text{act}(\mathcal{T}_{\mathcal{H}}, t) \Rightarrow t + C_h(H) - e_h^c(t) \leq d_h^c(H)$$

$$G_h^{BM}(t) \stackrel{\text{def}}{=} e_h^c(t) \leq C_h(H) \wedge (a_h^c = t \wedge c > 1) \Rightarrow a_h^c - a_h^{c-1} \geq T_h(H)$$

$$R_l^{BM}(t) \stackrel{\text{def}}{=} \text{true}$$

$$G_l^{BM}(t) \stackrel{\text{def}}{=} (a_l^c = t) \Rightarrow (f_l^c = t)$$

hence any newly arrived *LO*-crit job is immediately finished (aborted).

Fail safe/restarts – FS. The final “strategy” is fail safe, perhaps via fail stop, followed by a subsequent restart (which may use a cold, warm or hot standby). It is not the purpose of this paper to review these approaches to fault tolerance. But for completeness we note that wherever possible there should be a mode (*FS*) which guarantees a fail safe outcome.

$$P_S^{FS} \stackrel{\text{def}}{=} true$$

$$R_S^{FS}(t) \stackrel{\text{def}}{=} true$$

$$G_S^{FS}(t) \stackrel{\text{def}}{=} t \leq (\eta^{BM} + D_S^{FS})$$

$$Q_S^{FS} \stackrel{\text{def}}{=} safe_shut_down$$

where D_S^{FS} is the (relative) deadline of the scheduler in this mode – there is a bound on how far t can reach.

This mode must be the lowest preference mode (i.e. be at the base of the lattice). It can always be entered, but must only be entered when all Schedulers in other modes have rely conditions that are false. Note we give the Scheduler a deadline in this mode to instigate the shut-down activity, but no further functional information can be given as the Scheduler is no longer operational.

6.2 Robust and resilient mode changes

In the above discussion a number of Scheduler modes have been introduced. They naturally form a sequence based on preference; the inverse of this sequence describes the behaviour of the system as it experiences graceful degradation:

$$N \rightarrow FO \rightarrow FR \rightarrow GD \rightarrow BM \rightarrow FS$$

An application could have a number of intermediate modes between *FR* and *BM*. In addition there could be a number of “best-effort” (not guaranteed) behaviours/modes between *BM* and *FS*.

For the set of operational modes it will be necessary to show they form a hierarchy:

$$R^N \Rightarrow R^{FO} \Rightarrow R^{FR} \Rightarrow R^{GD} \Rightarrow R^{BM} \Rightarrow R^{FS}$$

Moreover, at the time a rely condition becomes invalid and the next mode is entered (at times $\eta^N, \eta^{FO}, \eta^{FR}, \eta^{BM}$), it can be proven (see Lemmas 2 and 3) that the new rely condition is true and henceforth the guarantee condition holds.

In contrast to this gradual decline in functionality, a system that is programmed to recover will move directly from any of the degraded modes back to mode *N*. This move is driven by preference; but to reenter the Normal mode there will be some prerequisites. As noted in Section 5.3 this could be simply that at the time the Normal mode is re-entered there are no active tasks that had been released prior to this time.

7 Conclusions and Future Work

There is extensive published work on Mixed-Criticality scheduling and implementation, but not on their formal specification. We believe formalisation is essential since the notion of mixed criticality has subtle semantics: often concepts such as correctness, resilience and robustness are neither straightforward nor intuitive for such systems. The R/G approach has proved a successful formalism for specifying non-real-time safety-critical systems and our main contribution in this paper is to extend R/G to (i) time, and (ii) multiple criticalities.

The proposed framework is based on an ordering of modes (in general, this would form a lattice) with the normal mode (N) being at the top and a Fail Stop (FS) mode at the base. Each mode has an R/G coupling with a move down the ordering accompanied by a weakening of the rely and guarantee conditions. Examples were used to show that to obtain a true hierarchical relationship between the rely conditions (e.g. $R^A \Rightarrow R^B$, for modes A and B), it is often necessary to strengthen the R^A and/or weaken the R^B conditions. A movement of the system down the ordering (from mode A to B) occurs only when forced by R^A no longer being true. At this time it is necessary to prove that R^B remains true. The return of the system back to mode N is sanctioned by the rely and pre conditions of N being reestablished.

The examples presented in this paper have demonstrated that the developed approach has the expressive power necessary to enable a wide range of possible runtime strategies to be precisely specified and evaluated (in terms of their internal consistency). Further work will address the application of the R/G specifications in the development of the necessary run-time code that will be needed to support these mixed-criticality protocols. This would benefit from mechanical proof support as undertaken by the PROSA team [21, 10]. Although this work is not covered in the current paper there is ample evidence that R/G specifications can form the basis for the formal development of implementations. A useful example is tackled in [43, 41]: although not scheduling per se, Simpson's 4-slot algorithm is a delicate piece of intricate code for asynchronous communication mechanisms. A number of other examples of developments based on R/G specifications are listed and/or tackled in [48, 31, 44, 9].

References

- 1 J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
- 2 J.-R. Abrial. *The Event-B Book*. Cambridge University Press, Cambridge, UK, 2010.
- 3 J. Baik and K. Kang. Schedulability analysis for task migration under multiple mixed-criticality systems. In *Proc Korean Society of Computer Science*, page X, 2019.
- 4 S. K. Baruah, A. Burns, and Z. Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviours. In *Proc. ECRTS*, pages 131–140, 2016.
- 5 S.K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *Proc. of the 19th Annual European Symposium on Algorithms (ESA 2011) LNCS 6942, Saarbruecken, Germany*, pages 555–566, 2011.
- 6 S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.
- 7 I. Bate, A. Burns, and R.I. Davis. An enhanced bailout protocol for mixed criticality embedded software. *IEEE Transactions on Software Engineering*, 43(4):298–320, 2016.
- 8 I. Bate, A. Burns, and R.I. Davis. Analysis-runtime co-design for adaptive mixed criticality scheduling. In *Proc. of forthcoming IEEE RTAS, Pre publication version privately communicated.*, 2022.
- 9 R. Bornat and H. Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, 25(6):893–931, 2013.
- 10 S. Bozhko and B.B. Brandenburg. Abstract response-time analysis: A formal foundation for the busy-window principle. In Marcus Völöp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

- 11 A. Burns. Why the expressive power of programming languages such as Ada is needed for future cyber physical systems. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 3–11. Springer, 2016.
- 12 A. Burns, S. Baruah, C.B. Jones, and I. Bate. Reasoning about the relationship between the scheduler and mixed-criticality jobs. In *Proc. 7th Int. RTSS Workshop On Mixed Criticality Systems (WMC)*, pages 17–22, 2019.
- 13 A. Burns and S.K. Baruah. Towards a more practical model for mixed criticality systems. In *Proc. 1st Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 1–6, 2013.
- 14 A. Burns, R. Davis, S. K. Baruah, and I. Bate. Robust mixed-criticality systems. *IEEE Transactions on Computers*, 67(10):1478–1491, 2018.
- 15 A. Burns and R.I. Davis. Response-time analysis for mixed-criticality systems with arbitrary deadlines. In *Proc. Workshop on Mixed Criticality Systems (WMC)*, pages 13–18, 2017.
- 16 A. Burns and R.I. Davis. A survey of research into mixed criticality systems. *ACM Computer Surveys*, 50(6):1–37, 2017.
- 17 A. Burns and R.I. Davis. Mixed criticality systems: A review (13th edition). Technical Report MCC-1(13), available at <https://www-users.cs.york.ac.uk/~burns/review.pdf> and the White Rose Repository, Department of Computer Science, University of York, 2022.
- 18 A. Burns and I.J. Hayes. A timeband framework for modelling real-time systems. *Real-Time Systems Journal*, 45(1–2):106–142, June 2010.
- 19 A. Burns, I.J. Hayes, and C.B. Jones. Deriving specifications of control programs for cyber physical systems. *Computer Journal*, 63(5):774–790, 2020.
- 20 G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *IEEE Real-Time Systems Symposium*, pages 286–295, 1998.
- 21 F. Cerqueira, F. Stutz, and B.B. Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *Proc. 28th Euromicro Conference on Real-Time Systems (ECRTS)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 273–284, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- 22 Diego Machado Dias. *Mechanising an algebraic rely-guarantee refinement calculus*. PhD thesis, School of Computing, Newcastle University, 2017.
- 23 T. Fleming and A. Burns. Incorporating the notion of importance into mixed criticality systems. In L. Cucu-Grosjean and R. Davis, editors, *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 33–38, 2014.
- 24 O. Gettings, S. Quinton, and R.I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proc. International Conference on Real-Time Networks and Systems (RTNS)*, pages 237–246, 2015.
- 25 C. Gill, J. Orr, and S. Harris. Supporting graceful degradation through elasticity in mixed-criticality federated scheduling. In Jing Li and Zhishan Guo, editors, *Proc. 6th Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 19–24, 2018.
- 26 X. Gu and A. Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 47–56. IEEE, 2016.
- 27 X. Gu and A. Easwaran. Dynamic budget management and budget reclamation for mixed-criticality systems. *Real-Time Systems*, 55:552–597, 2019.
- 28 X. Gu, K.-M. Phan, A. Easwaran, and I. Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *Proc. 27th ECRTS*, pages 13–24. IEEE, 2015.
- 29 I. J. Hayes. Generalised rely-guarantee concurrency: An algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, November 2016.
- 30 I.J. Hayes, M. Jackson, and C.B. Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 154–169. Springer Verlag, 2003.

- 31 I.J. Hayes and C.B. Jones. A guide to rely/guarantee thinking. In Jonathan Bowen, Zhiming Liu, and Zili Zhan, editors, *Engineering Trustworthy Software Systems – Third International School, SETSS 2017*, volume 11174 of *LNCIS*, pages 1–38. Springer, 2018.
- 32 C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- 33 L. Huang, I-H. Hou, S.S. Sapatnekar, and J. Hu. Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems. In *Proc. of the 26th International Conference on Real-Time Networks and Systems*, RTNS, pages 159–169. ACM, 2018.
- 34 P. Huang, P. Kumar, N. Stoimenov, and L. Thiele. Interference constraint graph: A new specification for mixed-criticality systems. In *Proc. 18th Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2013.
- 35 S. Iacovelli, R. Kirner, and C. Menon. ATMP: An adaptive tolerance-based mixed-criticality protocol for multi-core systems. In *Proc. IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9, 2018.
- 36 M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low criticality tasks in mixed criticality system. In *Proc. 1st WMC, RTSS*, pages 43–48, 2013.
- 37 C.B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.
- 38 C.B. Jones. Specification and design of (parallel) programs. In *Proc. of IFIP*, pages 321–332. North-Holland, 1983.
- 39 C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. URL: <http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/Jones1990.pdf>.
- 40 C.B. Jones and A. Burns. A rely-guarantee specification of mixed-criticality scheduling. In Valentin Cassano and Nazareno Aguirre, editors, *Mathematical Foundations of Software Engineering: Essays in Honor of Tom Maibaum on the Occasion of his Retirement*, Tribute Series. College Publications, 2022.
- 41 C.B. Jones and I.J. Hayes. Possible values: Exploring a concept for concurrency. *Journal of Logical and Algebraic Methods in Programming*, 85(5):972–984, 2016.
- 42 C.B. Jones, I.J. Hayes, and M.A. Jackson. Deriving specifications for systems that are connected to the physical world. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer Verlag, 2007.
- 43 C.B. Jones and K.G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.
- 44 C.B. Jones and N. Yatapanage. Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example. *Formal Aspects of Computing*, 31(3):353–374, 2019. on-line April 2018.
- 45 J.C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Digest of Papers, The Fifteenth Annual International Symposium on Fault-Tolerant Computing*, pages 2–11, Michigan, USA, 1985.
- 46 J. Lee, H.S. Chwa, L.T.X. Phan, I. Shin, and I. Lee. MC-ADAPT: Adaptive task dropping in mixed-criticality scheduling. *ACM Trans. Embed. Comput. Syst.*, 16:163:1–163:21, 2017.
- 47 J. Lee and J. Lee. Mc-flex: Flexible mixed-criticality real-time scheduling by task-level mode switch. *IEEE Transactions on Computers*, page online, 2021. doi:10.1109/TC.2021.3111743.
- 48 Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 455–468, New York, NY, USA, 2012.

- 49 D. Liu, N. Guan, J. Spasic, G. Chen, S. Liu, T. Stefanov, and W. Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(7):975–991, July 2018.
- 50 D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees. In *Proc. IEEE RTSS*, pages 35–46, 2016.
- 51 R. Medina, E. Borde, and L. Pautet. Directed acyclic graph scheduling for mixed-criticality systems. In Johann Bliederger and Markus Bader, editors, *Reliable Software Technologies – Ada-Europe*, pages 217–232. Springer International Publishing, 2017.
- 52 R.M. Pathan. Improving the quality-of-service for scheduling mixed-criticality systems on multiprocessors. In Marko Bertogna, editor, *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76 of *Leibniz International Proc. in Informatics (LIPIcs)*, pages 19:1–19:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.
- 53 S. Ramanathan, A. Easwaran, and H. Cho. Multi-rate fluid scheduling of mixed-criticality systems on multiprocessors. *Real-Time Systems*, 54:247–277, 2018.
- 54 B. Randell, J-C. Laprie, H. Kopetz, and B. Littlewood(Eds.). *Predictably Dependable Computing Systems*. Springer, 1995.
- 55 B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, 1978.
- 56 J. Ren and L.T.X. Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *Proc. 27th ECRTS*, pages 25–36. IEEE, 2015.
- 57 H. Su, P. Deng, D. Zhu, and Q. Zhu. Fixed-priority dual-rate mixed-criticality systems: Schedulability analysis and performance optimization. In *Proc. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 59–68. IEEE, 2016.
- 58 H. Su, N. Guan, and D. Zhu. Service guarantee exploration for mixed-criticality systems. In *Proc. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2014.
- 59 H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proc. of the Conference on Design, Automation and Test in Europe, DATE*, pages 147–152, 2013.
- 60 H. Su, D. Zhu, and D. Mosse. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Proc. RTCSA*, 2013.
- 61 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.
- 62 N. Vreman, A. Cervin, and M. Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In Björn B. Brandenburg, editor, *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- 63 H. Xu and A. Burns. Semi-partitioned model for dual-core mixed criticality system. In *23rd International Conference on Real-Time Networks and Systems (RTNS 2015)*, pages 257–266, 2015.
- 64 H. Xu and A. Burns. A semi-partitioned model for mixed criticality systems. *Journal of Systems and Software*, 150:51–63, 2019.

Achieving Isolation in Mixed-Criticality Industrial Edge Systems with Real-Time Containers

Marco Barletta  

Università degli Studi di Napoli Federico II, Italy

Marcello Cinque  

Università degli Studi di Napoli Federico II, Italy

Luigi De Simone  

Università degli Studi di Napoli Federico II, Italy

Raffaele Della Corte  

Università degli Studi di Napoli Federico II, Italy

Abstract

Real-time containers are a promising solution to reduce latencies in time-sensitive cloud systems. Recent efforts are emerging to extend their usage in industrial edge systems with mixed-criticality constraints. In these contexts, isolation becomes a major concern: a disturbance (such as timing faults or unexpected overloads) affecting a container must not impact the behavior of other containers deployed on the same hardware. In this paper, we propose a novel architectural solution to achieve isolation in real-time containers, based on real-time co-kernels, hierarchical scheduling, and time-division networking. The architecture has been implemented on Linux patched with the Xenomai co-kernel, extended with a new hierarchical scheduling policy, named `SCHED_DS`, and integrating the RTNet stack. Experimental results are promising in terms of overhead and latency compared to other Linux-based solutions. More importantly, the isolation of containers is guaranteed even in presence of severe co-located disturbances, such as faulty tasks (elapsing more time than declared) or high CPU, network, or I/O stress on the same machine.

2012 ACM Subject Classification Software and its engineering → Real-time systems software

Keywords and phrases Real-time, Mixed-criticality, Containers, Edge computing

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.15

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.8.1.1>

Funding This work has been partially supported by the project COSMIC of UNINA DIETI and by the R&D project “REINForce: REsearch to INspire the Future” (CDS000609), funded by the Italian Ministry for Economic Development (MISE).

1 Introduction

Nowadays, we are witnessing the spread of cloud (including fog and edge) technologies in industrial domain and cyber-physical systems, such as Industrial Internet of Things (IIoT) [31], Industry 4.0, automotive [2], enabling the remote housing of critical software components on the edge of infrastructure, according to the *software defined everything* trend [51]. In these scenarios, edge computing systems can be seen as mixed-criticality systems (MCSs) [5], consolidating real-time tasks with different levels of criticality along non-real-time tasks, all of them on a reduced number of computing nodes in order to reduce the size, weight, power, and cost of hardware. An example is represented by the predictive maintenance of a wind turbine based on artificial intelligence, where the real-time data acquisition/preprocessing is run on the same edge node of a deep neural network [41]. Another example is real-time



© Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 15; pp. 15:1–15:23

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



control in Industry 4.0 applications, where control activities are designed as tasks with different levels of criticality that perform the control functions, visualization, and interaction with other devices on the same node [47].

Given this scenario, timeliness and isolation become major concerns, since a *deadline miss* could result in a severe failure or hazard. For example, if a critical virtualized ECU on a car is late at handling a brake pedal command, due to a co-located disturbance caused by a non-critical component, this could result in a crash [44]. Similarly, late sensor feedback in a smart factory could bring to the blocking of a production line or even cause injuries.

Recently, container-based virtualization has been gaining the limelight in cloud environments for several reasons: low overhead, simplified migration/orchestration of software appliances, and increased scalability if compared to traditional hypervisor-based approach. Containers are thus considered today a key technology to achieve high workload consolidation and to better exploit hardware resources. For the same reasons, they are starting to attract interest for the realization of real-time systems [7, 48]. However, the adoption of containers in industrial edge systems requires facing compelling challenges due to isolation guarantees needed between different co-located workloads. First, strict resource reservation (such as CPU and network bandwidth) must be guaranteed with acceptable overheads (e.g., low scheduling latency), to grant predictable timing behavior to networked tasks running within critical containers. Second, disturbances must not affect the correct execution of such tasks, other containers (e.g., timing faults in tasks elapsing longer than declared), or the system in general (e.g., unexpected high CPU, I/O, or network load on the same machine).

Several approaches implementing real-time containers have been proposed in literature. Among these, we focus on Linux co-kernel-based solutions due to their wide use in industrial settings, e.g., for robot control [18, 58] and industrial networking support [30], and to their known benefits in terms of latency [13, 26] and isolation, as they make Linux fully preemptable in favor of real-time tasks. Nevertheless, co-kernel-based container solutions available to date enforce isolation through active monitoring, which introduces overhead and represents a single point of failure. In addition, existing real-time containers solutions focus on CPU reservation, neglecting network isolation issues, which are fundamental for networked real-time tasks in industrial edge systems.

We aim to overcome existing issues, proposing a novel architectural solution to achieve container isolation in mixed-criticality edge systems. Specifically, our contributions are:

- a *hierarchical scheduling solution*, named `SCHED_DS`, for real-time containers managed by a real-time co-kernel, which results in promising scheduling runtimes (about hundreds of *ns* on our setting) with no need of active monitoring but proactively isolating CPU;
- a *schedulability test*, adapted from early results [16], to verify the feasibility of deployment of a newly created container on existing infrastructure, useful for orchestration purposes;
- a *POSIX-compliant API* to foster the transparent adoption of the solution by practitioners;
- the integration of the solution with a *real-time networking stack* (specifically, RTnet [33]), to achieve network bandwidth guarantees with time-division medium access.

The proposed architecture has been implemented on the Xenomai co-kernel [32], by modifying its `SCHED_QUOTA` to fit a hierarchical deferrable server model (allowing theoretical treatment for schedulability analysis). The resulting policy, `SCHED_DS`, is transparently adopted by real-time tasks running within containers through our API, which also enables containers to use the real-time network. Experimental results show the benefits of the proposed solution in our settings, which exhibit a scheduling runtime in the order of hundreds of nanoseconds, and an overhead lower than 0.1%. More importantly, they highlight the isolation properties of our solution, as tasks run within critical containers are not impacted by faulty tasks running in other containers or by the presence of high CPU, network, or I/O load on the same hardware. Finally, the proposed solution outperforms, in terms of task activation

latency, recent state-of-the-art implementations based on low-latency Linux containers and hierarchical scheduling, in respect to which our solution can be seen as complementary. For instance, compared to `PREEMPT_RT`, we assessed a relative improvement of at least 30%.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the proposed architecture (see Subsection 3.1), the system model and schedulability analysis (see Subsection 3.2), and the proposed hierarchical scheduling solution (see Subsection 3.3). Section 4 gives implementation details of the architecture on top of the Xenomai co-kernel. Section 5 provides a thorough experimental analysis for proving the applicability of the proposed architecture. Section 6 concludes the paper.

2 Related Work

Currently, mixed-criticality systems development, particularly in edge computing scenarios, brings several stringent requirements like tiny memory footprints (a few MBs or even KBs), fast instantiation times (tens of milliseconds or less), high network throughput (10-40 Gb/s), high consolidation (e.g., being able to run thousands of instances on a single commodity server), and a reduced attack surface for certification purposes. For this reason, lightweight virtualization approaches, like containers, are becoming attractive and starting to be explored in industrial domains.

PREEMPT_RT. Recent studies have started to explore the possibility to use containers to run real-time tasks. For example, [36] presents an empirical study on the problem of minimizing computational and networking latencies for Radio Access Networks through lightweight containers. The study analyzes the performance of Docker containers running on the top of a Linux kernel patched with the well-known `PREEMPT_RT` [17, 45]. This modifies the Linux kernel in order to provide real-time guarantees (e.g., predictability, low latencies) still using a single-kernel approach. The results highlight that using `PREEMPT_RT` improves latencies on Docker containers when compared to a generic kernel. Similarly, authors in [38] propose a sandboxed environment, still based on Docker containers on a `PREEMPT_RT`-based Linux kernel, in the context of an automotive scenario. In [29] a container-based architecture for real-time automation controllers is proposed, using Docker and LXC containers on top of a Linux patched with the `PREEMPT_RT` real-time kernel patch. Experiments have been done using both Docker and LXC containers running on top of a Linux OS patched with the `PREEMPT_RT` real-time kernel patch. Tests emphasize that the use of containers for control applications can meet the requirements of the target systems.

RT-cgroups. Other solutions in literature are based on the control groups (*cgroups*) [40]. Currently, the Linux kernel offers the *real-time group scheduling* (`rt-cgroups`) to host real-time containers [54]. However, the implemented algorithm is somehow unclear [1] and even worse the `PREEMPT_RT` patch has been in conflict with this kind of group scheduling, allowing only a low latency configuration (until v5.15.34) [25] and preventing the fully preemptable one. In [1], the authors propose to modify the `rt-cgroups` implementation by using a real-time deadline-based scheduling policy. The offered solution extends the `SCHED_DEADLINE` policy to schedule Linux control groups. It implements a two-level hierarchical scheduling framework [19] [42] in Linux, allowing user threads to be scheduled with fixed priorities inside the control group scheduled by `SCHED_DEADLINE`.

Co-kernel approach. Besides *cgroups*, alternative approaches include using real-time Linux co-kernels, such as RTAI [20] and Xenomai [32]. The main idea behind the co-kernel approach is to have a small real-time core running at higher priority, which can intercept interrupts and defer them if needed. This choice allows keeping the benefits of the Linux ecosystem (e.g., fully re-using container engines and control group features), while making Linux fully-preemptable in favor of real-time tasks and including the support for real-time networking in the co-kernel, making them a good fit for industrial settings [18, 30, 58]. Generally, RTOS and co-kernel approaches outperform `PREEMPT_RT` in terms of latencies and task-switch times, while Linux is better on average performance, as expected by its GPOS nature [6, 13, 14, 26, 27, 45]. On this line, *Tasci et al.* [50] leveraged both real-time patch and co-kernel approaches to run real-time control applications within Docker containers. In our past work ([8] and [9]) we adopted RTAI to schedule hard real-time tasks within containers, using either fixed priorities in the former or Earliest Deadline First (EDF) [35] in the latter. The idea is to fully inherit the advantages of the co-kernel, in terms of real-time performance and functionalities, while letting tasks within containers keep using the same application programming interface. However, both solutions require active monitoring of tasks run within containers, to tolerate timing failures, which introduces overhead and a single point of failure to the architecture. Further, our solution in [8] requires a global fixed-priority assignment that may limit the adoption of built-in orchestrators, as priorities might be re-assigned if a container needs to be moved from a host to another during its lifespan.

Container orchestration. Container-based virtualization is an attractive choice for time-sensitive edge systems also for the orchestration capabilities provided, such as migration, balancing, and high-availability mechanisms. For example, in [28] and [4] the need for a Kubernetes-compatible [53] edge container orchestrator is highlighted since edge devices are gaining enough power to run containerized services while remaining small and low-powered. Different comparisons and analyses to understand the advantages and disadvantages of the existing orchestrators for edge cloud are made in the two studies. In [24, 49] containers based on *rt-cgroups* are integrated into Kubernetes in order to orchestrate real-time containers for low latency applications, while in [12] the same containers have been integrated into OpenStack in order to build a framework for NFV for next generation networks. In [59] a *latency-aware* edge computing platform based on Storm is introduced to run computer vision applications with a real-time response time exploiting edge resources. In the context of the LF Edge foundation [52], Xilinx is currently developing a Xen-based lightweight orchestration solution called *RunX* [57]. *RunX* allows running containers as VMs, either with the provided custom-built Linux-based kernel with a Busybox-based ramdisk, or with a container-specific kernel/ramdisk. Despite some efforts from both industry and academia, there is the need for further analysis in the context of mixed-criticality edge computing systems.

Hypervisor-based solutions. A consolidated trend for time and space partitioning in mixed-criticality systems is to use hypervisor-based solutions, with the aim to completely separate real-time kernels from non-real-time ones. *RT-Xen* [56] is a real-time hypervisor scheduling framework, which extends Xen to support VMs with real-time performance requirements as well as to enable both global and partitioned VM schedulers. *XTRATUM* [39] is a type 1 hypervisor designed for real-time embedded system. *XTRATUM* can be used to build partitioned systems, and provides both temporal and spatial separation. The Wind River *VxWorks RTOS* [55] features a Virtualization Profile that integrates a real-time embedded, type-1 hypervisor into VxWorks. The hypervisor is able to slow down general-purpose

operating systems to ensure that real-time ones can execute without performance impact. *Jailhouse* [46] is a Linux-based partitioning hypervisor for mixed-critical applications; it enables asymmetric multiprocessing to run both bare-metal applications or guest operating systems (including RTOS). Jailhouse splits physical resources into isolated compartments named cells, with a root cell dedicated to run the Linux kernel and the hypervisor itself, and non-root cells assigned to one guest OS. *Bao* [37] is an open-source lightweight embedded hypervisor for mixed-criticality systems. Bao aims at providing isolation for fault-containment and real-time guarantees, through a static partitioning hypervisor architecture: resources are statically partitioned and assigned at VM instantiation time.

Our proposal advances the state-of-the-art in terms of readiness for industrial settings, since we propose a complementary solution to Linux real-time containers that fully exploits the advantages of co-kernels, while overcoming the current limitations of co-kernel-based container solutions. In particular, we avoid global fixed-priority assignment and active monitoring, using hierarchical scheduling and enabling proactive temporal protection at the co-kernel level, to provide guaranteed CPU bandwidths (i.e., utilization) to different tasks grouped in different containers. Further, the proposed architecture is designed to easily support container orchestration tools in the context of edge computing scenarios, and it integrates a real-time networking stack accessible from containers, an aspect of crucial importance for mixed-criticality edge and neglected by the current literature on real-time containers. Finally, concerning hypervisor-based solutions, our container-based approach allows running multiple isolated Linux systems on a single host with minimal space and performance overhead, avoiding the need of running full VMs (each one replicating the entire OS stack). Other tiny hypervisor-based solutions are based on the concept of *static partitioning* [37, 39, 46]. This makes the environment less prone to changes since this kind of hypervisor includes a one-to-one mapping between virtual CPUs and physical CPUs, and devices are mapped directly into the guest memory areas. Despite the lightness of these approaches, in our context, the use of container-based solutions paves the way to service orchestration and migration, automatic and easy deployment of tasks, which is fundamental with changing environments at the edge level.

3 Proposal

3.1 Architecture

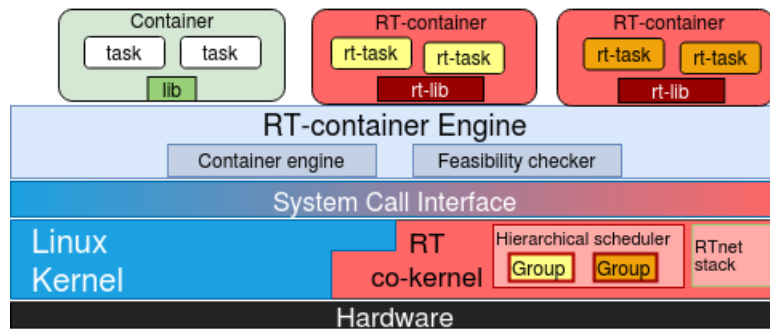
Figure 1 depicts our proposed architecture for achieving isolation in real-time containers. Similarly to earlier proposals [8, 9], we envision the concept of real-time containers (*rt-containers*) hosting real-time tasks (*rt-tasks*) and run on the top of an operating system with real-time scheduling capabilities. The idea is letting *rt-containers* run along with traditional containers (running non-real-time tasks) on the same machine, with temporal isolation guarantees for *rt-tasks*. This enables a container-based mixed-criticality environment.

The proposal encompasses an engine layer based on the Docker *container engine* (although other container engines should work too) and includes a *feasibility checker*, which executes a schedulability test to verify (in terms of both CPU and network) if a new *rt-container* can be created on the computing node, without affecting the containers already hosted. This is particularly useful for orchestration purposes. Moreover, compared to solutions in [1, 54], our proposal is based on *Linux kernel* coupled with a *real-time co-kernel*.

Differently from existing co-kernel-based solutions [8, 9], which enforce temporal protection of *rt-tasks* through a dedicated real-time monitoring component, our proposal leverages a *hierarchical group scheduling* approach, called `SCHED_DS`. The idea is to map the *rt-tasks*

15:6 Achieving Industrial Edge MCS Isolation with RT-Containers

running within an *rt-container* in a *task group* at OS co-kernel level, where each group has its own bandwidth (i.e., slices of CPU time) limitation. In this way, the CPU consumption of *rt-tasks* is limited, by design, over a globally defined period. The *rt-tasks* with common requirements are pooled in groups; each group receives a share of the global period. Therefore, our proposal does not require active monitoring for temporal protection of *rt-tasks*, which represents a single point of failure as well as a source of overhead [8,9]. Moreover, differently from the solution in [8], the priority assignment is not globally fixed: within an *rt-container*, the developer is free to choose task priority levels, as the group CPU bandwidth will be independently guaranteed by the hierarchical scheduler.



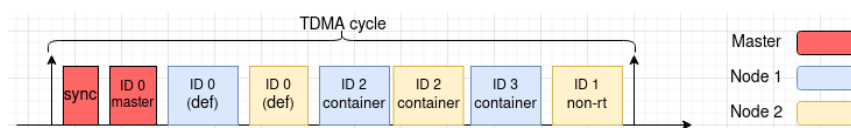
■ **Figure 1** The proposed *rt-container* architecture.

Our proposal includes also the *rt-lib* layer, which provides a transparent mapping of *rt-tasks* onto the underlying real-time core. *rt-lib* exposes a POSIX compliant API towards *rt-tasks* by hiding the information exchanged between the container level and the OS layer, including the one to leverage the co-kernel scheduling policies. It is worth noting that, once a task within a *rt-container* is mapped to a real-time group (through the *rt-lib* API) and starts to run, it is scheduled directly by the real-time co-kernel. Hence, it fully preempts Linux and all indirection levels above have no impact on real-time path of the tasks and their scheduling latency.

Finally, we want to remark that our solution integrates also a real-time networking stack for real-time containers, in contrast with past studies. This could be useful in a container-based cloud/edge environment, where several containers, with both non-real-time and real-time requirements, can communicate over the network [11,43]. We rely on the *RTnet stack* [33] usually shipped with real-time co-kernels (such as Xenomai and RTAI) to provide a real-time networking stack. *RTnet* works with a pluggable real-time medium access control (MAC) policy on standard Ethernet hardware. It offers, by default, a MAC TDMA (Time Division Multiple Access) policy, which consists in allocating a time slot to each physical node of the network for sending/receiving traffic.

Real-time networking guarantees for *rt-containers* are enforced by assigning a slot to each *rt-container*, as if they were physical nodes on the network, taking advantage of the *RTnet* multi-slot configuration. Therefore, the *rt-lib* is in charge of mapping a socket opened by an *rt-container* with a *RTnet* socket that has its own slot for transmitting packets. Figure 2 shows an example of TDMA scheduling in the context of *rt-containers*, in which each slot is identified on the node by an ID, and each node has a default slot for sending traffic (ID 0).

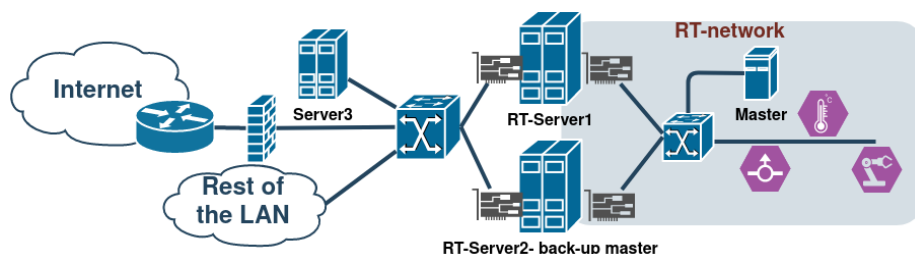
The network relies on a master node providing sync frames, which delimit the TDMA cycles. Since slots cannot overlap, an entity such as an orchestration system should store all the information needed (e.g. slot times and offsets) to assign slots to containers and to



■ **Figure 2** Example of TDMA scheduling.

change dynamically the network configuration. Indeed, RTnet allows adding, removing, or modifying time slots dynamically. For these reasons, the container admission request should contain the network requirements (time slot needed) as well, other than CPU bandwidth. For non-real-time networking instead, well-known methods can be used such as non-real-time buddy tasks, which use the standard Linux networking stack. We highlight that non-real-time traffic is allowed in RTnet in a tunneled way, if necessary.

Figure 3 depicts a typical industrial edge networking scenario in which our proposal can be successfully adopted. The scenario includes a real-time RTnet network interconnecting two edge *RT-Servers* (acting as slaves), a master node, sensors, and actuators. The *RT-Servers* run several containers and carry out part of the workload with critical timing requirements; however, they can also run non-real-time workloads. The master node represents a high precision clock node supporting RTnet TDMA, which is kept separated from *RT-Servers* to improve reliability and security. The *RT-Servers* are connected to both the industrial RTnet and the Internet with two distinct Network Interface Controllers (NICs). In this scenario the rt-containers architecture in Figure 1 can be deployed on *RT-Servers* in order to manage the scheduling and orchestration of both real-time and non-real-time workloads while ensuring isolation of rt-containers, which may contain critical tasks to retrieve data from sensors and send commands to actuators through the industrial network.



■ **Figure 3** Potential scenario: deployment of rt-containers on real-time edge servers.

3.2 System Model

We assume a system composed of M rt-containers, each of them with an assigned CPU bandwidth (i.e., utilization) U_j^C , a priority level P_j , possibly a network timeslot TS_j and a criticality level CL_j , where $j \in [0, M]$. The criticality represents the severity of consequences in case of failure and it is a notion different from priority, that could be assigned by the schedulability algorithm.

Each rt-container hosts N_j sporadic hard real-time tasks $\tau_i^j : i = 0 \dots N_j - 1$, characterized by a WCET C_i^j , a minimum inter-arrival separation (or period) T_i^j and a priority level $P_i^j : i = 0 \dots N_j - 1$. We assume T_i^j to be coincident with the relative deadline D_i^j of the task. Priorities within a container can be freely assigned by the application designer. We assume that tasks can suffer from timing failures, i.e., they can run for a time greater than their declared WCET, because of a wrong timing analysis or a bug inside the code, like an

endless loop. Overall, the system is composed of a set Γ of N tasks, each of them assigned to an rt-container with a given bandwidth. With $\Gamma(j)$ we indicate the subset of tasks within the j -th rt-container. Thus: $\Gamma = \Gamma(0) \cup \dots \cup \Gamma(M-1)$ and $\Gamma(k) \cap \Gamma(h) = \emptyset$, where $k \neq h$. The minimum CPU bandwidth of each rt-container is defined as the sum of its task CPU utilization, defined as $U_i^j = C_i^j/T_i^j$. We define the overall system bandwidth as: $U_{TOT} = \sum_{j=0}^{M-1} \sum_{i=0}^{N_j-1} U_i^j$.

We assume non-real-time tasks, internal or external to containers, running with a best-effort policy, receiving the bandwidth unused by the real-time tasks. The model must provide isolation from potential timing failures due to both excessive non-real-time workload on the same machine and faulty tasks, exceeding their declared WCET, run within rt-containers. In order to provide isolation, we rely on bandwidth reservation, assuming a two-level scheduler with a deferrable server¹ at the root level: each rt-container is thus mapped on a server. At the leaf level, the policy is a fixed priority preemptive scheduler, serving tasks within containers. The admission check is performed by the RT-container engine via the *feasibility checker* (implementation details in Section 4), or by an *orchestrator*. We can take advantage of the schedulability analysis presented in [15, 16], where an exact response time analysis improving the already existing analyzes for hierarchical fixed priority schedulers is presented. We recap briefly the test. The response time analysis is described by equations (1) and (2).

$$L_i(w_i) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \quad \text{with } w_i^0 = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right) (T_S - C_S) \quad (1)$$

$$w_i^{n+1} = L(w_i^n) + \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + \sum_{\substack{\forall X \in hp(S) \\ \text{servers}}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) T_s\right) + J_X}{T_X} \right\rceil C_X \quad (2)$$

Equation (1) represents the task load at priority level i and higher, ready to be executed in the *busy period* w_i . $hp(i)$ is the set of tasks that have priorities higher than task τ_i (the task under analysis) within the same server (i.e., a container in our study), and J_j is the release jitter of the task, which is 0 for a bound task and $T_S - C_S$ for unbound tasks, due to the functioning of the server. Equation (2) is a recursive relation for w_i , namely the interval between the time a task is released and its completion time, where $hp(S)$ are the higher priority servers (i.e., containers) for the server S of the task under analysis, J_X is the release jitter of the higher priority server X , which is $T_X - C_X$ for a deferrable server.

The equations model the critical instant for a task scheduled belonging to the server. The recurrence starts with the value in Equation (1) and ends either when $w_i^{n+1} = w_i^n$, in this case $w_i^n + J_i$ indicates the task response time, or when $w_i^{n+1} > D_i - J_i$, thus the task is not schedulable. It is worth nothing that for the schedulability of a task only the budget of higher priority servers is needed. Therefore, even if tasks belonging to other servers (i.e., other rt-containers) are faulty, the schedulability analysis does not need to be modified.

In order to ease the implementation, we hypothesize that the refill timer, i.e., the timer driving the replenishment of the deferrable server budget, is shared between every task group, thus server periods elapse in a lockstep fashion and there is no chance to suffer from back to back hits, which is typical in deferrable servers. This simplifies the third term in Equation (2), which can be rewritten as:

¹ A deferrable server [35] is the simplest of bandwidth-preserving servers, and it provides a good tradeoff between implementation complexity and low response times. It has an execution budget C_s that is replenished each period T_s . A deferrable server preserves its budget until the end of the period.

$$\sum_{\substack{\forall X \in hp(S) \\ servers}} C_X \quad (3)$$

This is possible since having common periods for all servers, a server can run for no more than its runtime budget in each period. This simplification can be demonstrated analytically and we provided the demonstration in an appendix ². This also simplifies the server schedulability analysis, since we just have to keep the sum of budgets beneath the period value. Therefore, the server scheduling test becomes: $\sum_{\forall S} C_S \leq T$, where S represents a server, and C_S represents its budget in a period, which is obtained as $C_S = U_S * T$, where U_S is the server bandwidth and T is the common period.

Since the period is fixed, we can use algorithms presented in [16] for optimal schedulability, i.e., the optimal priority assignment or the optimal server capacity allocation. Even if theoretically both algorithms could be used, the optimal priority assignment one requires to assign in advance a bandwidth to containers, which makes it hard to obtain a schedulable set. It is more practical to know container criticality that can be associated with a suitable priority and then compute the needed bandwidth. Therefore, we decide to use the optimal server capacity allocation algorithm described in [16].

Similarly to server scheduling, the test for the network is: $\sum_{\forall N} \sum_{\forall S_{net} \in N} TS_S \leq T_{cycle}$, where N is an RTnet node, S_{net} is a container requiring RTnet, T_{cycle} is the length of the TDMA cycle and TS_S are timeslots not overlapping in time. It is important to notice that despite the network timeslot being independent of the CPU time, it must be taken into account for task dependencies.

3.3 The SCHED_DS policy

In this section, we describe the proposed hierarchical scheduling solution, named SCHED_DS, for real-time containers managed by a real-time co-kernel. We assume two levels of runqueues: one for the groups and one for tasks belonging to each group. The policy works as follows. It picks the highest priority group with a ready thread, checks its budget and if expired it is moved to an expired list and another group is picked. If the group has budget, the policy picks the highest priority ready thread within the group. The thread is removed from the runqueue, and if the group becomes empty, it is removed from the group runnable queue. The thread starts running and a timer is armed to expire at the end of the budget. Periodically, a refill timer moves all expired groups back to the runqueue and replenishes budgets. When a thread becomes ready it is enqueued in the runnable queue of its group and if it is the only thread in the queue the group is enqueued in the group runnable queue. Algorithm 1 presents the pick function of SCHED_DS.

4 Implementation

4.1 A Xenomai-based implementation

For implementing the SCHED_DS policy, we chose Xenomai as a co-kernel due to its flexibility, maintainability, and extensibility, and its POSIX-compliant library. Above all, Xenomai recently introduced SCHED_QUOTA and SCHED_TP policies as partitioned hierarchical scheduling solutions, useful to overcome SCHED_FIFO limits. The SCHED_QUOTA policy enforces a

² Appendix is available at <http://www.fedoa.unina.it/13352/>

■ **Algorithm 1** sched_ds_pick pseudocode.

tg = thread group, rq = runqueue, T_{old} is the scheduled out thread, otg = group of T_{old} .

<pre> 1: procedure SCHED_DS_PICK 2: now ← current time 3: if tg of T_{old} is null then 4: goto pick (6) 5: subtract (now - start_time) from otg budget 6: pick highest priority ready tg from rq 7: if tg is null then 8: stop limit timer 9: return null 10: if tg is empty then 11: dequeue tg 12: goto pick (6) </pre>	<pre> 13: if tg runtime budget is 0 then 14: enqueue tg in expired queue 15: goto pick (6) 16: start_time of tg ← now 17: pick highest prio ready thread from tg rq 18: if otg == tg AND limit timer is running AND budget_refilled == false then 19: goto out (22) 20: budget_refilled ← false 21: arm limit timer to go off at now+tg budget 22: decrease active threads in tg 23: return selected thread </pre>
---	---

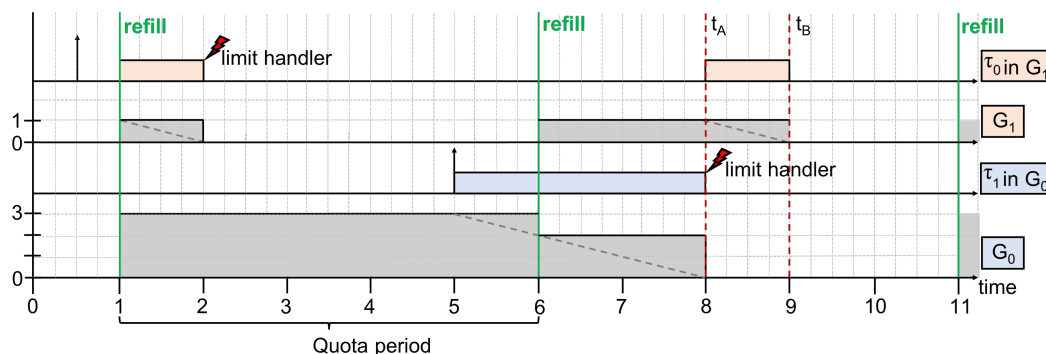
limitation on the CPU consumption of threads over a globally defined period, known as the *quota interval*. This is done by pooling threads with common requirements into groups, and giving each group a share of the global period. On the other hand, the SCHED_TP policy is a temporal partition scheduler that allows installing a schedule made up of consecutive time windows in a fixed-cyclic style, leaving slots for non-real-time tasks. Even though a prototype based on SCHED_TP was built and tested, we preferred leveraging on SCHED_QUOTA because of its flexibility (e.g., it allows adding and removing single groups at runtime), which is required in cloud/edge contexts. We remark that Xenomai maintainers have partially already addressed the lack of the PID namespace support, which anyway does not undermine the contribution of this study.

4.2 The SCHED_QUOTA Limitations

The SCHED_QUOTA policy acts in this way: it picks the highest priority ready thread, and the budget of its group is checked. If it is expired, the thread is moved to an expired list and another thread is picked, otherwise, it starts running with a timer armed to expire at the end of the budget. Periodically, the refill timer moves all expired threads back to the runqueue and replenishes budgets. This implementation revealed three major limitations: **(L1)** the vanilla policy is not actually hierarchical since scheduling is at thread-level; further, the notion of *quota group* only regards the budget evaluation, which is done for each picked thread, causing an $O(N)$ complexity; **(L2)** there is no notion of group priority, since SCHED_QUOTA relies on the unique fixed-priority runqueue, thus giving no assurance about isolation between threads of different groups; **(L3)** the scheduler entails a credit mechanism, along with other subtle problems, that prevent from modeling a *quota group* as a deferrable server.

Starting from (L3), we highlight a devious problem: as soon as the refill timer expires, the budgets are refilled, a rescheduling is forced, and the consumed CPU time is subtracted from the budget of the outgoing quota group. This could result in a group starting the new period with a reduced budget if it kept the highest priority before and after the refill. A related problem is that when a group starts executing, the limit timer is armed to expire when the budget is exhausted, without taking care if this moment in time precedes or follows a budget refill. Figure 4 shows an example of the described problem. We consider two groups, *Group 0* (G_0) and *Group 1* (G_1), where G_0 has a capacity equals to 3 over a quota period equal to 5, and a priority higher than G_1 . A G_1 thread (τ_0 in G_1) starts a request and will be served according to remaining budget for G_1 . Further, at the end of the first quota period (i.e., at t_5), a *Group 0* thread (τ_1 in G_0) starts to execute, with the limit timer armed to expire at

time t_A (i.e., current time, t_5 , plus the remaining capacity). After t_5 , a refill occurs. Due to the implementation, and assuming that the thread τ_1 in G_0 consumed 1 unit of capacity, the budget of *Group 0* passes from 3 to 2, but the limit timer is left untouched. At t_A , the timer expires and *Group 0* is scheduled out. Thus, in the second quota period, *Group 0* runs for a total of 2 units of time, even if its capacity is 3. We expect the *Group 0* to run for 3 units every 5 to resemble a deferrable server, and the timer armed to expire at t_B .



■ **Figure 4** SCHED_QUOTA problem. Dashed lines for group budget resemble the ideal behavior of a deferrable server, while solid lines are the real behavior as implemented in the proposed policy.

Moreover, according to limitation (L2) (see above), two threads belonging to different groups or other scheduling classes share the same range of priority levels within the shared queue. This would mix in time the execution of SCHED_QUOTA threads and other threads, nonetheless mixing up threads belonging to different groups with no clue of group priority, with no assurance about temporal isolation of different groups. This behavior of SCHED_QUOTA is inadequate to achieve a proper hierarchical scheduler. A solution that uses a user-level priority remapping would result in a restricted priority range, along with another problem (see limitation (L1) above): when a budget expires, the scheduler tries to pick every ready thread of that group in priority order, evaluating exhausted budget for each thread; since the number of threads in a group is not bounded, this operation has an $O(N)$ complexity, where N is the number of consecutive exhausted threads. Similar analysis keeps for refill function that moves expired threads to runqueue.

4.3 Implementation details

We implemented the proposed SCHED_DS policy by modifying the existing SCHED_QUOTA policy in Xenomai in order to obtain a truly hierarchical scheduler that solves the limitations described above. The Xenomai-based implementation is provided in [3]. The main modifications regard the functions `refill_handler`, which replenishes budgets every quota period, `xnsched_quota_enqueue/requeue/dequeue`, which handle the queue moving, and `xnsched_quota_pick`, which selects the thread to be executed. We created an additional level of scheduling with a runqueue for each group, in order to avoid the single common runqueue, and extending the group structure to encompass a priority level needed to create an order relation between groups to be queued. Furthermore, we created a fixed-priority FIFO runqueue that hosts runnable groups along with a list holding expired groups. We addressed the limitations mentioned in Subsection 4.2 as described in the following:

- **L1.** In the pathological situation formerly considered, the SCHED_DS policy will evaluate the budget of the whole group instead of evaluating for every single thread, moving only group structures if needed. These operations are $O(1)$ because the number of groups is upper-bounded by a kernel parameter (by default 32).

15:12 Achieving Industrial Edge MCS Isolation with RT-Containers

- **L2.** SCHED_DS policy isolates threads of different groups within the queue, since we do not use anymore the shared FIFO runqueue, allowing usage of the entire priority range without mixing threads of different groups.
- **L3.** SCHED_DS policy adds a boolean that is set when the refill timer expires and evaluated during `xnsched_quota_pick` execution, when a true value prevents the reduction of the budget of the formerly running quota group.

Compared to the `xnsched_quota_pick` implementation, we removed some unnecessary checks by improving performance and reducing overhead. Indeed, we allowed empty groups in the group runqueue, dynamically checking for this condition and removing them. This modification has the aim to relieve the following recurring pattern: a group with only one thread is scheduled, next the thread is removed from the group and after that, the group is removed from the queue. When rescheduling is triggered, the currently running thread is requeued as well as its group, causing a heavily ordered insertion, that could be potentially useless if the currently running thread is still the highest priority thread. According to the example shown in Figure 4, the boolean check of `budget_refilled` implies no budget reduction at the scheduling after the refill, the group then starts the new quota period with a full budget. Indeed, the check at line 18 of Algorithm 1 will fail, and the scheduler is obliged to arm once again the limit timer, which will now expire in t_B , in fulfillment of our requirement about the deferrable server behavior.

4.4 Feasibility Checker and `rt-lib`

We implemented a *feasibility checker* that parses the input and computes the tests introduced in Subsection 3.2. This consists of 600 LOC of Python code, which calls two C programs that use Xenomai services to bring up and tear down groups.

The *rt-lib* layer is implemented as a header file that relies on the `-wrap` linker flag to override the POSIX wrapper functions defined by Xenomai. Overridden functions belong to two distinct groups: one for hiding the complexity of thread management and one for the RTnet socket management. The first group of functions uses the original data structures to find and operate on the related extended ones. This exposes to the user a standard POSIX API that under the hood replaces real-time scheduling policies with SCHED_DS. On the other hand, networking functions ensure that a newly created socket calls an `ioctl` to bind to the allotted RTnet timeslot.

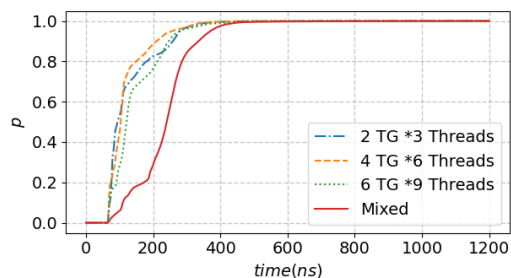
5 Experimental Results

In this section, we report the experimentation we performed on the implemented architecture. The experimentation aims to: *i*) measure scheduling policy runtimes; *ii*) estimate the overall throughput overhead as seen by the tasks to understand if the proposed algorithm can be practically used with negligible risk and to derive a suitable server period; *iii*) assess temporal isolation under several disturbances conditions, evaluating the solution in terms of number of failures, specifically, the *deadline misses* of the real-time tasks under non-real-time stress and faulty real-time container; *iv*) estimate the RTnet error deviation for sending packets through measurements achieved on a real network deployment; *v*) estimate the activation latency of the scheduler, comparing results with state-of-art solutions for real-time containers. In Table 1, there is a summary of experimental parameters used in the following tests.

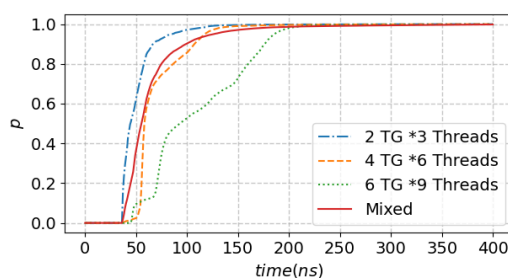
The host system is equipped with an Intel Core I5-6500, 16 GB DDR4 RAM, Samsung 970EVO SSD, running Linux Mint 20.1, kernel v5.4.77 patched with Xenomai v3.1, and Docker v20.10.7. To reduce hardware sources of non-determinism, we disabled power optimizations, frequency scaling, Intel SpeedStep, TurboBoost, and C-States.

■ **Table 1** Values and rationale behind parameters used in the experimentation.

Parameter	Ref. Sec.	Value	Rationale
Groups/Threads	5.1	2*3 up to 6*9	Understand how execution times vary with numbers of threads spread along an order of magnitude.
Runs	5.1	5	Sufficient to average out transient behaviors.
Repetitions	5.2	50	Computed by the sample size formula, to have 90% confidence.
Quota period	5.2	1 ms to 10 ms	Reasonable for the hardware used and DS runtimes obtained.
Target quota	5.2	40%	Sufficient runtime to show overhead effects.
Yielding period	5.2	0.1, 1, 10 ms	Different orders of magnitude, in the range below the quota period.
Duration	5.2	10 s	Enough to verify the overhead, on the base of the period.
U_{TOT} limit	5.3	85%	From [9], utilization > 90 % brings system to instability.
WCET buffer	5.3	30 μs	Max activation latency measured through the Xenomai official guide plus hardware unpredictability measured.
Additional band	5.3	1 %	Compensate scheduling overhead under stress.
Repetitions	5.3	30	Tradeoff between statistical significance and time needed.
Containers/tasks	5.3	2*2 up to 2*6	Reasonable numbers to find schedulable tasksets with small periods.
Task periods	5.3	1 ms up to 2 s	Reasonable real-time periods for a general purpose hardware.
Duration	5.3	60 s	Reasonable with regard to task periods.
TDMA cycle	5.4	6 ms	Reasonable time based on the roundtrip time measured.
Repetitions	5.5	60	Sample size formula for 90% confidence.
Sampling period	5.5	100 μs	Default period of the test.
Duration	5.5	60 s	Enough to experiment outliers due to kernels. Several orders of magnitude above the period.
Task WCET	5.6	1.8 ms	Avoid budget limitation of the policies to simplify analysis.
Task Period	5.6	10 ms	Reasonable for the WCET to have as many sampling as possible.



(a) CDF of runtimes of `xnsched_pick_next`.



(b) CDF of runtimes of `refill_handler`.

■ **Figure 5** CDF runtimes of SCHED_DS (TG = Thread Group).

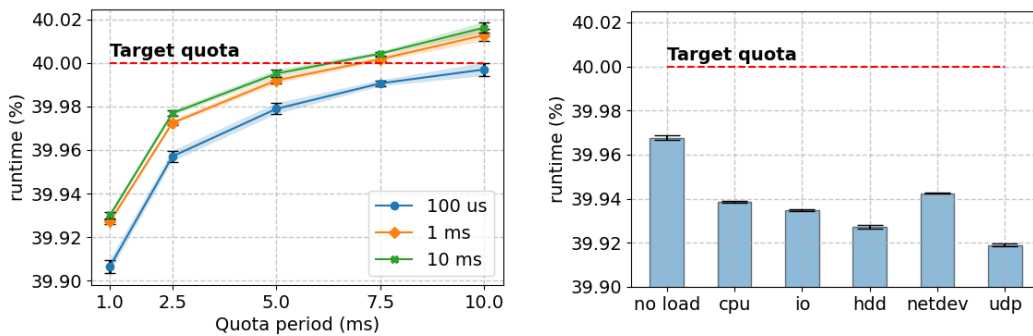
5.1 SCHED_DS Runtimes

We estimate the runtime of the main functions of the scheduler for a first quality evaluation. Both the `xnsched_pick_next` and the `refill_handler` runtimes were plotted, in order to understand which quota period could have been suitable for the system. The `xnsched_pick_next` checks the current thread status and, if needed, calls a `requeue` and a `sched_pick` for each scheduling policy in priority order, until a thread is found. The `refill_handler` was described previously in Section 3.3 and represents a periodic fixed overhead. Therefore, we can have an upper bound of enqueue, dequeue, and requeue functions, and an upper bound of the `xnsched_ds_pick` function, i.e., the one described in Algorithm 1. Runtimes are taken with a couple of clock monotonic timestamps, at the beginning and end of the functions under exam, under different workloads: 2 thread groups with 3 threads

each; 4 groups with 6 threads each; 6 groups with 9 threads; a mixed workload created as described in Section 5.3, with a random group number between 2 and 6, and a random thread number between 2 and 7 for each group. For the first three loads, two tasks for the group are permanently running in order to exhaust the budget at every period. The selected number of groups/threads have the only aim to understand how execution times vary with an increasing number of threads during nominal execution. Results in terms of CDF (averaged over 5 runs) are shown in Figures 5a and 5b. The workloads run on a single CPU and thread/group numbers have the only purpose to exercise the system with an increasing stress, to have an estimation of the overhead. A replenishment takes in the worst case 630 ns (Figure 5b), with a much lower average time, while the pick function has a worst case of 1,343 ns (Figure 5a), but it generally presents much lower runtimes: around 450 ns the CDF value is almost one.

5.2 SCHED_DS Overhead

In order to estimate the overhead induced by the proposed scheduler, we adapted the test for `SCHED_QUOTA` of the Xenomai suite. The test is an application that creates FIFO threads, which increment a counter and then yield in favor of another thread after a specified amount of time, called *yielding period* hereinafter. The obtained counter will be then used as a reference value. Next, a number of `SCHED_DS` threads, in the same amount as the FIFO threads, are created. The `SCHED_DS` threads belong to a group created with a determined quota, and have the same body as the FIFO ones. At the end of the application (after a timeout of 10 seconds), the `SCHED_DS` counters are compared to the FIFO reference. In this way, the effective percentage of the runtime of `SCHED_DS` threads can be computed. It should be noted that the results depend on the length of the quota period. The smaller period is used, the more overhead is induced and the smaller effective runtime percentage is computed at the end of the test. The `SCHED_DS` threads within the test are run with *1ms*, *2.5ms*, *5ms*, *7.5ms*, *10ms* *quota periods*, assigning a target quota equal to 40%. Moreover, we experiment *100 μs*, *1ms*, and *10ms* as yielding periods for both `SCHED_FIFO` and `SCHED_DS` threads. For each test, we run 12 threads (FIFO threads to compute the reference values and `SCHED_DS` threads to compute the overhead), with a random activation phase with regard to the server period to reduce anomalies that depend the first and last period execution. Each test is repeated 50 times for statistical significance purposes. Afterward, the average of the runtimes percentage, fixing the same quota period and yielding period, are computed and plotted with 90% of confidence.



(a) Varying quota and yielding periods.

(b) Under stress scenarios.

■ **Figure 6** Runtime percentages of running `SCHED_DS` threads.

Figure 6a shows the obtained runtime results by varying both quota and yielding periods, along with the quota target percentage, i.e., 40% indicated by the dotted red line. It can be noted that when the quota period is $1ms$ the induced overhead is approximately 0.05 – 0.1%, while with a quota period of $2.5ms$ the estimated overhead is lower, i.e., between 0.02 and 0.05%. Differently, with a quota period equal to $10ms$ the results show a runtime percentage greater than 40%. This is probably due to both the thread phasing with regard to the beginning of the first period and the deferrable server behavior: indeed, threads can benefit from the entire runtime budget over a reduced quota period because of late arrival. For lower yielding periods, the overhead is greater, as expected. Furthermore, we highlight that the runtime percentage difference between the yielding period of $100\mu s$ compared to both $1ms$ and $10ms$ is clear since at $1ms$ and $10ms$ the scheduling policy preemption dominates the number of yielding preemptions. According to the results obtained for the overhead, we set the quota period for the subsequent test equal to $2.5ms$ since it represents a good choice for the schedulability compared to the other periods tested.

We compute also the overhead of SCHED_DS under various stress conditions. The disturbances are generated via the `stress-ng` tool [22]. We impose hard pinning for the CPU core that executes the stress load by using the `-taskset` flag. We test different stress load scenarios, i.e., *no load* (tests executed with no load), *cpu* (a heavy arithmetic computation to fully use one CPU core), *io* (an IO load using one worker spinning on `sync()` command to force writes data buffered in memory out to disk), *hdd* (a disk load that starts one worker generating various operations towards the disk), *netdev* (a network load where a worker exercises various netdevice ioctl commands for all available network interface controllers) and *udp* (a network load starting a pair of client and server workers that transmits data using UDP on localhost). Regarding stress tests, we still perform 50 repetitions for statistical significance purposes. The quota period is set equal to $2.5ms$ and the yielding period equal to $1ms$. Figure 6b shows the obtained results, which highlight that scheduling efficiency slightly decreases under stress conditions. The results under stress conditions compared to the *no load* ones are approximately 0.03 – 0.04% lower, with *udp* and *hdd* as worst loads because of the high volume of interrupts generated.

5.3 Failure Isolation Test

In this section, we perform tests for the proposed scheduling policy for revealing potential temporal isolation issues. The tests include, first, the scheduler correctness (whether it behaves as expected). Further, we want to obtain real performances in the field of SCHED_DS under synthetic workloads, against several stress conditions. We demonstrate that the provided proactive temporal isolation successfully prevents failures (deadline misses) propagation within the system, crossing criticality levels. Specifically, the test assumes two criticality levels, thus including two real-time containers: one container with a high priority and the other one with a low priority.

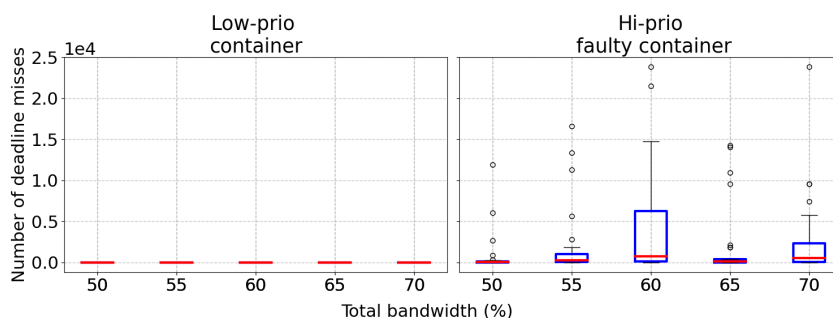
The total bandwidth U_{TOT} is spread over the two containers using the *RandFixedSum* algorithm [21] (we used the Python implementation [23]), such that $U_{TOT} = U_{LOW} + U_{HI}$, where U_{LOW} and U_{HI} are the bandwidths of low-priority and high-priority rt-containers, respectively. For both containers a group of tasks is created through the *RandFixedSum* algorithm as well, such that the tasks bandwidth for each container is $U_j = \sum_{i=0}^{N_j-1} U_i^j$, where $j \in \{LOW, HI\}$ and N_j is the number of tasks of the container j . The periods used as input for the algorithm range in likely times for real-time tasks (ms up to s). Priorities are assigned to the two task groups according to the order of creation, while the algorithm in [16] and Equations 1, 2, 3 are used to determine the minimum bandwidth needed for the containers to have a schedulable task set.

Tasks within a task group are ordered following Rate Monotonic (RM) rules [34]. If the sum of required bandwidth needed to have a schedulable task set is greater than 85%, the current task set is discarded, and a new one is generated. The required bandwidth of each container is increased by 1% to compensate for the scheduling overhead. In the algorithm, a little margin of 30 μs is added to WCET of each task to consider maximum Xenomai scheduling latency plus a margin due to hardware unpredictability. Moreover, for each task, there is a lower threshold WCET of 30 μs that allows avoiding borderline situations, e.g., workloads too short to be properly emulated on x86 architectures. For the same reason, each group must have a total utilization greater than 0.1% quota.

Once created a task set with the given constraints and computed the needed bandwidth for each container, quota groups are created, and synthetic load is created through the *rt-app* [10], exploiting the *CPU property* to pin tasks on a CPU core, the *run property* to create a workload of a determined time, the scheduler policy and absolute timer properties to select the `SCHED_DS` policy and generate a periodic timer. The *rt-app* has been modified and recompiled for our purposes, linking real-time libraries. Each thread logs some information for each iteration in a pre-allocated memory area. The slack time, defined as next activation time minus current finish time, can be used to compute deadline miss occurrences; indeed, a negative slack means that the task has finished after the start of the next period.

Tests are run with varying total bandwidth U_{TOT} : 50, 55, 60, 65, 70 percent of the CPU usage. In each container, there is a random number of threads, chosen between 2 and 6 (excluded). 30 repetitions are done for each combination of total bandwidth and stress condition. Stress conditions considered in this test include a scenario, namely *low-hi*, in which the low-priority container behaves as declared, while the high-priority container is faulty, i.e., it executes tasks with a runtime of 1.8 times than declared. This scenario aims to detect if quota groups are correctly isolated in time: the high-priority container is faulty since has an actual load almost twice the declared. The desired behavior is that the lower priority container will be isolated from the faulty container. Thus, we expect deadline misses in the high-priority container but no deadline miss in the low-priority container, which could be potentially more critical than the high-priority one. The other tested stress scenarios are the same as the ones described in Section 5.2. These scenarios aim to understand the behavior of the system under various non-real-time workloads. We expect to have no deadline miss at all. Each experiment lasts 60 seconds. Task periods are sampled with a *loguniform* distribution, with a granularity of 1 ms, with a minimum period of 1 ms, and a maximum period of 2 s. As expected, no failures came out from experiments under non-real-time stress-ng load. It is worth mentioning that, although the server period is 2,500 μs , the schedulability test can guarantee shorter period tasks as well the time to execute, and despite the needed bandwidth is quite high, several tasks with a period of 1 ms have been scheduled in the low-priority container. We tested the 1 ms quota period and their schedulability is a lot easier. The results meet our expectations under the *low-hi* scenario as well. The high-priority container presents a high number of failures (in terms of deadline misses), while the low-priority container achieves the correct execution.

Figure 7 shows the boxplots of the deadline misses in the *low-hi* scenario. Each point represents the sum of the number of deadlines misses from all tasks in a container, while on the x-axis there is the total bandwidth used to generate the task set. It is worth noting that some tests run with no deadline miss: this is likely due to the extra bandwidth, or the pessimistic behavior of the schedulability test. The test seems to have quite encouraging results, as `SCHED_DS` can resist several high-stress conditions and temporal isolation is assured.



■ **Figure 7** Boxplots of deadline misses for the *low-hi* scenario across different usage percentage of total bandwidth U_{TOT} .

5.4 RTnet

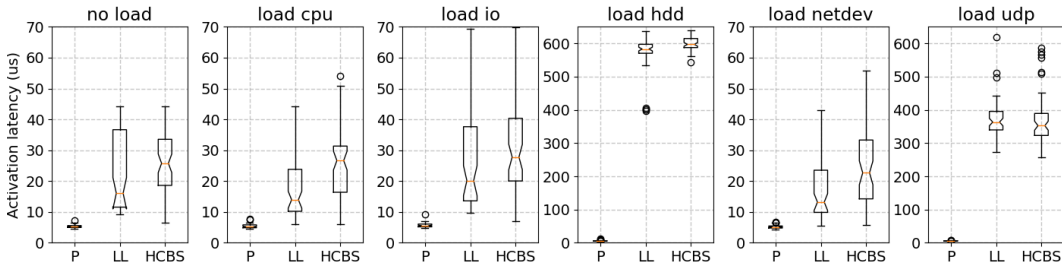
In this section, we perform tests to show the real-time capabilities of the proposed RTnet wrapper used within containers and in presence of non-real-time traffic disturbances. The aim is to highlight that the RTnet stack is characterized by a low average and standard deviation of the sending delay (defined as the difference between the expected sending time and the actual one) even under non-real-time tunneled traffic. Tests were run on two HP z230 workstations, equipped with an Intel Core i7-4790, 16 GB RAM, and an i217-LM network controller with `rt_e1000e` driver. NICs were linked to a 100 MBit/s switch through cat6 Ethernet cables. The machines hosted Ubuntu Server 21.04, with Linux kernel v5.4.77 patched with Xenomai v3.1.1.

The TDMA cycle was set to 6 ms, with slots of 1 ms. The transmission delay of the *sync frame* from the setup was estimated at around $112 \mu s$ (computed as described in [33]). We start a couple of containers on the slave node, i.e., the *RT-Server1* in Figure 3, in which they spawn one periodic thread each, with coprime periods to force coincidence for network transmission within the same TDMA cycle. These tasks send a UDP packet towards the master, i.e., the *Master* node in Figure 3, for each period. Further, at the host level on the *RT-Server1*, we used the *socat* tool to send, continuously, a file in UDP broadcast, in order to generate non-real-time traffic. These UDP packets are automatically divided into fragments by the RTnet stack to fit the slot reserved for non-real-time traffic. We recorded the actual sending timestamp and scheduled sending timestamp for the *sync frames*, and sending timestamps for slave packets. The slave properly used the estimated transmission delay to adjust the slot starting times [33]: the sending offset of the outgoing packets with regard to the arrival time of *sync frames* were not integer multiples of slot durations, and $112 \mu s$ was the offset that minimized the sum of squared errors, defined as the difference of time to the nearest multiple of slot duration from sync frame. The mean, minimum, maximum, and standard deviation of the difference between scheduled sending timestamp and actual sending timestamp for *sync frames* were respectively 385.8, 257, 1,499, and 103.2 ns. On the other hand, keeping into account the correction of $112 \mu s$, the slave errors for sending frames were characterized by an average of 211.8 ns and a standard deviation of 1,006.7 ns, showing good predictability with regard to a non-real-time RTnet stack.

5.5 Task Activation Latencies

In this section, we aim to estimate the tasks activation latencies provided by our solution and use them to compare the proposal with the following alternative solutions for real-time containers based on hierarchical scheduling: *i*) Linux low-latency vanilla `rt-cgroups` (LL

hereinafter) and *ii*) *rt-cgroups* patched as described in [1] (HCBS hereinafter). Kernels in LL and HCBS are in low-latency configuration due to reasons explained in Section 2. Despite we set our solution as complementary (i.e., designed for usage in different use cases), we compare our implementation with LL and HCBS since they are the only ones to date based on group scheduling. The vanilla *rt-cgroups* allow dividing CPU time, specifying how much time can be spent running in a given period. The *runtime* is allocated to each real-time group (*rt-group*), and other groups will not be allowed to use it. The time not allocated to a *rt-group* is used to run `SCHED_OTHER` tasks. Currently, this feature still lacks an EDF scheduler to use non-uniform periods. The low-latency kernel version is the same as Xenomai (v5.4.77), while HCBS supports an older kernel version (i.e., v5.2.8). Kernel settings are almost the same, the only differences between the configurations are constrained by patches. The latency test provided by Xenomai test suite is modified to run with `SCHED_FIFO` for Linux, making a lift and shift of data structures used, creating an independent program. We further adapted this latency test to support `SCHED_DS` threads.



■ **Figure 8** Boxplots of task activation latencies. In *no load* scenario, for HCBS, we removed two outliers at 114 and 118 μs for readability. P is the proposed solution.

We run the latency test for 60 seconds for each targeted solutions (i.e., the proposed one, *LL*, and *HCBS*) according to stress load scenarios described in Section 5.2. For each test, we saved the maximum latency and the overruns. We repeated each test 60 times for statistical significance purposes. The task period is left as the default value used in the original Xenomai latency test, i.e., 100 μs .

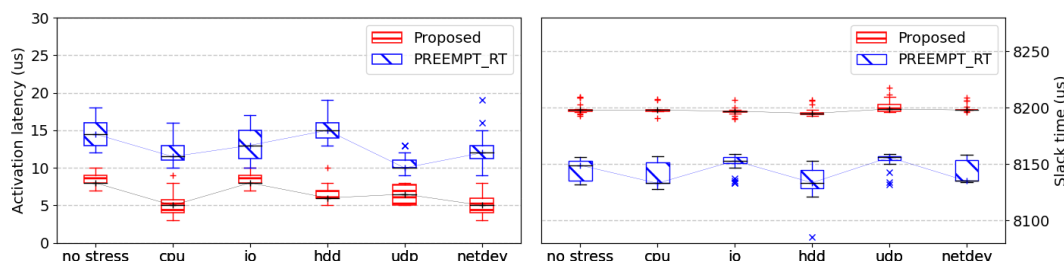
Figure 8 shows boxplots of the latency experienced by the analyzed solutions under different disturbances, while Table 2 provides the mean and standard deviation of the obtained maximum latencies. It can be noted that *No stress*, *cpu*, *io*, and *netdev* loads provide similar results. The proposal seems to be robust against all the disturbances, giving persistently lower latencies ($\sim 10 \mu\text{s}$). Under *udp* and *hdd* loads, both the LL and HCBS provide higher latencies, with a mean of the maximum latencies about 60 times worse than the proposed solution and a non-negligible standard deviation. We expected this behavior since both *udp* and *hdd* loads raise a high volume of interrupts, which heavily affect kernel performance despite the low-latency configuration of the Linux kernel. Our co-kernel-based implementation is not affected significantly by such high interrupts load, making it a good fit for critical high-frequency tasks for industrial control. Finally, we tried to switch the SSD disk with HDD one. We noticed a non-negligible difference under the *hdd* load, concluding that latencies are mostly caused by drivers, which probably present non-preemptible sections. For the sake of brevity, we do not report this experiment. The difference between *LL* and *HCBS* is not significant, thus the main reason behind high latencies is not the hierarchical scheduler itself, but the preemption model of Linux. On the other hand, Xenomai makes Linux fully preemptible, thus Xenomai outperforms other kernels.

■ **Table 2** Maximum task activation latency.

Load	Proposed		LL		HCBS	
	Mean [μ s]	Std	Mean [μ s]	Std	Mean [μ s]	Std
no load	5.332	0.509	22.334	12.075	28.396	18.464
cpu	5.367	0.665	18.477	10.743	25.795	11.813
io	5.695	0.726	25.016	13.083	30.771	14.328
hdd	5.983	1.061	572.969	50.047	599.308	19.923
netdev	5.231	0.600	17.718	10.643	25.082	14.313
udp	5.879	0.253	369.252	58.109	368.706	73.646

5.6 Comparison with PREEMPT_RT

Since the low-latency configuration is not able to always guarantee timeliness, in this section we directly compare Xenomai against PREEMPT_RT. The PREEMPT_RT patch has recently solved the conflict with `rt-cgroups`, from kernel v5.15.34. However, we highlight that without the HCBS patch, the `rt-cgroup` scheduler does not fit any theoretical model for schedulability. We ran `rt-app` in a container for 60 seconds, sampling the worst task activation latency and slack time along the minute for each of the stress loads considered in the previous test. The slack time is defined as $d - a - C$, where d is the task deadline, a is the arrival time, and C its WCET. We repeated each test 60 times for statistical significance purposes. The `rt-app` generates a single thread with a period of 10 ms and a runtime of 1.8 ms, to avoid budget limitation of the scheduling policies. Obtained results are shown in Figure 9.



■ **Figure 9** Boxplots of both worst task activation latencies (lower is better) and slack times (higher is better) under different stress loads.

Activation latencies, although comparable, are always lower for Xenomai under each stress, with an average improvement of at least 30% in the case of the `udp` load, and above 50% for the `hdd` load, which is still particularly troublesome for Linux. The lower latencies with regard the low-latency configuration confirm our statement that high latencies in the previous experiments are due to non-preemptible sections. Even the slack time for the proposed system is closer to the expected (i.e., 8200μ s). Moreover, the slack time of our solution presents a lower standard deviation with regard to PREEMPT_RT, probably due to a lower predictability and higher the complexity of Linux, along its difficulty to handle workloads. Once again, under `hdd` load, Linux presents the worst outlier.

6 Conclusion

In this work, we introduced a novel architecture for real-time containers by leveraging a hierarchical deferrable server scheduler in a real-time co-kernel. We integrated the solution with a real-time networking stack (i.e., RTnet) for communication purposes, and provided the schedulability test and a user-level APIs to deploy the containers. The proposed architecture has been implemented extending the Xenomai co-kernel; the source code has

been made publicly available. Extensive experimental tests have been performed, showing that the proposed solution is promising in terms of latency, overhead, and, most importantly, isolation against disturbances. Specifically, a low-priority container can resist against severe misbehavior of a high-priority container, with no impact in terms of timing failures (i.e., no deadline miss). Further, comparing the proposed architecture with solutions in the state of the art, we obtain benefits in terms of task activation latencies. Future works aim at supporting rt-containers in the context of container orchestration platforms (e.g., Kubernetes), in order to create a fully-automated mixed-criticality industrial edge/cloud solution.

References

- 1 L. Abeni, A. Balsini, and T. Cucinotta. Container-based real-time scheduling in the linux kernel. *SIGBED Rev.*, 16(3):33–38, November 2019.
- 2 Amazon Inc. Getting started with cloud-native automotive software development. URL: <https://catalog.us-east-1.prod.workshops.aws/v2/workshops/12f31c93-5926-4477-996c-d47f4524905d/en-US>. Accessed 17th June 2022.
- 3 Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Xeno-containers, GitLab repo. <https://dessert.unina.it:8088/marcobarlo/xeno-containers>.
- 4 Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *Proc. ZEUS*, pages 65–73, 2021.
- 5 A. Burns and R. I. Davis. Mixed Criticality Systems – a review. *Tech Rep of the University of York*, 2018. URL: <https://www-users.cs.york.ac.uk/burns/review.pdf>.
- 6 Felipe Cerqueira and Björn Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual workshop on operating systems platforms for embedded real-time applications*, pages 19–29. SYSGO AG, 2013.
- 7 Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Elsevier Future Generation Computer Systems*, 2021.
- 8 Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Eur-omicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 9 Marcello Cinque, Raffaele Della Corte, and Roberto Ruggiero. Preventing timing failures in mixed-criticality clouds with dynamic real-time containers. In *2021 17th European Dependable Computing Conference (EDCC)*, pages 17–24. IEEE, 2021.
- 10 Multiple contributors. Scheduler tools /rt-app. <https://github.com/scheduler-tools/rt-app>. Accessed 17th June 2022.
- 11 Breno Costa, Joao Bachiega Jr, Leonardo Rebouças de Carvalho, and Aleteia PF Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 55(2):1–34, 2022.
- 12 Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in openstack for nfv services. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.
- 13 N. T. Dantam et al. The ach library: A new framework for real-time communication. *IEEE Robotics Automation Magazine*, 22(1):76–85, 2015.
- 14 Neil T Dantam, Daniel M Lofaro, Ayonga Hereid, Paul Y Oh, Aaron D Ames, and Mike Stilman. The ach library: A new framework for real-time communication. *IEEE Robotics & Automation Magazine*, 22(1):76–85, 2015.
- 15 R.I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–398, 2005. doi:10.1109/RTSS.2005.25.

- 16 Rob Davis and Alan Burns. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.
- 17 Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso Cucinotta. Demystifying the real-time linux scheduling latency. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 18 Raimarius Delgado, Bum-Jae You, and Byoung Wook Choi. Real-time control architecture based on xenomai using ros packages for a service robot. *Journal of Systems and Software*, 151:8–19, 2019.
- 19 Zhong Deng and JW-S Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium*, pages 308–319. IEEE, 1997.
- 20 Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM). RTAI - the RealTime Application Interface for Linux. <https://www.rtai.org/>. Accessed 17th June 2022.
- 21 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*, January 2010.
- 22 Colin Ian King et al. Stress-ng GitHub repository. <https://github.com/ColinIanKing/stress-ng>. Accessed 17th June 2022.
- 23 Cucinotta et al. Taskset generator. https://gitlab.retis.santannapisa.it/t.cucinotta/rtsim/-/blob/master/src/taskset_generator/taskgen.py. Accessed 17th June 2022.
- 24 Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. Rt-kubernetes—containerized real-time cloud computing. In *37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*, 2022.
- 25 The Linux Foundation. Know Limitations of PREEMPT_RT patch. https://wiki.linuxfoundation.org/realtime/documentation/known_limitations. Accessed 17th June 2022.
- 26 C. Garre, D. Mundo, M. Gubitosa, and A. Toso. Real-time and real-fast performance of general-purpose and real-time operating systems in multithreaded physical simulation of complex mechanical systems. *Mathematical Problems in Engineering*, 2014, 2014.
- 27 Carlos Garre, Domenico Mundo, Marco Gubitosa, and Alessandro Toso. Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost. Technical report, SAE Technical Paper, 2014.
- 28 Tom Goethals, Filip De Turck, and Bruno Volckaert. Fledge: Kubernetes compatible container orchestration on low-resource edge devices. In *International Conference on Internet of Vehicles*, pages 174–189. Springer, 2019.
- 29 T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner. Container-based architecture for flexible industrial control applications. *Journal of Systems Architecture*, 84:28–36, 2018.
- 30 Thakor Bhishmapalsinh Jitendrasinh and Shripad Deshpande. Implementation of can bus protocol on xenomai rtos on arm platform for industrial automation. In *2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)*, pages 165–169. IEEE, 2016.
- 31 Kuljeet Kaur, Sahil Garg, Gagangeet Singh Aujla, Neeraj Kumar, Joel JPC Rodrigues, and Mohsen Guizani. Edge computing in the industrial internet of things environment: Software-defined-networks-based edge-cloud interplay. *IEEE communications magazine*, 56(2):44–51, 2018.
- 32 Jan Kiszka. Xenomai Homepage. URL: <https://source.denx.de/Xenomai/xenomai/-/wikis/home>. Accessed 17th June 2022.
- 33 Jan Kiszka and Bernardo Wagner. Rtnet-a flexible hard real-time networking framework. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 8–pp. IEEE, 2005.
- 34 John Lehoczky, Lui Sha, and Yuqin Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, volume 89, pages 166–171, 1989.
- 35 Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

- 36 C.-N. Mao et al. Minimizing latency of real-time container cloud for software radio access networks. In *IEEE 7th International Conference on Cloud Computing Technology and Science*, pages 611–616, 2015.
- 37 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 38 Philip Masek, Magnus Thulin, Hugo Sica de Andrade, Christian Berger, and Ola Benderius. Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle. *CoRR*, abs/1608.06759, 2016. [arXiv:1608.06759](https://arxiv.org/abs/1608.06759).
- 39 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.
- 40 Paul Menage. Cgroups. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>. Accessed 17th June 2022.
- 41 Hassan Ghasemzadeh Mohammadi, Rahil Arshad, Sneha Rautmare, Suraj Manjunatha, Maurice Kuschel, Felix Paul Jentzsch, Marco Platzner, Alexander Boschmann, and Dirk Schollbach. DeepWind: An Accurate Wind Turbine Condition Monitoring Framework via Deep Learning on Embedded Platforms. In *Proc. ETFA*, volume 1, pages 1431–1434, 2020.
- 42 M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. *10th IEEE International Conference on Computer and Information Technology, Bradford, pp. 1864-1871*, 2010.
- 43 Harald Mueller, Spyridon V. Gogouvtis, Andreas Seitz, and Bernd Bruegge. Seamless computing for industrial systems spanning cloud and edge. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 209–216, 2017.
- 44 Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, and Neeraj Suri. Mitigating timing error propagation in mixed-criticality automotive systems. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 102–109. IEEE, 2015.
- 45 Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Computing Surveys (CSUR)*, 52(1):1–36, 2019.
- 46 Siemens AG. Jailhouse hypervisor source code. URL: <https://github.com/siemens/jailhouse>.
- 47 José Simó, Patricia Balbastre, Juan Francisco Blanes, José-Luis Poza-Luján, and Ana Guasque. The role of mixed criticality technology in industry 4.0. *Electronics*, 10(3):226, 2021.
- 48 V. Struhár et al. Real-Time Containers: A Survey. In *2nd Workshop on Fog Computing and the IoT*, volume 80 of *OpenAccess Series in Informatics*, pages 7:1–7:9, Dagstuhl, Germany, 2020.
- 49 Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. React: Enabling real-time container orchestration. In *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2021.
- 50 T. Tasci, J. Melcher, and A. Verl. A container-based architecture for real-time control applications. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–9, 2018.
- 51 Lane Thames and Dirk Schaefer. Software-defined cloud manufacturing for industry 4.0. *Procedia cirp*, 52:12–17, 2016.
- 52 The Linux Foundation. Homepage of LF Edge Foundation. <https://elisa.tech/>.
- 53 The Linux Foundation. Kubernetes Home Page. <https://kubernetes.io/>.
- 54 The Linux Foundation. Real-time group scheduling. <https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt>. Accessed 17th June 2022.
- 55 Wind River Systems, Inc. WindRiver VxWorks Virtualization Profile. <http://www.windriver.com/products/vxworks/technology-profiles/#virtualization>. Accessed 17th June 2022.

- 56 Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2014.
- 57 Xilinx. RunX GitHub repository. <https://github.com/Xilinx/runx>.
- 58 Chengjing Yu, Xudong Ma, Fang Fang, Kun Qian, Shun Yao, and Yanping Zou. Design of controller system for industrial robot based on rtos xenomai. In *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 221–226. IEEE, 2017.
- 59 Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1270–1278. IEEE, 2019.

Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds

Reza Mirosanlou ✉

University of Waterloo, Canada

Mohamed Hassan ✉

McMaster University, Hamilton, Canada

Rodolfo Pellizzoni ✉

University of Waterloo, Canada

Abstract

In Commercial-Off-The-Shelf (COTS) systems-on-chip, processing elements communicate data through a shared memory hierarchy, and a coherent high-performance interconnect, where the de facto standard to handle shared data is through a coherence protocol. Driven by the extraordinary demands from modern real-time embedded system applications to generate, process, and communicate massive amounts of data, recent efforts aim to ensure timing predictability while integrating cache coherence in multi-core real-time systems. However, we observe that most of these efforts compromise system average performance upon offering predictability guarantees. Motivated by this observation, this work proposes an arbiter aimed at providing a predictable, coherent shared cache hierarchy solution, yet with a negligible performance degradation compared to COTS solutions. We achieve this goal by adopting a high-performance-driven architecture including a split-transaction bus and bankized shared cache. In addition, all accesses are arbitrated through a global ordering mechanism. Our proposed arbiter operates alongside conventional coherence protocols without requiring any protocol modifications. Furthermore, we leverage the *Duetto* reference model by pairing the proposed arbiter and a high-performance arbiter. We evaluate our solution based on both synthetic and SPLASH-3 benchmarks, showing that we can significantly outperform the state-of-the-art in predictable cache coherence, while offering a COTS-level performance.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Embedded hardware

Keywords and phrases Predictability, Cache, COTS, Arbitration, Real-time system

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.16

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback, and our shepherd for helping to significantly improve this paper. This work has been supported in part by NSERC, CMC Microsystems, and TII. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

1 Introduction

Enabling data sharing is imperative in modern embedded systems—automotive, Unmanned Air Vehicles (UAVs), and Internet-of-Things (IoT) to name a few. In these systems, massive amounts of data have to be collected (sensor fusion, cameras, etc), communicated through interconnect(s), and processed by various processing elements. As a result, recent efforts have been proposed to shift away from the independent task model, where tasks do not share data to a more-practical model that embraces data sharing and enables inter-core communication [5, 4, 21, 10, 33, 6, 34, 17]. Among these solutions, we find those leveraging cache-coherent interconnects to be promising due to their performance benefits as well as transparency to the software stack. In addition, cache coherence is already the standard de facto in Commercial-Off-The-Shelf (COTS) multi-core platforms.



© Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni;
licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 16; pp. 16:1–16:27



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, real-time embedded systems impose their own unique challenges, which do not exist in these performance-oriented COTS platforms. Predictability comes at the top of the list of these challenges; the architecture has to be predictable by-design to facilitate the timing analysis, which is necessary to provide timing guarantees to tasks running on the system. Originally designed with performance as the main goal, COTS cache coherent interconnects deploy several re-orderings and optimizations that hinder their predictability. It has been shown that even deploying a simple COTS coherence protocol such as the Modified-Shared-Invalid (MSI) protocol on top of a Time Division Multiplexing (TDM)-based interconnect revokes the system predictability [10, 16].

1.1 Related Work: Predictable Cache Coherence

To address this problem, recently the community has proposed several works that aim at implementing predictable cache coherence solutions. However, most existing solutions impose both coherence protocol as well as architectural modifications [10, 37, 18, 16, 17] or at the very least require specific hardware support [9]. These changes have led in early works to a quadratic increase in the worst-case memory latency (WCL) [10, 37, 18] (*Problem 1*). Moreover, mandating coherence protocol modifications discourages a real adoption of these solutions from the industry since adoption and verification of a new coherence protocol is known to be one of the most complex architectural tasks [36, 30] (*Problem 2*). PISCOT [15] addresses these problems by deploying of COTS coherence protocols on split-transaction interconnects. It also reduces the quadratic worst-case coherence latency to be linear in the number of cores. Nonetheless, PISCOT achieves this tight WCL by deploying two techniques that limit the overall memory performance compared to COTS solutions. The first is a TDM-based request bus, which is needed to enforce predictability, and the second is limiting the number of requests that each core can issue to the interconnect to one, which is needed to achieve the aforementioned tight latency (*Problem 3*). Additionally, similar to all existing work, PISCOT models the data bus and the last-level cache (LLC) as a single shared resource, and hence, no parallelism is possible in accessing the LLC (*Problem 4*). In COTS platforms, since bank processing times are much longer than the data transfer on the bus, LLC is usually a bankized memory, where different banks can process requests in parallel to improve system’s performance [2].

1.2 Contributions

Motivated by these limitations, this paper makes the following contributions: 1) We propose DUEPCO: a novel real-time arbitration scheme for managing memory accesses in the cache hierarchy. This arbiter models the cache hierarchy as independent and parallel resources: the request (control) bus, the response (data) bus, while each LLC’s bank is a resource of its own. This is key to leverage parallelism among these components to improve average performance, while tightening memory latency bounds (addressing *1* and *4*). More details about this arbiter are in Section 4. 2) Our proposed arbiter operates alongside conventional coherence protocols without requiring any protocol modifications (addressing *2*). 3) In Section 5, we provide a timing analysis that ensures predictability by statically bounding the worst-case latency suffered by any memory request. Unlike the solutions in [10, 16, 18, 37], and similar to [15, 9, 17], this bound is linear in the number of cores (addressing *1*). 4) To further address the performance-predictability trade-off, in Section 6 we show how to extend the Duetto reference model [25, 26] to our cache architecture. This is achieved by integrating two arbiters: a High-performance Arbiter (HPA) offers the system a COTS-level performance

most of the time, while the proposed Real-time Arbiter (RTA) runs in parallel and is only utilized when necessary to meet timing guarantees (addressing ❸). 5) Finally, in Section 7 we evaluate the proposed arbiter against the state-of-the-art predictable coherency solution as well as a baseline COTS solution using both synthetic and SPLASH-3 [31] benchmarks. Our evaluation shows that our arbiter outperforms state-of-the-art predictable solutions in terms of average memory latency by an average of $1.74\times$, while providing comparable worst-case latency bounds to the best predictable mechanism. Employing Duetto can further improve system throughput by up to $6.4\times$ at the cost of some degradation in latency bounds.

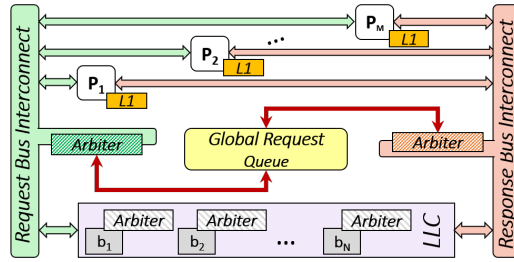
2 Background

2.1 Hardware Cache Coherence

Cache coherence is ubiquitous in shared memory multiprocessors since it enables shared data communication between cores while ensuring the system maintains data correctness. Data correctness is achieved when all cores consume (upon request by load/store instructions) the most recent copy of data. Coherence protocols employed in current Commercial-off-the-Shelf (COTS) systems are extensions of the basic Modified-Shared-Invalid (MSI) protocol [36]. MSI consists of three fundamental stable states: 1) **Modified (M)**: corresponds to memory blocks that have been modified (dirty) by a write in a private/shared cache; hence, the core/LLC is the *owner* of this cache line; 2) **Shared (S)**: corresponds to the blocks that are unmodified (clean) and held by one or more cores; 3) **Invalid (I)**: contains potentially stale and incoherent datum and both loads and stores will miss when accessing invalid blocks. Note that the protocol allows multiple cores to have a cache line in the **S** state, while only one core could have a cache line in the **M** state. According to the observed memory activity, the state of the cache line copies will be changed in the cache controllers. There exist two types of hardware cache coherence mechanisms based on how cores and the shared memory perceive the memory activity: 1) snooping bus-based cache coherence [36] in which all cores broadcast all the memory activities on the bus; 2) directory-based cache coherence [38] in which every activity will be communicated through a centralized directory that tracks the information regarding the cache lines among all cores. In this work, we employ snooping bus-based coherence as they are normally deployed in multi-core systems with up to 16 cores [32, 3, 27] and also deployed in state-of-the-art efforts [15, 10, 17, 18].

2.2 Arbitration

Simultaneous access to physical shared resources such as shared bus have a significant effect on the execution time of the applications. Therefore, having an arbiter is necessary when multiple cores try to access the shared resource simultaneously. Similar to the other shared resources in the system, different arbitration schemes lead to different timing characterizations of the system [7, 28, 11, 40, 17, 15, 10, 8, 7, 24, 29, 23]. The memory bus in multicore systems is one of the primary sources of interference. Therefore, a predictable arbiter guarantees that each requesting core is granted the bus eventually in a defined upper-bound amount of time. In COTS platforms, a high-performance arbiter is commonly used to maximize the overall performance. First-Come-First-Serve (FCFS) [19] is one example in this context that does not provide any latency guarantee for the shared memory accesses (assuming that there is no bound on reordering requests) and favors the cores that generate more requests to the shared resource and operate faster than the other cores in the system. On the other hand,



■ **Figure 1** Architecture model.

this could potentially lead to a case where a request from a slower core takes a very long time to get service. Another approach is to assign a fixed priority to a certain processor but this type of arbitration policy cannot provide a guarantee to lower priority requests.

The level of complexity of the coherence protocols heavily relies on the underlying deployed interconnect network architecture. For instance, atomic/unified buses significantly simplify the protocol implementation; however, the performance of the system will be significantly degraded as all the cores are required to wait until the granted core finishes its interconnect usage; hence, serializing the accesses. In COTS platforms such as ARM Corelink CCI550 and Intel’s QPI, the shared bus is usually implemented as a split-transaction interconnect in order to improve system performance. This is achieved by concurrently managing both coherent messages and data responses [35] such that the request bus and response bus will be separated. This architecture allows pipelining operation for the requests and responses on the bus and increases the flexibility in terms of the arbitration. In detail, a split-transaction bus can provide responses in an order different from the request bus depending on the arbitration on request and response buses.

3 System Model

In this section, we first detail the hardware architecture considered in this paper along with the coherency assumptions. Then, we explain how requests generated by the cores are processed by the proposed hardware architecture and how the latency for each request is constructed.

3.1 Architecture and Coherency

An overview of the proposed hardware architectural model is delineated in Figure 1. We consider a multi-core system with M Out-Of-Order (OOO) requestors¹ including processing cores, $P_1, \dots, P_i, \dots, P_M$ where each requestor has exclusive access to a private cache. All cores have also access to a shared memory that we assume is an on-chip Last-Level Cache (LLC). We assume that tasks running on the cores can share data among each other; hence, a coherency protocol must be employed in the system to allow coherent actions among cores and LLC. We consider both data transfers between a private cache and the LLC, as well as direct Cache-to-Cache (C2C) transfers between private caches, which exhibit improved average-case performance. In this paper, we adopt the MSI coherency protocol that includes three fundamental stable states as discussed in Section 2. Notice that we use the MSI as an

¹ We use cores and requestors interchangeably throughout the paper.

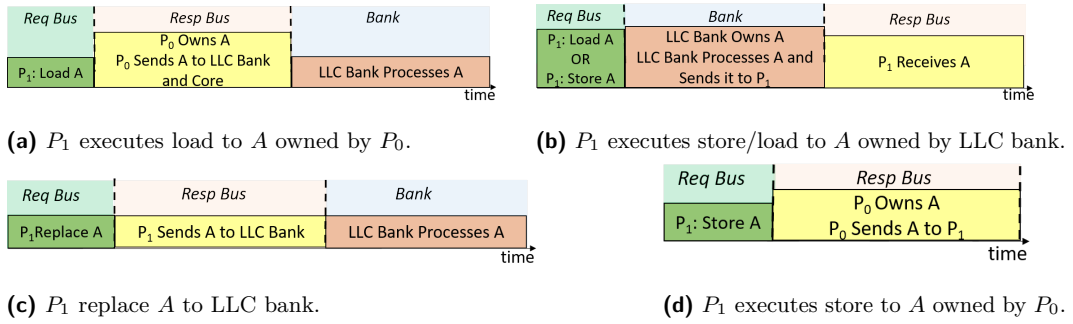
exemplar protocol; however, the proposed solution can work in tandem with other protocols. This is because we do not modify the coherency protocol by any means and all proposed elements of the design are independent of the details of the implemented protocol. This also simplifies the verification efforts compared to the approaches that alter protocols.

Instead of a unified interconnect commonly deployed in real-time architectures, we consider a split-transaction bus in which all communications between cores and LLC is done using two separate buses: 1) *request bus*, which is responsible for broadcasting the coherency messages; 2) *response bus*, which is a dedicated interconnect to transfer the data responses from/to cores. These two buses operate in parallel to improve the performance of the system. The *request bus* and *response bus* take a certain amount of time to transfer message packet and data response, which we represent with t_{REQ} and t_{RESP} , respectively. In order to maximize the parallelism in the system, we propose to bankize the LLC such that multiple requests can be processed simultaneously. Bankizing LLC is a common approach in COTS platforms to increase system's performance such as in Intel's architecture [2]. Therefore, we assume that the LLC consists of N independent banks, $b_1, \dots, b_i, \dots, b_N$ where each bank consumes a certain amount of time t_{BANK} to process writing data to (or retrieving data from) the cache data array inside each bank. In addition, LLC banks could be shared among all cores [12] or partially shared similar to [22]. In our model, we assume the former. Similar to existing related works [40, 17, 15, 10, 8], in this paper, we only focus on the interference suffered by the L1-LLC traffic due to coherence, shared cache(s), and shared interconnects and do not model the extra interference occurring in off-chip memory due to LLC misses. This latter can be bounded using other existing orthogonal approaches such as [39, 12, 13, 7].

We assume that each cache entity (private and LLC) has its own set of interconnect buffers: **TxMsg**, **RxMsg**, **TxResp**, and **RxResp** to register the incoming/outgoing messages and data responses. **RxMsg** contains the incoming message packets from the request bus. We assume that every message will be decoded immediately in the private cache and each LLC bank even if the bank is busy writing/retrieving data from its data array. **TxMsg** contains the outgoing message packets from any core/bank that must snoop on the request bus. For instance, if a core asks to modify a cache line that it is not in possession of, the core must inform other cores by a coherency message (GetM) and push it in its own **TxMsg** buffer to be propagated on the request bus. This allows other cores/LLC to be aware of this action. **RxResp** contains the data responses coming from the response bus. **RxResp** at each core includes data response that the bank provides or data response due to a cache-to-cache transfer. **RxResp** at LLC bank includes the data response supplied by the cores in case of write-back. Notice that unlike **RxResp** buffers of each core, the data responses placed in LLC **RxResp** must be processed in the bank which takes t_{BANK} to process. Finally, **TxResp** includes the responses that need to be transferred on the response bus. The request bus, response bus, and LLC banks act as independent shared resources which conduct their own independent arbitration policies. In detail, the request bus arbiter is responsible to arbitrate the messages residing in **TxMsg** and the data responses inside **TxResp** buffers are arbitrated through the response bus arbiter. Similarly, each arbiter at LLC bank arbitrates the message/responses in **RxMsg** and **RxResp** buffers.

3.2 Request Processing and Order of Arbitration

From the perspective of the coherency architecture, a requestor issues *requests* to the system based on the following activities in L1 cache: 1) load miss requests; 2) store miss requests, including stores to a cache line in **S** state; 3) replacement requests due to a write-back to shared memory or caused by an eviction. As mentioned earlier in this section, the proposed



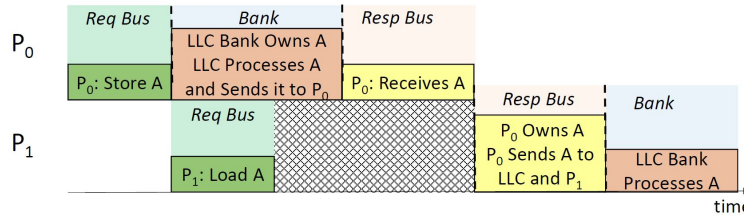
■ **Figure 2** The sequence of arbitration based on the request type.

architecture applies the arbitration schemes at all different resources including request bus, response bus, and each bank in LLC. Requests can experience different sequences of services on the arbitration resources. In detail, we consider three different *types* \mathcal{T} of requests, depending on the sequence of arbitration: 1) **REQ:RESP:BANK** meaning that it first needs to broadcast on the request bus, then the data response will be propagated on the response bus and finally the data response should be processed at LLC bank; 2) **REQ:BANK:RESP** representing requests that need to first broadcast on the request bus, then the shared bank must process and fetch the data response and finally this data response must be propagated over the response bus; 3) **REQ:RESP**: the last category is related to cache-to-cache transfers. In such a scenario, after broadcasting the message on the request bus, the response will be supplied by the owner core on the response bus.

Figure 2 depicts all possible cases in which a request can be processed based on its type and coherency status. Figure 2a represents a scenario where core P_1 aims to load cache line A ; therefore, it first needs to broadcast its action by sending the required coherency message on the request bus. Here, the owner of A is P_0 and the cache line A is in **M** coherency state; hence, according to **MSI** coherency protocol, P_0 must send A to both core P_1 and the LLC bank by pushing the response data into **RxResp** buffer of P_1 and the bank. Then, P_1 receives its data response; however, the data still needs to be processed inside the bank which requires t_{BANK} time. Note that all of these actions are eligible to execute after their corresponding arbitration issue them the grant to access the resource. Hence, the sequence of arbitration for this request follows **REQ:RESP:BANK**.

In Figure 2b a load/store request from P_1 targets cache line A which is owned by the shared bank. Therefore, the bank is responsible to process the request, and then it can be returned to the core through the response bus. Hence, the sequence of arbitration for this request follows **REQ:BANK:RESP**. For the replacement request shown in Figure 2c, after broadcasting the message, the core needs to transfer the data to the LLC bank by sending it to the **RxResp** buffer of bank. Hence, the sequence of arbitration for this request follows **REQ:RESP:BANK**. Finally, Figure 2d shows a scenario where P_1 tries to store to cache line A while the line is owned by P_0 . According to the **MSI** coherency protocol, the LLC bank is not required to acknowledge this action; therefore, sending the response from P_0 to P_1 suffices the store request and the sequence of arbitration for this request follows **REQ:RESP**.

Finally, to maintain the correctness of execution, the service order for requests to the same cache line must respect the order in which the requests are issued on the request bus. Once a request starts being issued on the request bus, we say that it *depends* on previous requests to the same cache line which have already been issued on the request bus but not completed yet. Since requests are issued one at a time on the request bus, this means that requests



■ **Figure 3** The lower priority request from P_1 depends on the higher priority request of P_0 to the same cache line A .

to the same cache line form a *chain of dependencies*, where requests are ordered based on when they are issued on the request bus. Due to dependencies, requests must complete in the same order in which they are issued on the request bus. In addition, some requests might not move to the next resource even though their process is finished at the current resource. Specifically, we say that a request is *ready* on a resource if it can be considered for arbitration at that resource. A request becomes ready on REQ when it arrives. For a RESP or BANK resource, a request becomes ready when it finishes processing at its previous resource, or when the previous request in the chain of dependencies finishes on that resource (if such previous request exists and uses the resource), whichever happens later. Figure 3 shows an example with two requests of different types targeting the same cache line A : the request of P_0 follows the scenario in Figure 2b while P_1 follows Figure 2a. Since the request of P_0 is issued on the request bus first, the request of P_1 depends on it and, to maintain consistency, it cannot start service on the response bus until P_0 finishes receiving the data from the response bus.

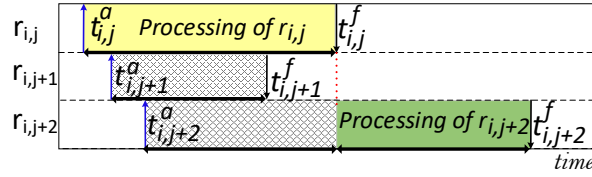
3.3 Latency Model

Now, we are able to precisely define the request latency from the core perspective. We assume that the requests of each core P_i can be ordered based on their arrival time. Hence, we can index them as $r_{i,1}, \dots, r_{i,j}, \dots$. Each request has a type \mathcal{T} according to its sequence. Since OOO cores might issue multiple requests simultaneously and the LLC contains many independent banks, they can serve multiple requests simultaneously. It is thus important to formally define the finish time and processing time of a request.

► **Definition 1** (Arrival and Finish Time). Let $t_{i,j}^a$ be the arrival time of request $r_{i,j}$, that is, the time at which $r_{i,j}$ is queued in TxMsg . The finish time $t_{i,j}^f$ of $r_{i,j}$ is the time at which the last action in its sequence is completed. This includes writing to the shared bank if the type of sequence is REQ:RESP: BANK or receiving the response from the response bus if the type of sequence is REQ: BANK: RESP and REQ: RESP . We say a request is outstanding if it has already arrived but is not finished yet. We say a request is pending if it is outstanding and it has already started or completed being issued on the request bus.

Note that while the concept of a pending request is not used in this section, it will be relevant when defining the arbiter behavior in Section 4 because, as previously explained, requests to the same cache line become dependent on each other once they become pending.

► **Definition 2** (Processing Latency). For any request $r_{i,j}$, let prec_j be the index of the request r_{i,prec_j} of P_i with latest finish time among those that arrived before $r_{i,j}$ (i.e., such that $\text{prec}_j < j$). Then the processing latency of $r_{i,j}$ is $\max(0, t_{i,j}^f - \max(t_{i,\text{prec}_j}^f, t_{i,j}^a))$.



■ **Figure 4** Processing latency example. Assume that all previous requests $r_{i,l}$ with $l < j$ finish before $t_{i,j}^a$. We use \uparrow for the arrival time $t_{i,j}^a$ of each request and \downarrow for its finish time $t_{i,j}^f$.

► **Definition 3** (Oldest Request). *At any time t , the oldest request of a requestor (if any) is the earliest arrived request of that requestor that is still outstanding at t .*

Figure 4 delineates a clarifying example, borrowed from [25], where three requests arrive from the same core such that $prec_{j+1} = prec_{j+2} = j$. $r_{i,j}$ becomes the oldest request as soon as it arrives at $t_{i,j}^a$. Initially, both $r_{i,j+1}$ and $r_{i,j+2}$ are non-oldest requests. Similar to related work, we are interested in bounding the processing latency of requests. This is due to the fact that when a requestor issues multiple requests and stalls until they finish, the stall time is upper bounded by the sum of the processing latencies of the requests. In detail, as discussed in [20, 39], the latency suffered by a real-time task running on a core accessing a shared resource can be bounded by the sum of the processing latency of the requests issued by the task. For this reason, the processing latency of $r_{i,j+1}$, which is covered by the processing time of $r_{i,j}$, is set to zero. Notice that a non-oldest request might or might not become the oldest request of its requestor. As shown in the example, $r_{i,j+2}$ becomes oldest once $r_{i,j}$ finishes, while $r_{i,j+1}$ never became oldest and its processing time is zero. For this reason, we only need to consider the processing latency of oldest requests. Finally, if a request becomes oldest, by definition it does so at time $\max(t_{i,prec_j}^f, t_{i,j}^a)$ when its processing latency starts.

3.4 Task Analysis

In Section 5, we will derive a bound on the processing latency for each of the three types of request defined in Section 3.2. The total access latency for a task can then be determined by summing the product of the number of requests of each type issued by the task by the WCL for that type [14].

We assume that a portion of accesses by the task targets data shared with other cores, while some accesses are to non-shared data. For each case, we need to retrieve the number of load miss requests, store miss requests, and the number of replacements from the task. For non-shared data, approaches based on either profiling or static analysis can be used to extract the number of requests. For shared data, to the best of our knowledge, no general method exists to determine which cache lines exist in the cache of the other cores at any point of time. A safe assumption can be adopted where every load request on shared data is considered a load miss, and every store request on shared data is considered a store miss [15]. However, if better assumptions can be made based on code analysis, our framework can take advantage of them by deriving different latency bounds for each type of request.

Note that based on Figure 2, for shared data, load misses can be of type `REQ:RESP:BANK` or `REQ:BANK:RESP`, as shown in Figures 2a and 2b, while store misses can be either `REQ:BANK:RESP` or `REQ:RESP`. For non-shared data, load and store misses can only be of type `REQ:BANK:RESP`. Replacements can only follow `REQ:RESP:BANK` as shown in Figure 2c. If we cannot determine the specific type of a request based on task analysis, we simply consider the largest latency among the types to which the request might belong.

4 Proposed Arbiter

This section describes the behavioral details of the proposed arbiter. The proposed arbiter considers the realistic hardware architecture introduced in Section 3 and maintains predictability by design while maximizing average-case performance. Based on the hardware architecture, there exist three distinct types of resources in the system. Formally, we capture the behavior of the proposed arbiter by a set of rules. In order to predictably manage interference among different cores, the arbiter maintains a unified Global Round-Robin (GRR) order of requestors across all resources. A requestor is removed from the GRR queue after the oldest request of that requestor completes at its last resource, and it is inserted at the back of the queue either immediately when it has any other request or when its next request arrives. At any point in time, the *Global Request Queue* shown in Figure 1 contains all outstanding requests in the system as well as their state in terms of their next resource that they need to get processed on. In addition, a work-conserving approach is used at each resource to increase overall system performance. Specifically, the proposed arbiter deploys a two-level arbitration mechanism: 1) oldest requests over non-oldest per core; 2) GRR order among the oldest requests; if no oldest request is ready, GRR over non-oldest requests.

Rule 1. (Global Round-Robin Ordering) The arbiter maintains a Global Round-Robin order of requestors across all resources. Each request is associated with a *GRR priority* as follows: given two outstanding requests $r_{p,q}, r_{i,j}$, $r_{p,q}$ has higher GRR priority than $r_{i,j}$ if: (1) $r_{p,q}$ is oldest and $r_{i,j}$ is not oldest; or (2) $r_{p,q}$ and $r_{i,j}$ are both oldest or both non-oldest, and P_p is ahead of P_i in the GRR order of requestors.

Note that GRR priorities for oldest requests are static, in the sense that they never change while an oldest request is outstanding: this is because the relative requestor order in the GRR queue is fixed once the request becomes oldest. However, GRR priorities for non-oldest requests are not static: specifically, a non-oldest request $r_{p,q}$ might have higher GRR priority than a non-oldest request $r_{i,j}$ at time t , but its GRR priority might become lower than $r_{i,j}$ at some later time t' once an oldest request of P_p completes, forcing P_p to be enqueued at the back of the GRR queue (assuming that $r_{p,q}$ does not become oldest at t').

The arbiter manages each resource independently, selecting the highest priority request that is ready on each resource. Since requests that are ready on the request bus do not depend on other requests, the arbiter manages the request bus according to strict GRR priorities. However, requests that are ready on a bank or the response bus might depend on each other. To correctly arbitrate in the presence of dependant requests, we further introduce a priority inheritance mechanism, where a lower-priority pending request inherits the priority of a higher-priority request that depends on it. Note a further complexity: a lower-priority pending request might target the same cache line as a higher-priority request, but the higher-priority request does not depend on it if the higher-priority request has not yet started being issued on the request bus. However, the higher-priority request will eventually become pending and thus depend on the lower-priority one. To capture such behavior, we extend the concept of dependency to the one of *eventual dependency* to include both requests that are currently dependent, but also requests that will be dependent in the future once they become pending. Based on this concept, we can define dynamic priorities that are used to arbitrate on each bank and the response bus.

Rule 2. (Priority Inheritance) The dynamic priority of a request is equal to the highest priority between its own GRR priority and (if the request is pending) the GRR priority of any other request that eventually depends on it.

Rule 3. (Resource Arbitration) The arbiter manages all three resources, including request bus, response bus, and each LLC banks, independently. On the request bus, the arbiter selects the ready request with the highest GRR priority. On the response bus and each bank, the arbiter selects the request that is ready on the corresponding resource and has the highest dynamic priority.

As before, eventually dependent requests form an *eventual chain of dependencies*, based on the order in which they either already became or will become pending. Note that since the request bus follows strict GRR priorities, an oldest request that is not pending yet will be preceded in the eventual chain of dependency by all already pending requests to the same cache line, as well as by all other oldest requests to the same cache line that are not pending yet and have higher GRR priority.

The proposed arbiter supports out-of-order execution, allowing processing cores to issue multiple requests simultaneously. Based on Rule 3, it is clear that if the arbiter allows many non-oldest requests to the same cache line to be sent, then an oldest request could arrive and suffer priority inversion on all those non-oldest requests. Therefore, to limit the amount of priority inversion in the system, we set a parameter $k_{ceil} \geq 0$, that controls the possibility of sending non-oldest requests ahead of a possible oldest request to the same cache line.

Rule 4. (Request Blocking) When applying Rule 3 on the request bus, the arbiter does not consider a non-oldest request $r_{i,j}$ if there are already other k_{ceil} pending non-oldest requests to the same cache line.

5 Latency Analysis

In this section, we detail the latency analysis for the proposed arbiter. Specifically, consider an oldest request under analysis r_{ua} of type \mathcal{T} targeting a bank b_k , and let $t_{ua}^a, t_{ua}^f, t_{prec_{ua}}^f$ be its arrival, finish time, and the finish time of the request with latest finish time among those that arrived before r_{ua} and belong to the same core. We first show how to compute an upper bound to the remaining latency (time to finish) $t_{ua}^f - t_{now}$ of r_{ua} at time t_{now} , based on the current state of the resource - following related work [25], we call this the *dynamic bound*. Then, we obtain the *static worst-case bound* $\Delta(\mathcal{T}, k_{ceil})$, i.e. an upper bound to the processing latency of any request of type \mathcal{T} for a given value of k_{ceil} , by maximizing the dynamic bound over all possible states of the system at time $t_{now} = \max(t_{prec_{ua}}^f, t_{ua}^a)$ when r_{ua} becomes oldest.

5.1 Dynamic Latency Analysis

Depending on its type \mathcal{T} and its current state at time t_{now} , r_{ua} will need to be serviced on one or more resources; in analogy to processor scheduling, we say that r_{ua} must *execute* on those resources. As in Section 3.2, we use **REQ** to denote the request bus, **RESP** for the response bus, and **BANK** for bank b_k .

We start by proving a fundamental property of the proposed arbitration scheme; namely, the fact that, despite the priority inheritance mechanism in arbitration Rule 2, a lower-priority request $r_{i,j}$ cannot increase its dynamic priority above r_{ua} after time t_{now} .

► **Lemma 4.** *Consider any request $r_{i,j}$ other than r_{ua} . If $r_{i,j}$ has lower dynamic priority than r_{ua} at t_{now} , or has not arrived in the system yet, then its dynamic priority cannot become higher than r_{ua} at any time $t > t_{now}$ while both requests are outstanding.*

Proof. We first show that the dynamic priority of r_{ua} cannot decrease after t_{now} . By arbitration Rule 2, the dynamic priority of r_{ua} at t_{now} is equal to either its GRR priority, or the GRR priority of a higher-priority request $r_{p,q}$ that eventually depends on r_{ua} . Since r_{ua} is oldest, such GRR priorities are static and cannot decrease. Furthermore, as noted in Section 3.2, the coherency protocol forces requests to finish in dependency order. Hence, if r_{ua} inherits the GRR priority of $r_{p,q}$ at t_{now} , then $r_{p,q}$ cannot finish before r_{ua} , and thus the dynamic priority of r_{ua} cannot decrease below the GRR priority of $r_{p,q}$.

Next, we show that the GRR priority of $r_{i,j}$ cannot become higher than the dynamic priority of r_{ua} : (1a) if $r_{i,j}$ is already oldest at time t_{now} , then its GRR priority is static and thus cannot increase; (1b) otherwise, the GRR priority of $r_{i,j}$ increases when it becomes oldest after t_{now} , but it must still be lower than the one of r_{ua} since GRR priorities for oldest requests are based on when they become oldest (which is when their core is pushed to the back of the GRR queue).

In summary, we have shown that the dynamic priority of r_{ua} cannot decrease after t_{now} , and the GRR priority of $r_{i,j}$ cannot increase past it. Therefore, $r_{i,j}$ can only acquire a higher dynamic priority than r_{ua} based on Rule 2 if it inherits the GRR priority of an oldest request $r_{p,q}$ such that: (A) $r_{p,q}$ is either r_{ua} or has a higher GRR priority than r_{ua} ; (B) at time t_{now} , $r_{i,j}$ has either not arrived yet, or does not inherit the priority of $r_{p,q}$; (C) at some time t after t_{now} , $r_{i,j}$ is inheriting the priority of $r_{p,q}$. We now show that this is impossible.

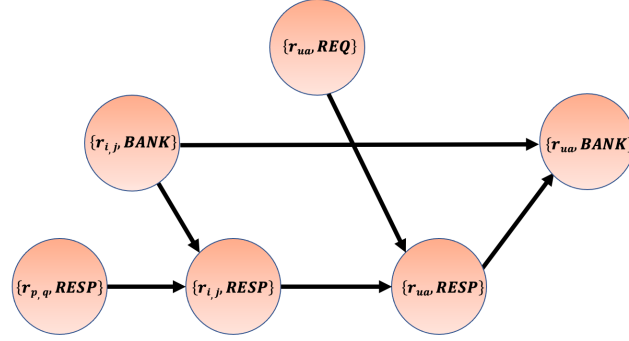
Since by (A) the static GRR priority of $r_{p,q}$ must be equal to or higher than r_{ua} , it follows that $r_{p,q}$ must already be outstanding and oldest at t_{now} . We next consider two possible cases: (2a) $r_{p,q}$ is pending at t_{now} ; (2b) not pending. In case (2a), note that the set of requests that $r_{p,q}$ depends upon are fixed when $r_{p,q}$ becomes pending; hence (B) and (C) cannot simultaneously hold. In case (2b), for (B) and (C) to be satisfied, $r_{i,j}$ must target the same cache line as $r_{p,q}$ and become pending after t_{now} and before $r_{p,q}$. However, this is again impossible: since the GRR priority of $r_{i,j}$ is lower than r_{ua} and thus $r_{p,q}$, it follows that by Rule 3, $r_{i,j}$ cannot be serviced on the request bus and become pending before $r_{p,q}$. ◀

Based on Lemma 4, we can evaluate at time t_{now} which requests have higher priority than r_{ua} and can thus interfere with it. In details, let \mathcal{R}_{REQ} ($\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$) be the set of outstanding requests with GRR (respectively, dynamic) priority higher than or equal to r_{ua} at time t_{now} , including r_{ua} itself, and which have not yet started executing on REQ (respectively, BANK or RESP)². Since GRR priorities for oldest requests are static, no request that is not included in \mathcal{R}_{REQ} can become higher GRR priority than r_{ua} , and thus interfere with it on REQ, at any time $t > t_{\text{now}}$; and by Lemma 4, the same holds for $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$ on BANK and RESP.

Next, we formalize the dependencies among requests that target the same cache line as r_{ua} . We do so by constructing a DAG, where each node represents the execution of one request in the eventual chain of dependencies on a resource.

► **Definition 5** (Dependency DAG). *The dependency DAG for r_{ua} at time t_{now} is a directed acyclic graph $G = (V, E)$ where: (1) V is a set of nodes; each node is of the form $\{r_{i,j}, res\}$, where $r_{i,j}$ is either r_{ua} or one of the requests that targets the same cache line as r_{ua} and precedes it in the eventual chain of dependencies, while res is one of the resources on which $r_{i,j}$ has not yet finished executing at t_{now} ; (2) E is a set of edges of two types: (2a) for*

² Note that the sets comprise only requests that have yet to start executing on a resource because in Lemma 7 we will account for the interference of requests that have already started executing on each resource in a different manner



■ **Figure 5** Example dependency DAG. The eventual chain of dependencies comprises requests $(r_{p,q}, r_{i,j}, r_{ua})$.

each request $r_{i,j}$, E includes an edge $\{r_{i,j}, res'\} \rightarrow \{r_{i,j}, res\}$ if the two nodes exist in the graph and $r_{i,j}$ executes on res' before res ; (2b) for each pair of consecutive requests $r_{p,q}, r_{i,j}$ in the eventual chain of dependencies, E includes edges $\{r_{p,q}, RESP\} \rightarrow \{r_{i,j}, RESP\}$ and $\{r_{p,q}, BANK\} \rightarrow \{r_{i,j}, BANK\}$ if the corresponding nodes exist in the graph.

Example: assume that at time t_{now} , there are three outstanding requests targeting the same cache line: $r_{p,q}$ of type REQ:BANK:RESP became pending first, and has already executed on BANK but not yet finished on RESP; $r_{i,j}$, also of type REQ:BANK:RESP, became pending after $r_{p,q}$ and has completed executing on REQ but not yet finished on BANK; and r_{ua} is of type $\mathcal{T} = \text{REQ:RESP:BANK}$ and not yet pending. Then, the eventual chain of dependencies is $(r_{p,q}, r_{i,j}, r_{ua})$, and the dependency DAG is depicted in Figure 5.

Note that by definition and based on Section 3.2, the dependency DAG contains an edge between two nodes whenever the second node cannot become ready before the first one finishes executing. Therefore, if a node $\{r_{i,j}, res\}$ in the dependency DAG has no predecessors, then it must be ready on res at t_{now} . Otherwise, it becomes ready once all predecessor nodes finish executing. We can then define the latency for a node as follows. The latency window for a node $\{r_{i,j}, res\}$ with one or two immediate predecessors spans from the time it becomes ready on res , until the time it finishes executing on res , and its latency is the difference between the two times. If $\{r_{i,j}, res\}$ has no predecessor, then its latency window spans from t_{now} until the time it finishes executing given that we do not want to account for latency before t_{now} . The latency for a (directed) path through G is simply the sum of the latencies of the constituent nodes. Finally, note that as pointed out in Section 4, requests in the eventual chain of dependencies for r_{ua} , and thus the dependency graph, are either already pending or are oldest. Therefore, arbitration Rule 4 cannot block any such request, and thus we do not need to consider it when determining the latency of a path.

We can now derive the remaining latency for r_{ua} . In details, in Lemma 6 we first show that its remaining latency must be equal to the latency of a critical path in the dependency DAG, that is, a path that starts from a node with no predecessors and finishes with the last node of r_{ua} . Then, in Lemma 7, we show that the latency of a critical path is upper bounded by Equation 1. Therefore, the remaining latency of r_{ua} can be computed by taking the maximum of Equation 1 over every potential critical path in the DAG.

► **Lemma 6.** *The remaining latency $t_{ua}^f - t_{\text{now}}$ for r_{ua} at time t_{now} is equal to the latency of some path \mathcal{P} in its dependency DAG whose last node is $\{r_{ua}, res\}$ and res is the last resource on which r_{ua} executes. In such critical path, each node becomes ready when the previous one finishes executing, except for the first node which has no predecessor and is ready at t_{now} .*

Proof. We iteratively construct the critical path \mathcal{P} as follows. We first start from the path that contains only node $\{r_{ua}, res\}$. Note that by definition, such node finishes executing at t_{ua}^f . We then have two possible cases: (1) the node has no predecessor; (2) it has one or two immediate predecessors. In case (1), r_{ua} is ready on res at t_{now} . Therefore, the latency of the node, which is equal to the latency of \mathcal{P} , is $t_{ua}^f - t_{now}$ and the lemma follows. In case (2), r_{ua} becomes ready on res , and thus the latency window for the node starts, when one of its immediate predecessor nodes finishes executing. Let $\{r_{i,j}, res'\}$ be such node. We can then add it to the beginning of \mathcal{P} and repeat the same reasoning: if the node has no predecessor, then $r_{i,j}$ must be ready on res' at t_{now} and thus its latency window spans from t_{now} to the beginning of the latency window for $\{r_{ua}, res\}$. Therefore, again the latency of \mathcal{P} is equal to $t_{ua}^f - t_{now}$. If instead $\{r_{i,j}, res'\}$ has one or two immediate predecessors, we continue the iteration by adding one such node to the path. But since the graph is by definition a DAG (hence has no cycles) and the number of nodes is finite, the iteration must eventually terminate. The lemma follows. \blacktriangleleft

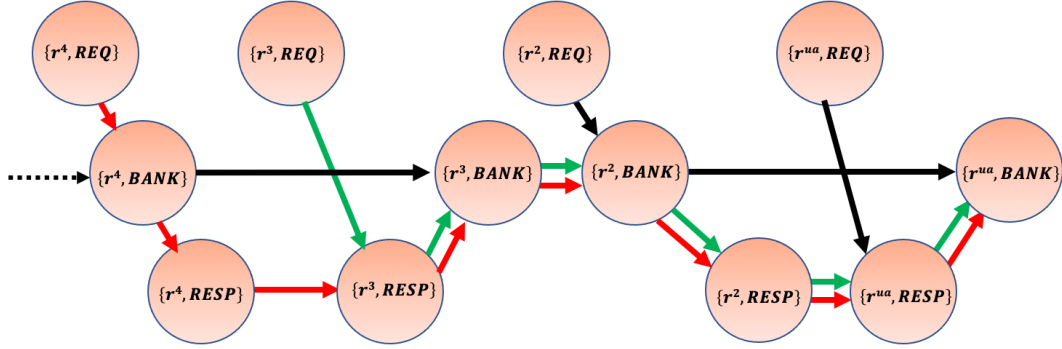
► **Lemma 7.** *The latency of a critical path \mathcal{P} is upper bounded by:*

$$c_{\overline{res}} + \sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} + K_{BANK}(\mathcal{P}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{P}) \cdot (t_{RESP} - 1), \quad (1)$$

where \overline{res} is the resource on which the first node in \mathcal{P} executes, $c_{\overline{res}}$ is the remaining time to finish executing the current executing request on \overline{res} at t_{now} (if any, otherwise 0), $\mathcal{S}_{\mathcal{P}}$ is the set of all resources on which nodes in \mathcal{P} execute (with the exception of the first node if it already started executing on \overline{res} at t_{now}), and $K_{BANK}(\mathcal{P})$ ($K_{RESP}(\mathcal{P})$) is the number of nodes in \mathcal{P} that executes on *BANK* (respectively, *RESP*) and are preceded in \mathcal{P} by a node that does not execute on *BANK* (respectively, *RESP*).

Proof. We first consider the case in which the first node in \mathcal{P} has not yet started executing on \overline{res} . For each $res \in \mathcal{S}_{\mathcal{P}}$, we consider the total latency of nodes in \mathcal{P} that execute on res . Since latencies are computed during time windows when a request in \mathcal{P} is ready on res , it follows that the only executions that can contribute to the total latency are: (1) for each node $\{r_{i,j}, res\}$, at most one request that is already executing on res at the beginning of its window. For the first node in \mathcal{P} , by definition the length of such execution is $c_{\overline{res}}$. For every other node, since the request is already executing, it can be bound as $t_{res} - 1$. (2) Requests in \mathcal{P} that have not yet started executing on res (since otherwise they would be included in the previous category). (3) Other requests that have not yet started executing on res and have higher GRR priority (if res is *REQ*) or higher dynamic priority (if res is *BANK* or *RESP*) than some request in \mathcal{P} that has not yet started executing on res . Note that lower priority requests can only be included in category (1). Also, as noted in Section 4, requests that precede r_{ua} in the extended chain of dependencies, and can thus be included in \mathcal{P} , must either be pending (and hence have higher or equal dynamic priority than r_{ua}) or be non-pending and have higher GRR priority than r_{ua} . Hence, requests in category (2) are included in \mathcal{R}_{res} ; and since no request that is not included in \mathcal{R}_{res} can acquire higher priority than r_{ua} after t_{now} , requests in category (3) must also be included in \mathcal{R}_{res} . Therefore, $|\mathcal{R}_{res}| \cdot t_{res}$ upper bounds the contributions on requests in categories (2) and (3).

It remains to determine the number of requests in (1). Note that if a node executing on res is preceded in \mathcal{P} by a node executing on the same resource (either *RESP* or *BANK*), then no request can be executing at the beginning of its window, since it corresponds to the time at which the preceding node finishes executing on res . Hence, the number of requests can be bounded by the number of nodes that are preceded by another node executing on a



■ **Figure 6** Example paths with maximal number of $\text{RESP} \rightarrow \text{BANK}$ and $\text{BANK} \rightarrow \text{RESP}$ edges for a r_{ua} of type $\mathcal{T} = \text{REQ}:\text{RESP}:\text{BANK}$. The eventual chain of dependencies comprises requests $(\dots, r^4, r^3, r^2, r_{ua})$.

different resource, plus possibly the first node in the path. By definition, this adds a latency of $c_{\overline{res}} + K_{\text{BANK}}(\mathcal{P}) \cdot (t_{\text{BANK}} - 1) + K_{\text{RESP}}(\mathcal{P}) \cdot (t_{\text{RESP}} - 1)$. Adding the contribution of categories (2) and (3) to the one of (1) yields Equation 1.

Finally, we consider the case in which the first node in \mathcal{P} is already executing on \overline{res} at time t_{now} . In this case, the latency of the first node is simply $c_{\overline{res}}$, and no other request can execute in its latency window. The same reasoning as above can then be applied to the other nodes in \mathcal{P} , given that $\mathcal{S}_{\mathcal{P}}$ does not account for the first node. This again results in a latency of $|\mathcal{R}_{res}| \cdot t_{res}$ for (2) and (3) and a latency of $K_{\text{BANK}}(\mathcal{P}) \cdot (t_{\text{BANK}} - 1) + K_{\text{RESP}}(\mathcal{P}) \cdot (t_{\text{RESP}} - 1)$ for (1), which added to $c_{\overline{res}}$ for the first node yields Equation 1. ◀

Example. consider the example in Figure 5. The three possible critical paths are $\mathcal{P}' = \{r_{p,q}, \text{RESP}\} \rightarrow \{r_{i,j}, \text{RESP}\} \rightarrow \{r_{ua}, \text{RESP}\} \rightarrow \{r_{ua}, \text{BANK}\}$, $\mathcal{P}'' = \{r_{i,j}, \text{BANK}\} \rightarrow \{r_{i,j}, \text{RESP}\} \rightarrow \{r_{ua}, \text{RESP}\} \rightarrow \{r_{ua}, \text{BANK}\}$, and $\mathcal{P}''' = \{r_{ua}, \text{REQ}\} \rightarrow \{r_{ua}, \text{RESP}\} \rightarrow \{r_{i,j}, \text{BANK}\}$. Note that no critical path can include edge $\{r_{i,j}, \text{BANK}\} \rightarrow \{r_{ua}, \text{BANK}\}$, as the DAG includes the longer path \mathcal{P}'' between $\{r_{i,j}, \text{BANK}\}$ and $\{r_{ua}, \text{BANK}\}$, and thus $\{r_{ua}, \text{BANK}\}$ must become ready once $\{r_{ua}, \text{RESP}\}$ finishes. Further note that for \mathcal{P}'' we have $\overline{res} = \text{BANK}$, $\mathcal{S}_{\mathcal{P}} = \{\text{BANK}, \text{RESP}\}$, $K_{\text{BANK}}(\mathcal{P}'') = 1$, $K_{\text{RESP}}(\mathcal{P}'') = 1$. Its latency can be upper bounded by summing: the remaining time c_{BANK} to finish executing the current executing request on BANK (if any); plus the maximum time $t_{\text{RESP}} - 1$ to finish executing a request once $r_{i,j}$ becomes ready on RESP; plus the maximum time $t_{\text{BANK}} - 1$ to finish executing a request once r_{ua} becomes ready on BANK; plus the time $|\mathcal{R}_{\text{BANK}}| \cdot t_{\text{BANK}}$ to execute requests with higher or equal priority (including the ones in the DAG) that have not yet started executing on BANK; plus the time $|\mathcal{R}_{\text{RESP}}| \cdot t_{\text{RESP}}$ to execute requests with higher or equal priority that have not yet started executing on RESP.

Lemmas 6 and 7 provide a way to estimate the remaining latency for r_{ua} . However, they require constructing the dependency DAG and all possible critical paths. As it will become clear in Section 6, to apply the Duetto reference model we need to estimate the remaining latency online in hardware at every clock cycle. Therefore, we next derive a simpler, albeit conservative, way to determine the remaining latency. Specifically, we replace $K_{\text{BANK}}(\mathcal{P})$ and $K_{\text{RESP}}(\mathcal{P})$ with upper bounds $\overline{K}_{\text{BANK}}(C)$ and $\overline{K}_{\text{RESP}}(C)$ which depend only on the number C of requests in the dependency DAG; this ensures that at run-time, we only need to maintain the list of requests in the eventual chain of dependencies and which resources they need to execute upon, but not the detailed DAG structure.

► **Lemma 8.** For $C \geq 1$, define:

$$\overline{K_{BANK}}(\mathcal{T}, C) = \begin{cases} \lfloor (C+1)/2 \rfloor & \text{if } \mathcal{T} = \text{REQ:}BANK:RESP \\ \lceil (C+1)/2 \rceil & \text{if } \mathcal{T} = \text{REQ:}RESP: BANK \\ \lceil (C-1)/2 \rceil & \text{if } \mathcal{T} = \text{REQ:}RESP \end{cases} \quad (2)$$

$$\overline{K_{RESP}}(\mathcal{T}, C) = \begin{cases} \lceil (C+1)/2 \rceil & \text{if } \mathcal{T} = \text{REQ:}BANK:RESP \\ \lfloor (C+1)/2 \rfloor & \text{if } \mathcal{T} = \text{REQ:}RESP: BANK \\ \lfloor (C+1)/2 \rfloor & \text{if } \mathcal{T} = \text{REQ:}RESP \end{cases} \quad (3)$$

For a r_{ua} of type \mathcal{T} and any critical path \mathcal{P} comprising nodes of C requests, it holds: (1) $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(C) - 1$ if the first node in \mathcal{P} executes on $BANK$, and $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(C)$ otherwise; (2) $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(C) - 1$ if the first node in \mathcal{P} executes on $RESP$, and $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(C)$ otherwise.

Proof. First note that by construction, only the first node in \mathcal{P} can execute on REQ ; all other nodes must execute on either $BANK$ or $RESP$. Therefore, $K_{BANK}(\mathcal{P})$ and $K_{RESP}(\mathcal{P})$ are maximized when \mathcal{P} comprises the maximum number of edges from a node executing on $RESP$ to a node on $BANK$ (an edge $RESP \rightarrow BANK$), and from a node on $BANK$ to a node on $RESP$ (an edge $BANK \rightarrow RESP$). Note that if two consecutive requests in the eventual chain of dependencies are of the same type (for example, $REQ:RESP: BANK$), then any path over $BANK$ and $RESP$ nodes belonging to those two requests can have only one such edge (in this example, the two possible paths are $RESP \rightarrow BANK \rightarrow BANK$ and $RESP \rightarrow RESP \rightarrow BANK$); on the other hand, if two consecutive requests are one of type $REQ:RESP: BANK$ and the other of type $REQ: BANK: RESP$, then the path can have one edge $RESP \rightarrow BANK$ and one $BANK \rightarrow RESP$. Hence, $K_{BANK}(\mathcal{P}), K_{RESP}(\mathcal{P})$ are maximized when requests in the eventual chain of dependencies switch between the two types.

Next consider $\mathcal{T} = \text{REQ:}RESP: BANK$. Figure 6 shows the resulting DAG when requests in the chain switch types, together with two possible critical paths with $C = 4$ (even) and $C = 3$ (odd) requests, where the first node executes on REQ . By construction, the path comprises an edge $RESP \rightarrow BANK$ for r_{ua} , and for every second request before it; hence, the number of such edges is $\lceil C/2 \rceil$. In addition, if C is even, there is a $REQ \rightarrow BANK$ edge for the first request. Hence, the maximum value of $K_{BANK}(\mathcal{P})$ can be computed as $\overline{K_{BANK}}(\text{REQ:}RESP: BANK, C) = \lceil (C+1)/2 \rceil$; except that if \mathcal{P} starts with a node on $BANK$, then that node is not preceded by any other node, hence the maximum value of $K_{BANK}(\mathcal{P})$ is $\overline{K_{BANK}}(\text{REQ:}RESP: BANK, C) - 1$. Similarly for $K_{RESP}(\mathcal{P})$, we note that the path comprises an edge $BANK \rightarrow RESP$ for the request before r_{ua} , and for every second request before it; plus an edge $REQ \rightarrow RESP$ if C is odd. Hence, the maximum value of $K_{RESP}(\mathcal{P})$ can be computed as $\overline{K_{RESP}}(\text{REQ:}RESP: BANK, C) = \lfloor (C+1)/2 \rfloor$, or $\overline{K_{RESP}}(\text{REQ:}RESP: BANK, C) - 1$ if the first node in \mathcal{P} is on $RESP$. This concludes the proof for $\mathcal{T} = \text{REQ:}RESP: BANK$.

For brevity, we omit the proof for $\mathcal{T} = \text{REQ:}RESP$ and $\mathcal{T} = \text{REQ:}BANK:RESP$, since the derivation is equivalent; in particular, note that $\overline{K_{RESP}}(\text{REQ:}RESP, C) = \overline{K_{RESP}}(\text{REQ:}RESP: BANK, C)$ but $\overline{K_{BANK}}(\text{REQ:}RESP, C) = \overline{K_{BANK}}(\text{REQ:}RESP: BANK, C) - 1$, since with $\mathcal{T} = \text{REQ:}RESP$ we miss the $RESP \rightarrow BANK$ edge for r_{ua} ; while for $\mathcal{T} = \text{REQ:}BANK:RESP$ we have $\overline{K_{RESP}}(\text{REQ:}BANK:RESP, C) = \overline{K_{BANK}}(\text{REQ:}RESP: BANK, C)$ and $\overline{K_{BANK}}(\text{REQ:}BANK:RESP, C) = \overline{K_{RESP}}(\text{REQ:}RESP: BANK, C)$ since the two cases are specular. ◀

► **Theorem 9.** The remaining latency $t_{ua}^f - t_{now}$ for a r_{ua} of type \mathcal{T} at time t_{now} is upper bounded by:

$$c_{init} + \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res} + \overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1), \quad (4)$$

where \mathcal{S}_{ua} is the set of all resources on which the C requests for nodes in the dependency DAG have not yet started executing, and $c_{init} = c_{REQ}$ if any node in the DAG executes on REQ , $c_{init} = 0$ otherwise.

Proof. By Lemma 6, the remaining latency of r_{ua} is equal to the latency of a critical path \mathcal{P} , which is upper bounded by Equation 1. Note that by definition, only the first node in \mathcal{P} might have already started executing at t_{now} . Hence, it holds: $\mathcal{S}_{\mathcal{P}} \subseteq \mathcal{S}_{ua}$, which implies $\sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} \leq \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res}$. Let C' be the number of requests for nodes in \mathcal{P} ; by definition, $C' \leq C$. By cases on the resource on which the first node in \mathcal{P} executes.

REQ: by definition we have $c_{init} = c_{REQ} = c_{\overline{res}}$ and by Lemma 8 and since $\overline{K_{BANK}}(\mathcal{T}, C)$, $\overline{K_{RESP}}(\mathcal{T}, C)$ are monothonic in C it holds $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(\mathcal{T}, C') \leq \overline{K_{BANK}}(\mathcal{T}, C)$, $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(\mathcal{T}, C') \leq \overline{K_{RESP}}(\mathcal{T}, C)$. Hence, the latency of \mathcal{P} in Equation 1 is upper bounded by Equation 4.

RESP: we have $c_{init} = 0$, $c_{\overline{res}} = c_{RESP}$ and by Lemma 8 and monotonicity it holds $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(\mathcal{T}, C) - 1$, $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(\mathcal{T}, C)$. Hence starting from Equation 1 and noting that by definition it must hold $c_{RESP} \leq t_{RESP} - 1$, we obtain:

$$\begin{aligned} & c_{RESP} + \sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} + K_{BANK}(\mathcal{P}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{P}) \cdot (t_{RESP} - 1) \\ \leq & c_{RESP} - (t_{RESP} - 1) + \sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} + \overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \\ & + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1) \\ \leq & \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res} + \overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1), \end{aligned} \quad (5)$$

hence the latency is again upper bounded by Equation 4.

BANK: we have $c_{init} = 0$, $c_{\overline{res}} = c_{BANK}$ and by Lemma 8 and monotonicity it holds $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(\mathcal{T}, C) - 1$, $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(\mathcal{T}, C)$; repeating the same derivation as for the RESP case, we again find that Equation 1 is upper bounded by Equation 4. \blacktriangleleft

5.2 Static Analysis

We compute the static worst-case latency $\Delta(\mathcal{T}, k_{ceil})$ by maximizing Equation 4 over all possible values of the parameters. The resulting bounds in Theorem 10 depend on k_{ceil} and M : by definition, there are at most M oldest request at any point in time, meaning that at most $M - 1$ requests can have higher GRR priority than r_{ua} . The theorem computes two separate bounds for $k_{ceil} = 0$ and $k_{ceil} > 0$: for the former, in the worst-case all M oldest requests target the same cache line, while for the latter, each oldest request targets a different cache line and suffers interference from k_{ceil} non-oldest requests based on arbitration Rule 4.

► **Theorem 10.** *The static latency for any request of type \mathcal{T} is upper bounded by:*

$$\begin{aligned} \Delta(\mathcal{T}, 0) &= t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot t_{BANK} + M \cdot t_{RESP} \\ &+ \overline{K_{BANK}}(\mathcal{T}, M) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, M) \cdot (t_{RESP} - 1) \end{aligned} \quad (6)$$

if $k_{ceil} = 0$, while if $k_{ceil} > 0$ it is upper bounded by:

$$\begin{aligned} \Delta(\mathcal{T}, k_{ceil}) &= t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot (k_{ceil} + 1) \cdot t_{BANK} + M \cdot (k_{ceil} + 1) \cdot t_{RESP} \\ &+ \overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{RESP} - 1). \end{aligned} \quad (7)$$

Proof. By definition, the static latency is equal to the maximum remaining latency of any request r_{ua} of type \mathcal{T} that becomes oldest at time t_{now} . By Theorem 9, such latency is upper bounded by Equation 4; hence, we can upper bound $\Delta(\mathcal{T}, k_{\text{ceil}})$ by maximizing Equation 4.

Note that the equation is maximized by using the maximum value $c_{\text{init}} = c_{\text{REQ}} = t_{\text{REQ}} - 1$, and latency contributions on all three resources such that $\mathcal{S}_{ua} = \{\text{REQ}, \text{BANK}, \text{RESP}\}$. By definition, the set \mathcal{R}_{REQ} can comprise at most r_{ua} itself and one oldest request for each other core; hence, we have at most $|\mathcal{R}_{\text{REQ}}| = M$.

We next consider the number of requests C for nodes in the dependency DAG of r_{ua} , which comprise r_{ua} and requests that precede r_{ua} in its eventual chain of dependencies, as well as the number of requests $|\mathcal{R}_{\text{BANK}}|, |\mathcal{R}_{\text{RESP}}|$. Let H be the number of oldest requests that precede r_{ua} in the eventual chain of dependencies. By Rule 4, the number of non-oldest requests that precede r_{ua} in the chain is bounded by k_{ceil} . Hence, the maximum number of requests in the dependency DAG is $C = k_{\text{ceil}} + H + 1$. Since requests in $\mathcal{R}_{\text{BANK}}, \mathcal{R}_{\text{RESP}}$ must have higher dynamic priority than r_{ua} , the only requests that can be included are: (1) the C requests in the DAG; (2) the remaining $M - H - 1$ oldest requests; (3) non-oldest requests that inherit the priority of such $M - H - 1$ oldest requests; again by Rule 4, their number is bounded by $(M - H - 1) \cdot k_{\text{ceil}}$. Summing over (1)-(3), we obtain: $|\mathcal{R}_{\text{RESP}}| = |\mathcal{R}_{\text{BANK}}| = M \cdot (k_{\text{ceil}} + 1) - H \cdot k_{\text{ceil}}$. If $k_{\text{ceil}} = 0$, then the cardinality of the sets is constant in H , while C , and thus the values of $\overline{K}_{\text{RESP}}(\mathcal{T}, C)$ and $\overline{K}_{\text{BANK}}(\mathcal{T}, C)$, are non-decreasing in H ; hence, Equation 4 is maximized when $H = M - 1$, yielding Equation 6.

Finally, consider the case of $k_{\text{ceil}} > 0$. Note that based on Equation 3, 2 and independently from \mathcal{T} , it holds: $\overline{K}_{\text{RESP}}(\mathcal{T}, k_{\text{ceil}} + 1 + H) \leq \overline{K}_{\text{RESP}}(\mathcal{T}, k_{\text{ceil}} + 1) + H$, and similarly $\overline{K}_{\text{BANK}}(\mathcal{T}, k_{\text{ceil}} + 1 + H) \leq \overline{K}_{\text{BANK}}(\mathcal{T}, k_{\text{ceil}} + 1) + H$. Substituting the values of the parameters in Equation 4 we thus obtain:

$$\begin{aligned}
& t_{\text{REQ}} - 1 + M \cdot t_{\text{REQ}} + M \cdot (k_{\text{ceil}} + 1) \cdot t_{\text{BANK}} - H \cdot k_{\text{ceil}} \cdot t_{\text{BANK}} \\
& + M \cdot (k_{\text{ceil}} + 1) \cdot t_{\text{RESP}} - H \cdot k_{\text{ceil}} \cdot t_{\text{RESP}} \\
& + \overline{K}_{\text{BANK}}(\mathcal{T}, k_{\text{ceil}} + 1 + H) \cdot (t_{\text{BANK}} - 1) + \overline{K}_{\text{RESP}}(\mathcal{T}, k_{\text{ceil}} + 1 + H) \cdot (t_{\text{RESP}} - 1) \\
\leq & t_{\text{REQ}} - 1 + M \cdot t_{\text{REQ}} + M \cdot (k_{\text{ceil}} + 1) \cdot t_{\text{BANK}} - H \cdot k_{\text{ceil}} \cdot t_{\text{BANK}} \\
& + M \cdot (k_{\text{ceil}} + 1) \cdot t_{\text{RESP}} - H \cdot k_{\text{ceil}} \cdot t_{\text{RESP}} \\
& + \overline{K}_{\text{BANK}}(\mathcal{T}, k_{\text{ceil}} + 1) \cdot (t_{\text{BANK}} - 1) + H \cdot (t_{\text{BANK}} - 1) + \\
& + \overline{K}_{\text{RESP}}(\mathcal{T}, k_{\text{ceil}} + 1) \cdot (t_{\text{RESP}} - 1) + H \cdot (t_{\text{RESP}} - 1) \\
\leq & t_{\text{REQ}} - 1 + M \cdot t_{\text{REQ}} + M \cdot (k_{\text{ceil}} + 1) \cdot t_{\text{BANK}} + M \cdot (k_{\text{ceil}} + 1) \cdot t_{\text{RESP}} \\
& + \overline{K}_{\text{BANK}}(\mathcal{T}, k_{\text{ceil}} + 1) \cdot (t_{\text{BANK}} - 1) + \overline{K}_{\text{RESP}}(\mathcal{T}, k_{\text{ceil}} + 1) \cdot (t_{\text{RESP}} - 1), \tag{8}
\end{aligned}$$

which is the expression in Equation 7. \blacktriangleleft

6 Applying Duetto to Cache Coherence Design

In this section, we first briefly review the Duetto reference model introduced in [25] to address the average performance and predictability trade-off in shared resource management in multi-core systems. We then discuss how the reference model must be extended to account for the peculiarities of our cache system and form the DUEPCO architecture.

6.1 Background: Duetto Reference Model

The key idea behind Duetto is that it augments a COTS High-Performance Arbiter (HPA) with a Real-time Arbiter (RTA) such that both arbiters operate in parallel. The RTA is analyzable in the sense that it provides strict latency bounds on requests; specifically, we

use the arbiter described in Section 4. The HPA is designed to maximize average-case performance. In our implementation, the HPA uses commodity FCFS at all resources but respects the dependencies among requests to the same cache line. Arbiters control resources by issuing commands—for our cache resource, the command dictates which requests, if any, should start executing on each resource. Every clock cycle, *Duetto* selects either the command from the HPA or the command from the RTA and issues it to the resources. Ideally, the system utilizes the HPA most of the time, hence benefiting from its performance gains, and only switches to the RTA if there is a risk that a latency guarantee will be missed. Note that the global request queue is shared by the HPA and RTA. Since both arbiters make decisions based on the requests stored in the queue, a request is removed from the queue only after it finishes based on the actual commands issued by *Duetto*. Given that both arbiters can process requests out-of-order, this does not add extra complexity to the queue implementation.

In more details, for each requestor P_i and each request type \mathcal{T} , *Duetto* associates a relative deadline $D_i(\mathcal{T})$ which represents the maximum tolerable processing latency for any oldest request of that type and requestor. Such deadline can be configured by the system designer (for example, by writing to memory-mapped registers exposed by the hardware), and must be used in place of the static WCL when performing task analysis. Higher deadline values increase the worst-case resource access latency for a task, but can enable *Duetto* to remain in HPA mode longer. A *DTracker* module is responsible for tracking the absolute deadline of each oldest request based on its associated relative deadline and the time it becomes oldest. As long as $D_i(\mathcal{T}) \geq \Delta(\mathcal{T}, k_{ceil})$ for all requestors and types, *Duetto* formally guarantees that all absolute deadlines will be met. This is achieved by using a Worst-case Latency Estimator (WCLator) module to estimate the remaining processing latency of each outstanding request at run-time, assuming that *Duetto* selects the HPA in the current clock cycle and then switches to the RTA in all future cycles. If for every requestor, the estimated finish time of its oldest request from WCLator is lower than or equal to its absolute deadline, then *Duetto* selects the HPA. Otherwise, it selects the RTA. The key observation is that using online information on the state of the system (resources, state of the RTA arbiter, and queued requests) allows us to greatly reduce the pessimism inherent in the static latency computation; hence, unless the system becomes overloaded, *Duetto* can keep selecting the HPA. Note that in [25], the model is demonstrated on a simple resource. However, in [26], *Duetto* has been applied to the design of a more complex DRAM controller where, similarly to our cache system, each request requires multiple commands to be serviced.

6.2 Model Extensions

Compared to the approach in [25, 26], we must extend the reference model in two fundamental ways to apply it to our cache design. First of all, it is important to notice that for the *Duetto* deadline guarantee to hold, the static worst-case latency must be computed assuming any valid state of the resource at the time t when the request under analysis becomes oldest; this is because the HPA might be selected at any time before t . However, when we computed the static latency in Theorem 10, we bounded the cardinality of sets \mathcal{R}_{BANK} and \mathcal{R}_{RESP} assuming that arbitration Rule 4 always applies, as this ensures that no more than k_{ceil} non-oldest requests can inherit the priority of an oldest request. Unfortunately, the HPA does not need to satisfy such rule, and can instead execute any number of requests to the same cache line on the REQ bus before an oldest request to that line arrives and its latency is considered by the WCLator; at which point it is too late to switch to the RTA.

To address this issue, we make a conceptual change to the model of the resource. Specifically, we declare that all states where there are more than k_{ceil} pending non-oldest requests to the same cache line are invalid. This ensures that the derived static bound is correct, but does not solve the underline problem as now the HPA might be issuing invalid commands. In [25], we suggest that when the HPA cannot be guaranteed to work correctly, a checker module can be added to check the validity of the commands issued by the HPA. Therefore, we add an additional checker component that works as follows: every clock cycle, the checker receives from the RTA information on the number of pending requests per cache line, which the RTA maintains to enforce Rule 4. If $c_{REQ} = 0$ and the global request queue contains at least one non-oldest request that must execute on the REQ bus and targets a cache line for which there are k_{ceil} pending non-oldest requests, the HPA might issue such request on REQ and reach an invalid resource state. Hence, in this case the checker overrides the WCLator to forcibly select the RTA. While this approach solves the unbounded priority inversion problem, it has a downside: for low values of k_{ceil} and/or heavy data sharing among requestors, the checker might be forced to continuously select the RTA, resulting in performance loss. We explore this behavior in more details in the evaluation Section 7.

The second extension is related to the request type. Both [25] and [26] assume that the type of a request is known when the request becomes oldest. However, in our system, the sequence of resources accessed by a request, and thus its type, is only known after the request is executed on the REQ bus and the owner of the corresponding cache line is determined. For this reason, before an oldest request finishes executing on REQ, the WCLator must use the smallest among all deadlines for the possible types for the request. Once the request type is known, the WCLator switches to using the deadline for that type.

6.3 WCLator Design

We designed the WCLator following the methodology outlined in [25]. For each oldest request and given the state of the resource and global request queue, we first enumerate all commands that the HPA could issue in this clock cycle. For each command, we then use the dynamic analysis of Theorem 9 (possibly with modified value of the parameters) to compute the remaining latency for the request. The WCLator then compares the largest computed latency against the deadline for the request to determine whether the HPA can be selected. As noted in Section 6.2, in our implementation we do not know the type of a request until it finishes executing on REQ. Hence, to safely account for the values of \mathcal{S}_{ua} , \mathcal{R}_{BANK} and \mathcal{R}_{RESP} in Equation 4, we assume that all outstanding requests that have not yet finished executing on REQ must access both BANK and RESP.

Consider an oldest request r_{ua} . To illustrate the behavior of the WCLator, we enumerate the cases assuming that $c_{init} = c_{REQ}$ (the case for $c_{init} = 0$ is similar but easier, since the chain of dependencies for r_{ua} cannot be affected):

1. If $c_{REQ} > 0$, then no command can be issued on REQ in this clock cycle, and no estimation is required. Note that if $c_{BANK} = 0$ or $c_{RESP} = 0$, the HPA could start executing a request on BANK or RESP in this clock cycle; however, because Equation 4 always assumes the worst case where the maximum blocking time is suffered on successive resources, it follows that the bound is still safe no matter the command issued by HPA on BANK and/or RESP. Therefore, for the remaining cases, we assume $c_{REQ} = 0$ and consider the command issued on REQ.
2. No command: the HPA might be non-work conserving and decide to issue no command in the current cycle. In this case, the bound is equal to Equation 4 plus one, to account for the wasted clock cycle.

3. r_{ua} : since r_{ua} will start executing, we would need to apply Equation 4 after removing it from \mathcal{R}_{REQ} , but setting $c_{REQ} = t_{REQ}$ to account for the r_{ua} execution. In addition, if there are higher-priority requests to the same cache line as r_{ua} which have not yet executed on REQ, we would need to remove such requests from $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$ and adjust $\overline{K}_{BANK}(\mathcal{T}, C), \overline{K}_{RESP}(\mathcal{T}, C)$. Note that the obtained bound will always be lower than case 2); therefore, in practice the WCLator does not need to consider this case.
4. A lower-priority request $r_{i,j}$, which does not inherit a higher priority than r_{ua} after becoming pending: we use Equation 4 with $c_{REQ} = t_{REQ}$.
5. A lower-priority request $r_{i,j}$ that inherits a higher priority than r_{ua} : in addition to the previous case, we need to include the request in $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$. Furthermore, if $r_{i,j}$ targets the same cache line as r_{ua} , $\overline{K}_{BANK}(\mathcal{T}, C), \overline{K}_{RESP}(\mathcal{T}, C)$ must be adjusted.
6. A higher-priority request $r_{p,q}$ to either the same or a different cache line than r_{ua} : we use Equation 4 with no change to parameters. Since this bound is always lower than 2), again we do not need to consider it.

In this work, our goal is to evaluate the performance of the proposed system architecture and arbitration scheme based on cycle-accurate simulation; a full system implementation is deferred to future work. Nonetheless, given its potential complexity, we briefly comment on the WCLator implementation. Since the WCLator is a hardware component, all the cases above can be estimated in parallel for each request under analysis, and each result compared against the request deadline. Hence, the hardware latency is dominated by the computation of Equation 4. The value of c_{REQ} can be maintained by a simple counter. Similarly, the value of $|\mathcal{R}_{res}|$ can be maintained in separate counters for each r_{ua} and each resource based on the state of the GRR arbitration. Multiplications can be avoided by storing the corresponding values in look-up tables indexed by the corresponding parameter. Another counter can be used to keep track of the number of requests C in the eventual chain of dependencies for each r_{ua} , and index a look-up table that returns the value of $\overline{K}_{BANK}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \overline{K}_{RESP}(\mathcal{T}, C) \cdot (t_{RESP} - 1)$. The final computation is then obtained by adding at most 5 terms together, which can be performed with cascaded adders of width at most 11 bits for $M = 8$ requestors. In summary, we do not expect the WCLator implementation to constrain the clock speed.

7 Evaluation Results

We employed the open-source, cycle-accurate architectural simulation framework provided by [15] to evaluate the performance of the proposed mechanisms and compare them with other solutions. We emulate quad- and octa-core systems clocked at 2.5 GHz. Each system has a 32 KB 4-way set-associative per-core private L1 data cache (similar to ARM Cortex A53 [1]), and a 4 MB 8-ways set-associative L2 shared cache consisting of multiple separated banks. The cores are OOO and can issue up to 10 memory requests in parallel. Both L1 and LLC have a cache line size of 64 bytes. Each core/LLC bank is equipped with a dedicated cache controller that implements the MSI coherence state machine. We compare results for the proposed arbiter described in Section 4, which we denote as RTA, against state-of-the-art approaches including PMSI [10], PMSI* [17] and PISCOT [15] which also provide analytical WCL bounds and present the best average-case performance. PMSI employs unified bus architecture and provides relative high-performance gains compared to other approaches such as shared data-aware scheduling and private cache bypassing through deploying cache coherence modifications and accessing the shared data. However, its WCL is quadratic in the number of cores in the system. PMSI* follows a systematic approach that achieves the same

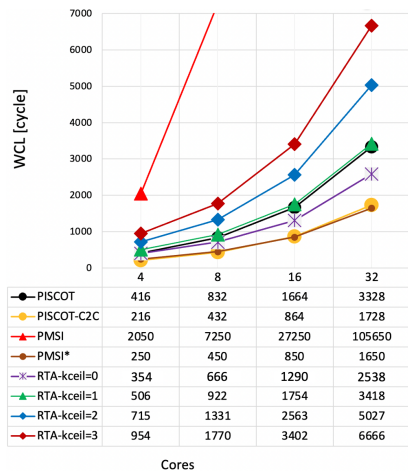


Figure 7 Per-request worst-case latency.

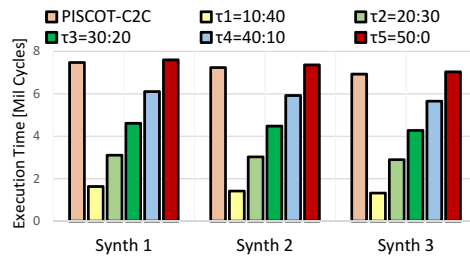


Figure 8 Sensitivity test for RTA against PISCOT-C2C.

static WCL as bypassing the shared cache and provides a tighter WCL bound compared to PMSI. However, both of these techniques rely on many coherency modifications and expose performance loss compared to other approaches. On the other hand, PISCOT decouples the request and response bus and leverages the split-transaction interconnect to achieve a tighter WCL compared to PMSI and considerable performance gains. We also compare against the COTS HPA as described in Section 6.1, which aims to achieve high average-case performance.

Request bus latency is configured to 4 cycles ($t_{REQ} = 4$). The response bus latency in PISCOT is comparable to the TDM slot size in PMSI as well as PMSI* and we set them to 50 cycles in our evaluation similar to [15]. However, for RTA, the latency of all resources is configurable. Throughout this section, unless otherwise specified, we configure RTA with $t_{RESP} = 10$ cycles, 8 banks that consume $t_{BANK} = 40$ cycles to process requests and parameter $k_{ceil} = 1$. Similar to existing works [18, 10, 15], we assume that accesses that hit in the L1 cache take a single clock cycle and, as discussed in Section 3.1, LLC is a perfect cache (all LLC accesses are hits) to avoid extra delay from accessing the off-chip memory subsystem.

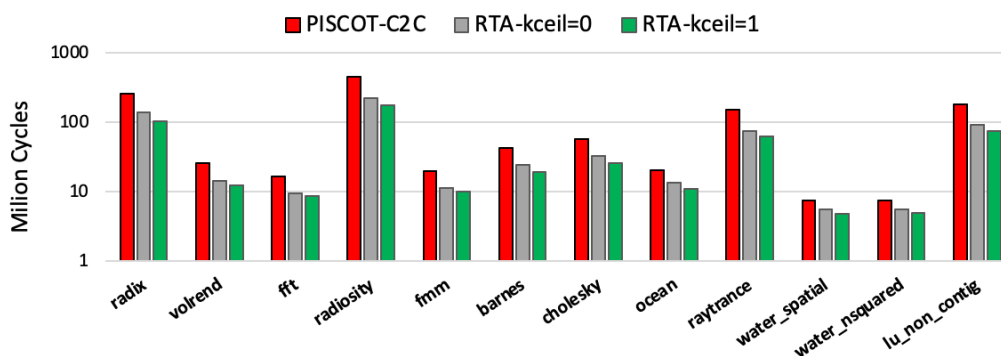
We employ SPLASH-3 [31] benchmark suite as it is a representative of multi-threaded applications with data sharing. In addition, we craft a wide set of synthetic benchmarks that stresses the implemented solutions. All contain mixed read and write requests to the LLC and we engineered the requests' addresses such that all requests miss in the L1 cache; hence, stress on the bus and the shared cache banks will be maximized. Different benchmarks in this set exhibit different sharing percentage as well. Due to space limitation, we show the results of three of these benchmarks (Synth 1, Synth 2, Synth 3) that show unique insights. There is no data-sharing among the cores in Synth 1 while Synth 2 and Synth 3 exhibit 10% and 20% data-sharing respectively. In all benchmarks, the foreground core represents a high load core that bursts requests to bus/LLC, and the background cores are accessing the shared bus/LLC less frequently. Interleaving across the banks is handled using address bits themselves such that a core could access all banks as much as possible. In detail, we use bit 6th (bits zero to 5th are for the cache line offset) towards the MSB in the address bits of the request to determine which LLC bank it needs to be processed in.

7.1 Per-Request Worst-Case Latency

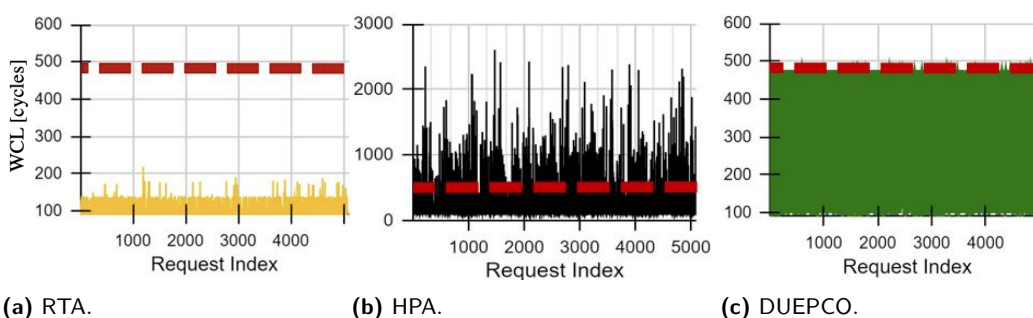
Figure 7 shows the static WCL bounds for requests generated by the cores and misses in L1 caches (see Section 5) from REQ:RESP:BANK type which represents the largest static WCL among the three types. We compare PMSI, PMSI*, PISCOT and PISCOT-C2C (with core to core transfers), and the proposed RTA mechanism with different values of the configurable parameter k_{ceil} . From this experiment, we can make the following observations: 1) PMSI shows a significantly higher latency bound compared to the other approaches, and the latency bound increases quadratically with scaling the number of cores. The significant added latency is due to the coherence interference on the shared data. PMSI* on the other hand presents tight static WCL bound but at the cost of performance degradation [15, 17]; 2) PISCOT shows looser bound compared to both PISCOT-C2C and PMSI* but similar to RTA since core to core transfers enable the arbiters to bypass the LLC when an owner core must respond to other cores; 3) RTA with $k_{ceil} = 1$ shows up to $1.18\times$ looser bound compared to PISCOT but significantly tighter bound compared to PMSI. Notice that this extra amount in latency bound is due to the scheduling decisions that are made in RTA which allow one non-oldest request to process in LLC banks. This gives the system a significant advantage in terms of average performance as we will show in the next sections. It is worthwhile to stress the existing trade-off between RTA with different values of k_{ceil} and PISCOT-C2C. RTA with $k_{ceil} = 0$ represents a configuration that forces the ordering of processing such that requests are only processed if they are oldest (no non-oldest request is allowed to process in the shared banks). This improves the WCL bound such that it becomes tighter and very similar to PISCOT-C2C. However, Duetto does not work with this configuration of RTA ($k_{ceil} = 0$) since this prevents the system from reasonably leveraging the performance of the HPA; the checker module is forced to select the RTA if there is any non-oldest request needing to be serviced on the request bus.

7.2 RTA Sensitivity Test

The underlying architecture proposed in Section 3 is fully configurable to resemble the conventional high-performance bus/LLC designs. In this section, we conduct a sensitivity test on the RTA using synthetic benchmarks to justify the most efficient (and the worst) design that is aligned with commercial architectures and compare it against PISCOT-C2C. We configured a quad-core system with $t_{REQ} = 4$ and then gradually varied t_{BANK} and t_{RESP} latencies. In order to run a fair comparison, the parameters are determined such that $t_{RESP} + t_{BANK} = 50$, the response bus latency for PISCOT-C2C. Assuming $\tau = t_{RESP} : t_{BANK}$ represents a configuration of RTA in which the latency of shared banks in LLC is t_{BANK} and the latency of response bus equals t_{RESP} , Figure 8 shows the execution time of the foreground core running each of the three synthetic benchmarks. As discussed, RTA increases the parallelism through bankized LLC. Therefore, as we increase t_{BANK} in LLC and coincidentally decrease t_{RESP} , we observe that the system performance improves by finishing the task under analysis faster. In other words, by reducing the response bus latency, a significant amount of arbitration stress will be transferred to the banks rather than the response bus; hence, the system's overall performance increases by allowing more transactions to be serviced simultaneously. In detail, the core under analysis in RTA, τ_1 running `Synth 1` outperforms PISCOT-C2C by $4.58\times$ in terms of overall throughput of the system. Note that in τ_5 where there is no parallelism in RTA, we observe a negligible performance loss compared to PISCOT-C2C (maximum 1% in overall throughput) since response bus arbiter in PISCOT-C2C is FCFS while RTA employs a fair round-robin mechanism through GRR. Going forward, we chose τ_1



■ **Figure 9** Total observed memory latency of Splash-3. Values in y-axis are in log scale.



(a) RTA.

(b) HPA.

(c) DUEPCO.

■ **Figure 10** Observed latencies under different arbitration schemes.

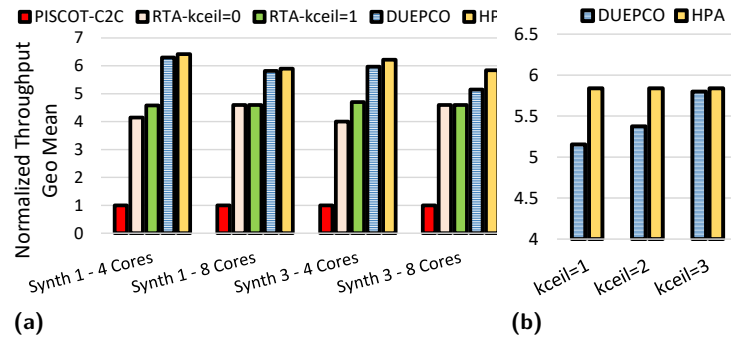
as it resembles the configuration with a higher level of parallelism resembling a more realistic architecture. For example, Intel’s architectures utilize a bankized shared cache to hide the shared bank processing time, which is higher than the data transfer on the bus [2].

7.3 Average Performance: SPLASH-3

We next evaluate the average performance of RTA against PISCOT-C2C based on SPLASH-3 benchmarks. Figure 9 shows the cumulative processing latency of all memory requests generated by a quad-core system. Overall, RTA shows an average latency reduction of $1.74\times$ compared to PISCOT-C2C for $k_{ceil} = 0$, and of $2.1\times$ for $k_{ceil} = 1$. This shows that even for realistic benchmarks, bankizing L2 leads to a significant improvement in the performance of the memory subsystem. Processing non-oldest requests leads to further performance improvements, but as previously noted based on Figure 7, this comes at the cost of increased WCL bounds.

7.4 Observed Request Latency

In the last two sets of experiments, we focus on the behavior of our Duetto design, which we call DUEPCO, compared to the RTA and HPA. Note that we configure the relative deadline for each type of request to be equal to its static WCL bound. Figures 10a, 10b, and 10c delineate the observed latency in number of cycles suffered by oldest miss requests of type `REQ: BANK: RESP` generated by a quad-core system. We show request latencies greater than 80 cycles for better visibility and run the experiment with `Synth 3` benchmark (other benchmarks/request types show similar behavior). The RTA latency bound for this setup is



■ **Figure 11** Total throughput of the system.

476 cycles based on the derived WCL analysis in Section 5 which is shown as a red bar in the figures. In HPA, we observe large latency spikes throughout the execution up to 3420 cycles since HPA favors requests from the cores that generate the highest number of requests, are faster, and target the banks that are idle which can starve (theoretically) or delay for a long time (practically) requests targeting busy banks. Figure 10a shows that RTA respects the latency bound for all requests from every core and the latencies are always below the WCL bound. However, there is a gap between the latencies and the static WCL bound since static analysis conducted in Section 5 must assume that the oldest requests of all cores access the same bank at the same time in addition to the non-oldest requests, which is unlikely in practice. Finally, Figure 10c shows that DUEPCO stretches the latency of requests towards the latency bound, but the bound is never violated. This allows the system to continue selecting the HPA as long as possible.

7.5 Average Performance: Synthetic Benchmarks

To measure the average performance of DUEPCO, we use the total throughput of the system based on synthetic benchmarks to stress the system. Figure 11a shows the geometric mean of throughput across all cores for RTA, HPA and DUEPCO normalized to the overall throughput of PISCOT-C2C. The figure represents the results for four different setups: 1) a quad-core system running `Synth 1`; 2) a quad-core system running `Synth 3`; 3) an octa-core system running `Synth 1`; 4) an octa-core system running `Synth 3`. We make the following observations: 1) RTA, HPA, and DUEPCO outperform the single-bank architecture approach deployed in PISCOT-C2C significantly, by up to $6.4\times$; 2) DUEPCO shows very small slowdown compared to HPA in `synth 1` and `synth 3 - 4 core` (at most 2%); 3) in an octa-core system and `synth 3` benchmark, we observe a slowdown of 11%. Following the discussion in Section 6, since DUEPCO employs RTA with $k_{ceil} = 1$, it has to exclude the invalid states from the HPA by switching to RTA. Recall that `Synth 3` benchmark exposes 20% data-sharing among the cores, and this leads to the case that multiple cores compete to access the same cache line in a particular bank. Therefore, DUEPCO selects the RTA regardless of the WCLator estimation according to the checker logic. However, by increasing the number of allowed requests to the same cache line (k_{ceil}), we expect that DUEPCO selects the HPA more often. As shown in Figure 11b, DUEPCO that employs RTA with $k_{ceil} = 3$ exhibits only 1% slowdown compared to HPA. Notice that relaxing the parameter k_{ceil} forces us to use a higher value for the static WCL bound for each oldest request as shown in Figure 7. Therefore, we do not consider higher values for the parameter.

8 Conclusions

Employing shared memory in multi-core platforms improves programmer productivity and degrades the obstacle to using such platforms in real-time systems. Hardware cache coherence can accommodate such shared memory and extend the advantages of on-chip caching to all system memory. However, extending hardware cache coherence throughout traditional schemes such as coherency protocol modifications to provide predictability hurts the performance of the system. In this work, we demonstrate that by employing the COTS interconnect architecture along with proposing to bankize the on-chip cache, DUEPCO is able to pair a clever global arbitration mechanism with Duetto to significantly improve the performance of the system while providing predictability. Notice that while we propose DUEPCO with simple buses, potentially the same arbitration scheme could be added to other bus architectures such as AXI in ARM platforms. However, the fundamental constraint to consider is that the arbiter must have exclusive visibility into the queues of each requestor.

References

- 1 Arm cortex-a53 mpcore processor technical reference manual r0p3. <https://developer.arm.com/documentation/ddi0500/e/level-1-memory-system/about-the-l1-memory-system>. Accessed: 2022-01-23.
- 2 Intel® 64 and ia-32 architectures optimization reference manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>. Accessed: 2021-07-20.
- 3 ARM. Arm® cortex®-r8 mpcore processor. <https://developer.arm.com/documentation/100400/0001/xdc1471434436160>, 2019.
- 4 Matthias Becker, Dakshina Dasari, Borislav Nikolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 14–24. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.14.
- 5 Micaiah Chisholm, Namhoon Kim, Bryan C Ward, Nathan Otterness, James H Anderson, and F Donelson Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 57–68. IEEE, 2016. doi:10.1109/RTSS.2016.015.
- 6 Giovanni Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, pages 27:1–27:25, Dagstuhl, Germany, 2019.
- 7 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Trans. Embed. Comput. Syst.*, 17(2), February 2018. doi:10.1145/3158208.
- 8 Mohamed Hassan. Heterogeneous mpsoCs for mixed-criticality systems: Challenges and opportunities. *IEEE Design & Test*, 35(4):47–55, 2017.
- 9 Mohamed Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 10 Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246. IEEE, 2017.
- 11 Mohamed Hassan and Hiren Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016. doi:10.1109/RTAS.2016.7461327.

- 12 Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316. IEEE, 2015.
- 13 Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018. doi:10.1109/TCAD.2018.2857379.
- 14 Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsocs. In *Euromicro Conference on Real-Time Systems*, 2020.
- 15 Salah Hessian and Mohamed Hassan. The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–230, 2020. doi:10.1109/RTSS49844.2020.00029.
- 16 Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 70(12):2098–2111, 2021. doi:10.1109/TC.2020.3037747.
- 17 Anirudh Mohan Kaushik and Hiren Patel. A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–117, 2021. doi:10.1109/RTAS52030.2021.00017.
- 18 Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432, 2019. doi:10.1109/RTSS46320.2019.00044.
- 19 Manpreet S Khaira. Fast first-come first served arbitration method, November 12 1996. US Patent 5,574,867.
- 20 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014. doi:10.1109/RTAS.2014.6925998.
- 21 Namhoon Kim, Micaiah Chisholm, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–234, 2017. doi:10.1109/RTAS.2017.14.
- 22 Benjamin Lesage, Isabelle Puaut, and André Sez nec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 171–180, 2012.
- 23 Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters*, 19(2):105–109, 2020. doi:10.1109/LCA.2020.3008288.
- 24 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Drambulism: Balancing performance and predictability through dynamic pipelining. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94, 2020. doi:10.1109/RTAS48715.2020.00–15.
- 25 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1136–1141, 2021. doi:10.23919/DATE51398.2021.9474062.
- 26 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance. In *ACM International Symposium on Memory Systems (MEMSYS)*, pages 1–14, 2021.
- 27 NXP. Qorlq@ t4240, t4160 and t4080 multicore processors, 2018.
- 28 Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. *ACM SIGARCH Computer Architecture News*, 37(3), 2009.

- 29 Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *2008 Real-Time Systems Symposium*, pages 221–231, 2008. doi:10.1109/RTSS.2008.42.
- 30 Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.
- 31 Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.
- 32 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.
- 33 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling cache coherence to expose interference (artifact). In *Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, 2019.
- 34 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. On how to identify cache coherence: Case of the nxp qoriq t4240. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.
- 35 Ashok Singhal, Bjorn Lienres, Jeff Price, Frederick M Cerauskis, David Broniarczyk, Gerald Cheung, Erik Hagersten, and Nalini Agarwal. Implementing snooping on a split-transaction computer system bus, November 2 1999. US Patent 5,978,874.
- 36 Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.
- 37 Nivedita Sritharan, Anirudh Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445, 2019. doi:10.1109/RTSS46320.2019.00045.
- 38 Calvin K Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753, 1976.
- 39 Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383, 2013. doi:10.1109/RTSS.2013.44.
- 40 Zhuanhao Wu, Anirudh Mohan Kaushik, Paulos Tegegn, and Hiren Patel. A hardware platform for exploring predictable cache coherence protocols for real-time multicores. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 92–104, 2021. doi:10.1109/RTAS52030.2021.00016.

Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems

Mohamed Hossam ✉🏠

McMaster University, Hamilton, Canada

Mohamed Hassan ✉🏠

McMaster University, Hamilton, Canada

Abstract

The adoption of multi-core platforms in embedded real-time systems mandates predictable system components. Such components must guarantee the satisfaction of the timing constraints of various applications running on the system. One of the components that can break the system predictability is cache coherence, which ensures the correctness of shared data. This paper proposes a solution towards the enablement of predictable cache coherent real-time systems. The solution uses existing COTS coherence protocols and proposes a methodology to integrate them with legacy real-time arbiters without imposing any required modification to either of them. Doing so, the paper also works as an exploratory study of the integration of various coherence protocols with various predictable arbitration schemes leading to a total of 12 different architecture configurations. Evaluation against four state-of-the-art predictable coherence solutions as well as COTS-based solutions show that the proposed approach achieves the tightest existing latency bounds among predictable solutions with minimal performance degradation over the COTS ones.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Real-time system architecture; Computer systems organization → Multicore architectures

Keywords and phrases Coherence, Shared Data, Caches, Multi-Core, Real-Time, Memory

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.17

Supplementary Material PCC-sim, an open source cache simulator

Software: <https://gitlab.com/FanosLab/pcc-sim>

1 Introduction

Compared to traditional real-time embedded systems, modern applications of embedded systems impose fundamentally new set of challenges. Examples of these applications are prevalent in domains such as automotive, unmanned air vehicles (UAVs), space, and industry 4.0. With the transition towards more autonomy in these domains, the challenges are correlated to processing massive amounts of data, which requires unprecedented computation power as well as memory bandwidth. Moreover, this data processing must happen as quick as its incoming rate from the external physical environment (e.g. camera frame rate in a self-driving car or an incoming vital signal from a patient). This dictates a minimal acceptable average performance of the computing system. Meanwhile, the fact that these are safety-critical systems, they have the stringent requirement of predicable performance. This is expressed in terms of deadlines that should never be exceeded under all conditions. To be able to provide this predictable performance, the hardware architecture itself should be predictable to enable the derivation of reasonable bounds on the worst-case execution time (WCET) of all running tasks.

Architecting computing systems to meet all the aforementioned requirements becomes a challenging task since they can conflict with each other. An example of such conflict that is relevant to the focus of this paper is data sharing. Most existing research in the predictable



© Mohamed Hossam and Mohamed Hassan;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 17; pp. 17:1–17:23



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

management of multi-core hardware assumes tasks do not share data [8]. This is because data sharing can infringe the isolation properties promised by these techniques [18]. Nonetheless, this is no longer possible for the aforementioned domains since inter-task communication through data sharing is a key to the functionality of these systems [28, 17]. For example, in automotive, measured values from several sensors need to be consumed by multiple functions [11].

Considering average performance requirements, cache coherence seems to be the appealing solution to enable data sharing both from academic [22, 23] and industry [9] perspective. Therefore, this paper focuses on cache coherence as the data sharing mean in multi-core real-time systems. Despite having a recent attention from the real-time community, the approach followed by the majority of existing works is to ensure predictability by mandating modifications to either the commercial-off-the-shelf (COTS) coherence protocols only [14], the legacy predictable interconnects or arbitration schemes [17], or both [15, 18, 27, 20, 19]. **In contrast, this paper contributes to the efforts of enabling predictable and coherent sharing of data in real-time systems by proposing PCC: a solution that integrates COTS coherence protocols into legacy predictable real-time arbitration schemes without requiring any modifications to either of them.** This solution is based on the following observation. The usually abstracted bus architecture in the real-time research is physically composed of several parallel buses in COTS platforms. In particular, the interconnect between private caches and the shared cache either deploys a communication bus for the communication among the private caches, which is separate from the bus connecting these private caches to the shared cache [3], or it enables several point-to-point connectivity that allows for overlapping transfers [1, 7]. *This enables the data to be sent to different destinations simultaneously; in particular, from a core's private cache to another private cache as well as to the shared memory.* A more thorough discussion about this observation is presented in the system model in Section 4. Leveraging this observation, the paper makes the following contributions.

- 1) It illustrates how to predictably integrate COTS coherence protocols into the legacy predictable real-time arbiters without imposing any architectural modifications to the protocol itself nor to the underlying predictable arbitration scheme. This is key since it has been established by prior works that directly doing so will lead to unpredictable behaviors [18]. However, we show how exploiting the architectural capability mentioned in the previous observation can restore predictability to the real-time multi-core system upon integrating cache coherence to it.
- 2) The predictability of the solution is proven by a formal timing analysis that we introduce in Section 6. Unlike existing works, this analysis is generalized to apply to various real-time arbiters as well as various COTS coherence protocols. Additionally, a key aspect of this work is that the provided bounds stand the same regardless of the pipeline architecture of the cores, whether in-order or out-of-order (OoO).
- 3) To confirm the claimed integrability, we deploy a wide set of COTS coherence protocols as well as predictable arbitration schemes. In addition to the modified-shared-invalid (*MSI*) protocol that is adopted by most existing works, we also fully implement the *MESI* (E refers to Exclusive) and *MOESI* (O refers to Owned) protocols, which, unlike the simple *MSI* protocol, are common on multi-core platforms. For instance, the *MESI* protocol is adopted by the ARM's most-recent Cortex-R82 [4], while the *MOESI* is adopted by ARM's A53 processor [2]. For predictable arbiters, we exemplify the generality of the solution by implementing time division multiplexing (TDM), round robin (RR), weighted RR (WRR), and harmonic RR (HRR). This results in 12 different implemented and studied cache coherent architectures.

- 4) These rich system configurations enable us to conduct extensive case studies, which in turn lead to several novel observations about the various design trade-offs of choosing the coherence protocol as well as the arbitration mechanism from the predictability perspective of multi-core real-time systems. These observations are discussed in details in Section 7.
- 5) We compare against four predictable cache coherent techniques [15, 18, 19] as well as against conventional COTS coherence techniques. Results show that PCC is able to achieve the tightest bound for existing predictable coherence solutions, with a minimal performance degradation compared to COTS solutions.

2 Background

In this section, we cover the fundamentals of cache coherence protocols and shared bus arbitration.

2.1 Coherence Protocols

A Coherence protocol is the mechanism that cache controllers employ in multi-core systems to ensure the correctness of the data. The correctness is achieved by guaranteeing that all the cores have access to the latest version of the data. Thus, coherence protocols enforce Single-writer-multiple-reader (SWMR) invariant to keep the coherency of the data. The basic protocol that many COTS architecture implements is *MSI*.

MSI protocol consists of three main states: Modified(M), Shared(S), and Invalid(I) where each cache line in the private cache should be either in one of these states or in the transition to one of them. M state grants read and write permissions to the core that has the cache line. Due to SWMR invariant, only a single core can have a certain cache line in M state at a time, and other cores cannot privately cache this line during this time. Cores can request lines for modification by issuing *GetM* message on the shared bus. On the other hand, S state is a read only state, where multiple cores can have the same cache line in this state. Cores request lines for read by broadcasting *GetS* message on the bus. The last state is I which indicates that the data of a cache line is not available in the private cache or the data is stale. I state does not allow reading or writing to the data.

Extensions can be applied to *MSI* by adding one or more states such as the exclusive (E) and the owner (O) state. These extensions result in the famous protocols: *MESI*, *MOSI*, and *MOESI*. E state is similar to S state as both are read only states, but E state indicates that only one core has this cache line in its private cache. This allows such core to move from E to M silently without issuing *GetM* message. The other state, O is also a read only state, but it gives the core the ownership of the cache line, meaning this core should respond to other cores' requests for this line instead of the shared memory.

Transient States. Besides these standard states, which are called **stable** states, there are number of states that indicate the transitions between the stable states, and they are called **transient** states. Transient states are crucial due to the non-atomicity of the interconnect between cache memories. For instance, if core C_i requires to write to a cache line in the I state, it will issue a *GetM* message and wait for receiving the data. Before the data is received, the cache line can neither be in the I nor the M states; therefore, a transient state is required to define this transitional period. Conventionally, this state is named IM^d , which shows the source and destination states, and the superscript indicates the reason of the state (d indicates waiting for data). In order to complete this example, we can assume that while C_i is waiting for data, it observes a *GetM* message from another core C_j . Accordingly, C_i should change the state of the line from IM^d to IM^dI . IM^dI indicates that after the data arrives and write is performed, the cache line should be moved to I state. Some of these transient states are depicted in Figure 2 and discussed within the example in Section 3.

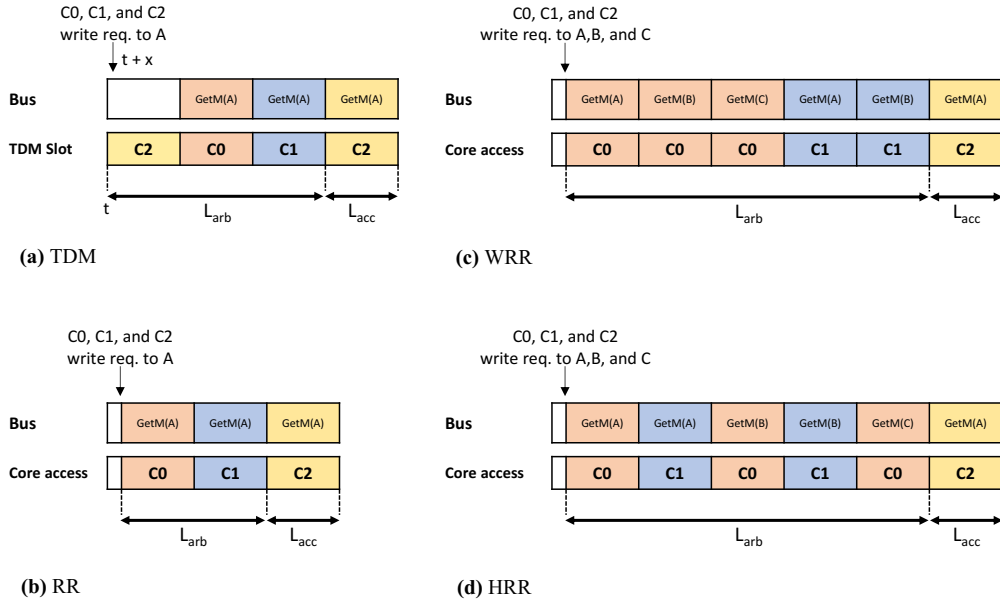


Figure 1 The worst-case latency of different bus arbiters. C_2 is the core under analysis.

2.2 Bus Arbitration

In embedded systems, the communication medium between private caches from one side and the shared memory from the other side is usually a shared bus. A bus arbiter is responsible for organizing accesses of the cores to the bus. Figure 1 shows the worst-case latency (WCL) of different predictable arbiters. A predictable arbiter should grant access to the bus to the requesting core in a predictable bounded latency. Figure 1a shows the latency of TDM arbiter, which dedicates a fixed time slot for each core to access the bus. The WCL of TDM is govern by $WCL = N \times S$, where N is the number of cores and S is the slot width. The width of the slot is chosen to have $S \geq (L_{arb} + L_{acc})$, where L_{arb} is the bus latency to broadcast a request and L_{acc} is the latency of data transfers over the bus. Similarly, Figure 1b shows the latency of RR arbiter, which is a dynamic arbiter that keeps a cyclic list of cores with ready requests. WCL of RR is calculated by $WCL = (N - 1) \times (L_{arb} + L_{acc})$. There are multiple variants of RR, the most common are WRR and HRR. Both WRR and HRR allow cores to have different arbitration weights. WRR’s WCL can be calculate as shown in Figure 1c, for a core C_j WCL is $\sum_{i=0, i \neq j}^{N-1} W_i \times (L_{arb} + L_{acc})$, where W_i is the weight of core C_i . WCL of HRR is calculated differently, as shown in Figure 1d the weights of the cores are distributed harmonically, therefore the WCL of C_j in HRR can be calculated using $(\lceil HP/W_j \rceil - 1) \times (L_{arb} + L_{acc})$, where HP is the hyper-period of the cyclic list of the cores, and it is equal to the summation of all the weights.

3 Related Work and Motivation

In the way of adopting the multi-core systems in real-time applications, many efforts have been conducted to facilitate this adoption by predictably managing interference among different cores upon accessing shared hardware resources in the system. Examples of these resources are caches [5, 12, 14, 15, 18, 19, 21, 27], interconnects [17, 20], and main memory [10, 13, 16]. Among these works, the most relevant to this paper are the ones focusing on enabling coherent data sharing through hardware coherence protocols [14, 15, 18, 19] and the ones aiming at predictably arbitrating accesses to the shared cache [17, 20].

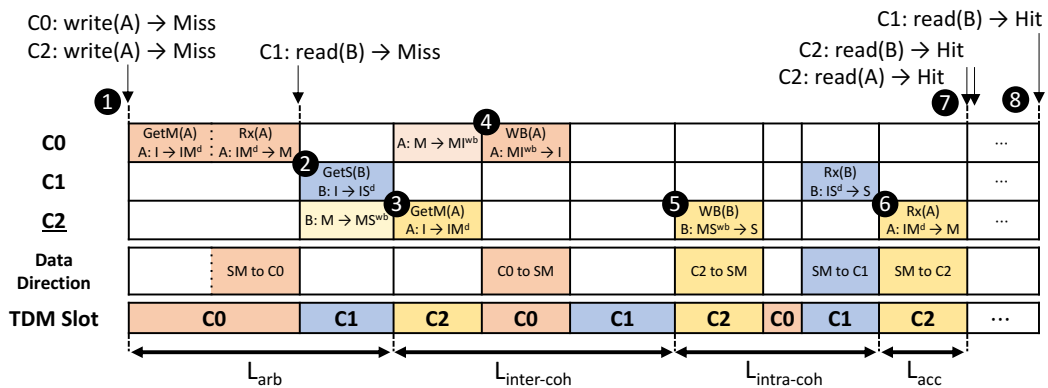
PMSI [15] provides predictable cache coherence by modifying the legacy MSI protocol and implementing it on top of a unified TDM bus. The work was extended in [18] by adopting a $MESI$ -based solution. PMSI (and PMESI) suffers from two main drawbacks. 1) Its WCL is quadratic in the number of cores, which makes it very pessimistic. 2) It requires modifications to both the COTS coherence protocol and the underlying architecture. The first drawback hinders its usability for real-time systems with tight latency requirements, while the second one makes it hard to adopt by industrial entities since designing and verifying new coherence protocols is known to be one of the most tedious architectural tasks [23, 24].

In order to elaborate PMSI's operation graphically, we use the example in Figure 2. The example crafts a scenario that highlights the key features and drawbacks of PMSI. Moreover, it is used to explain the rest of the related approaches, and in Section 5 we utilize the example to present PCC's operation and how it tackles the other approaches' downsides. The example shows the different latency components of the write request from core C_2 to the memory location A . Initially, at ❶ cores C_0 and C_2 have write requests to A which is cached in the shared memory. At ❷, C_1 has a read request from the memory location B which is modified by C_2 . Afterwards at ❸, C_2 has a read request from A followed by a read request from B . Towards the end at ❹, C_1 requests another read from B .

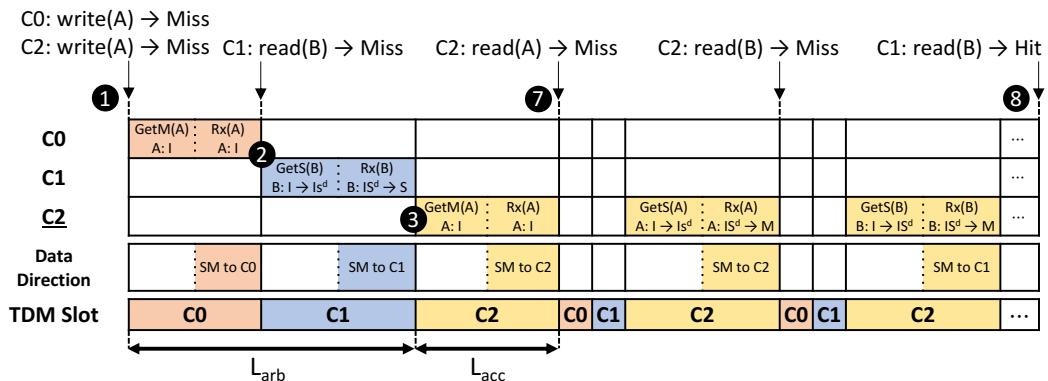
The breakdown of PMSI's latency is illustrated in Figure 2a, where cores issue their request messages only during their dedicated TDM slots. C_0 issues a **GetM** message at ❶ and receives the data from the shared memory in the same slot. On the other hand, C_2 issues its message at ❸ which entails that C_0 changes A 's state from **M** to **M^{wb}** according to PMSI protocol. **M^{wb}** indicates a transition from **M** state to **I** state after writing data back to the shared memory. Regarding the request latency, its first component is L_{arb} , which is the time between a core having a request until the request is broadcast on the bus. Since PMSI does not allow cache-to-cache transfers, C_2 has to wait for a complete TDM period to receive A after C_0 writes it back to the shared memory at ❹. The duration between the request broadcast and the data being ready to be sent (the time between ❹ to ❺) is the inter-coherence latency ($L_{inter-coh}$), which appears because of the coherence interference between the cores, such as the write-back from C_0 . Nonetheless, C_2 cannot receive A at ❺ because of the previous request from C_1 at ❷ which requires C_2 to write B back to the shared memory. Accordingly, C_2 receives A at ❻ instead of ❺, and this delay is defined as the intra-coherence latency ($L_{intra-coh}$) that results from the intra-core interference between a core's demanding requests and its write-backs. The last latency component is L_{acc} , and the total latency per request is the summation of all the aforementioned components. Clearly, the WCL of such behavior is very pessimistic as the latency of c_2 's **write(A)** illustrates. Finally, the requests at ❼ and ❽ are all hits, because PMSI allows read from the modified cached lines (such as A) as well as previously modified lines (such as B). These two features, which are inherited from MSI , result in a relatively good average-case performance as shown in Table 1.

DISCO [14] makes the observation that write requests are the reason of PMSI's excessive WCL due to the need for write-backs. Hence, it proposes to tighten this WCL by discriminating between read and write requests. This is done by prohibiting modified cache lines from being stored in the private L1 caches of cores; instead, all write requests must be serviced at the shared memory directly. The tight WCL can be observed from Figure 2b, where the write requests from C_0 and C_2 are serviced during the same slots they are issued in at ❶ and ❸. Moreover, C_1 's request at ❷ is serviced directly from the shared memory, unlike the case of PMSI, because C_2 is not allowed to privately cache the modified B line. Nonetheless, WCL comes at the cost of average performance which is embodied in the misses of the read

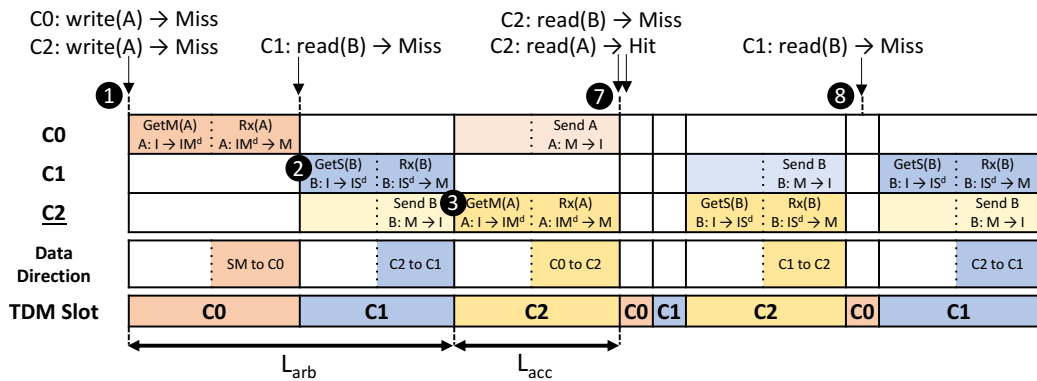
17:6 Predictable Cache Coherence



(a) PMSI/PMESI



(b) DISCO



(c) PMSI*

■ **Figure 2** The memory latency of the write request to *A* from core *C*₂ in a 3-core system that implements unified bus to connect private cache with the shared memory. The bus uses TDM arbitration with fixed slot width; however, the figure shows different sizes for the slots in order to fit into the page width. In the beginning of the scenario, *A* resided in the shared memory, while *B* was modified by *C*₂.

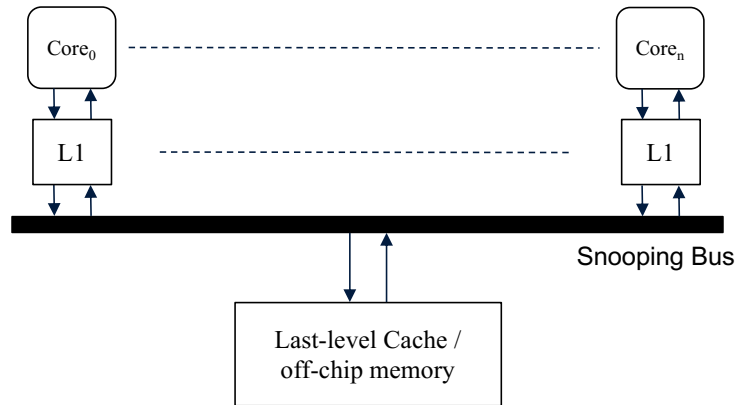
■ **Table 1** Summary of the properties of the related work along with PCC’s properties. \times indicates that the \times is not supported, while \checkmark indicates it is supported. features with several marks indicate a score for each work. For example, $\times\times\checkmark\checkmark$ indicates a score of 2/4, $\times\times\times\checkmark$ indicates 1/4, $\times\checkmark$ indicates a score of 1/2, and $\checkmark\checkmark$ indicates a score of 2/2.

Related work	Per-req. WCL	Bus Arch.	Supports OoO Exe.	Supports COTS Protocols	Supports different arbiters	Supports C2C Data Transfer	average performance
PMSI/ PMESI	Quadratic	Unified	$\times\times$	\times	\times	\times	$\times\times\checkmark\checkmark$
DISCO/ DISCO _{SharedW}	Linear	Unified	$\checkmark\checkmark$	\times	\checkmark	\times	$\times\checkmark\checkmark\checkmark$
PMSI* / PMESI*	Linear	Unified	$\times\times$	\times	\times	\checkmark	$\times\times\times\checkmark$
PISCOT	Linear	Split	$\times\checkmark$	\checkmark	\times	\checkmark	$\checkmark\checkmark\checkmark\checkmark$
PCC	Linear	Unified	$\checkmark\checkmark$	\checkmark	\checkmark	\checkmark	$\checkmark\checkmark\checkmark\checkmark$

requests from C_2 that come after ⑦. It also requires hardware support to enable selective bypassing of L1 caches for write requests. The author also proposed DISCO-SharedW in [14] as an enhanced version allowing the caching of modified cache lines in the L1 caches if they are to a private data that is not shared amongst cores. This optimization provides a good balance between a tight WCL and good performance. However, it assumes the availability of a prior information about the shared data. Accordingly, DISCO-SharedW scores a relatively high average-case performance in Table 1’s comparison.

The PMSI* and PMESI* solutions are proposed in [19] to be modified versions of the original PMSI/PMESI protocol. Similar to DISCO, they aim at reducing the quadratic WCL of PMSI by mitigating the impact of write-backs. Unlike DISCO, they limit the number of write-backs that have to go to the shared memory. This is achieved by deploying two techniques. 1) They enable direct communication between private caches, similar to this work. 2) They remove some of the standard transitions in the MSI and the $MESI$ protocols. In particular, if a core owns a line in the modified state, they disallow such line from being shared among several cores afterwards; i.e., such line can only be accessed by a single core at a time. The reason for this modification is to make this line directly transferable among cores while avoiding the need to transfer this line to the shared memory (until eviction). An example to this transition is C_1 ’s read request at ②, where C_1 receives B directly from C_2 but C_2 invalidates B after its transfer. While achieving a linear WCL in number of cores, this solution suffers from a significant performance penalty due to disabling simultaneous sharing of such lines and the possibly ping-pong effect as a result. The ping-pong effect appears between the read requests from C_2 and C_1 at ⑦ and ⑧; thus, PMSI* and PMESI* are given the worst performance score in Table 1.

PISCOT [17] follows a different approach by deploying a split-bus interconnect that implements a TDM request bus to broadcast coherence requests and a first-come first-serve (FCFS) response bus to transfer data responses. PISCOT is the only previous solution that enables the deployment of conventional coherence protocols without modifications. It also leverages the split-bus interconnect to achieve high average performance, while maintaining predictability with tight latency bounds. Another aspect of PISCOT compared to all existing works is its support to OoO cores by considering cores with multiple outstanding requests. Nonetheless, we find this support to be limited since it only services one request at a time; mainly to ensure the tight latency bounds. PCC similar to PISCOT enables the adoption of COTS coherence protocols in real-time systems without modifications. In doing so, unlike PISCOT, which requires a specific split-bus architecture, PCC enables the usage of legacy prevalent real-time bus architectures and arbiters. PCC also supports OoO cores and since it allows all requests to be non-preemptively serviced once their corresponding messages are broadcasted, it does not put any constraint on the OoO behavior of cores, while offering the tight WCL that is independent of the core pipeline architecture.



■ **Figure 3** The system model.

Table 1 summarize the differences between all the aforementioned approaches, and from that we can conclude the features that should be present in the proposed approach. Initially, the per-request WCL should be linear with the number of cores without compromising the average-case performance. Also, PCC should be independent of the coherence protocol therefore it can incorporate COTS protocols without any modifications. Moreover, it uses unified bus architecture with the ability of cache-to-cache data transfer to guarantee both tight WCL bounds and high performance. Finally, it should support OoO pipelines without changing the WCL, and it should not assume any prior information about shared data. All the previous point are also summarized in the last row of Table 1.

4 System Model

We consider a multi-core system that has N cores as shown in Figure 3.

Cores Architecture and Cache Hierarchy. Unlike most of the existing related work, we do not put a constraint on the pipeline architecture of the cores, they can be in-order or OoO or a mix of both. Each core has a private cache (L1). In addition, the system contains a shared memory, which it can be a last level cache, off-chip memory, or both of them. The writes from L1 to the shared memory is handled using write-allocate write-back policy, and the cache hierarchy is inclusive such that data existing in any L1 is a subset of the shared memory’s data. It is important to highlight that the proposed solution works for general cache architectures. For example, the solution works seemingly for different number of cache levels, where each core has several private caches and then the last private level of all cores connect to a shared cache. In that case, the solution works for the bus connecting to the shared cache. An exhaustive enumeration of all possible cache architecture is beyond the scope of this paper; thus, for conciseness, we focus on the model depicted in Figure 3. Another important notice is that we assume that the data will always be serviced within the depicted memory hierarchy in Figure 3 (i.e. the shared cache will always have the line in the valid state). This assumption is only to avoid the other sources of interference that are beyond the scope of this paper (e.g. I/O interference [6] or off-chip memory interference [10]).

Cache Coherence. The data is kept coherent among the private caches by incorporating any of the standard COTS coherence protocols. In this work, we exemplify by adopting three protocols: *MSI*, *MESI*, and *MOESI*.

Bus Architecture. L1 caches and the shared memory communicate through a *logical* shared snooping bus. We use the term logical here to differentiate between the logical bus model and the actual *physical* implementation details of this bus. As far as the paper is concerned, we make few assumptions about (requirements) the logical bus to be able to derive the bounds. Aside from these requirements, the *physical* implementation of the bus can be realized using any technique. These assumptions are as follows.

1) The bus allows the direct data transfers between L1 caches. Moreover, a transfer from a sender L1 cache to another can be overlapped with a transfer to the shared memory. In other words, a data sent from a core can be received by the shared memory and another core simultaneously. We find these assumptions to reflect techniques adopted in COTS architectures. For example, different existing ARM processors allow for direct transfers between private L1s, this includes processors both real-time (e.g. Cortex-R82 [4]) and application (e.g. Cortex-A53 [2]) families. Additionally, the data-coherent bus connecting private L1 caches and the snooping control unit (SCU) in Arm’s MPCore processor is separate from the bus connecting the SCU to the shared L2 cache, where the latter is AXI-based (e.g. in A9) or with the Cache Coherence Interconnect extensions to the AXI interface (CCI-400) such as in A15. On the other hand, for the CCI-550, the data interconnect is mentioned to be a fully connected cross bar [1]. Another vendor’s example is the QorIQ processor family from NXP, where the CoreNet Coherency Fabric enables point-to-point connections to pipeline the transfers between cores and shared memories [7]. Either having a dedicated bus or several parallel point-to-point connections will enable the required overlapped transfer.

2) The bus has a logical unified architecture. This means transferring a coherence message and a data message cannot be overlapped. During anytime instance, either a coherence message is being broadcast or a data message is being transferred. It is important to note that this assumption does not prevent the coherence and data messages from being sent on two different realized physical buses. It only requires unifying their arbitration such that their transfers are not overlapped. This is key to enable the integration of COTS coherence in predictable arbitration schemes with tight latency bounds, especially for OoO cores as we will discuss in section 6.

3) Cores are granted access to the bus using a predictable arbitration scheme. This work is not limited to a specific arbiter type, and we provide results, in Section 7, for TDM, RR, WRR, and HRR arbiters. Once a request is granted access to the bus by the arbiter, it will remain in service and no other request will be granted access until the in-service request is fulfilled. The maximum time to service any request is assumed to be L_{acc} . For slot-based arbitration schemes (such as TDM), the time slot of the bus arbiter should be long enough to fit the latency of broadcasting coherence messages besides transferring data (i.e. slot width should be at least L_{acc}).

The data sharing model. we don’t assume any constraints on data sharing or the shared address space. Additionally, our proposal can work with the timing interference management solutions like memory partitioning and coloring. Also, we don’t assume any restrictions on the task scheduler, so any task can run on any core without any implications on the system predictability.

5 Proposed Solution

PCC leverages the observations about the potential architectural features deployed by COTS platforms that are specified in the system model (Section 4) to facilitate the predictable integration of cache coherence in real-time systems without drastic degradation of performance.

17:10 Predictable Cache Coherence

In particular, the operation of PCC makes use of these two features: 1) direct cache-to-cache communication, and 2) a data transfer from a core can be simultaneously sent to another core as well as the shared memory. As a result, PCC operates according to Theorem 1.

► **Theorem 1.** *Once a request, $R_i(A)$ to any cache line A from any core C_i is granted access to the shared bus under PCC, it will be non-preemptively serviced without interference from any other request.*

Proof. Under the PCC system model described in Section 4, once a request from any core C_i is granted access to the shared bus, the actions conducted towards its fulfillment falls under one of the following three scenarios (depending on the type of the request as well as the specifications of the adopted coherence protocol).

1. **The requested cache line is already owned by C_i** In that case, to fulfil the request, it needs only to broadcast a coherence message. This will be the only action needed for requests that do not necessitate a data transfer. An example is the transition from O state to M state as a result of a write request under *MOESI* protocol.
2. **The requested cache line is owned by the shared memory.** In that case, C_i will also need to broadcast a coherence message and then receive the data from the shared memory. This will occur if the data is owned by the shared memory.
3. **The requested cache line is owned by another core, say C_j** In that case, C_i needs to broadcast a coherence message and then receive the data from C_j .

For Scenario 1, it is clear that once $R_i(A)$ is granted access to the bus using any predictable arbiter, it is an exclusive access, where it can issue its message non-preemptively; and hence, finishes its service.

For Scenario 2, after $R_i(A)$ gains access to the bus and broadcast its message (similar to scenario 1), it also requires to receive the requested data from the shared memory. According to the system model in Section 4, this also happens directly once $R_i(A)$'s message is broadcasted without any interruption from other requests.

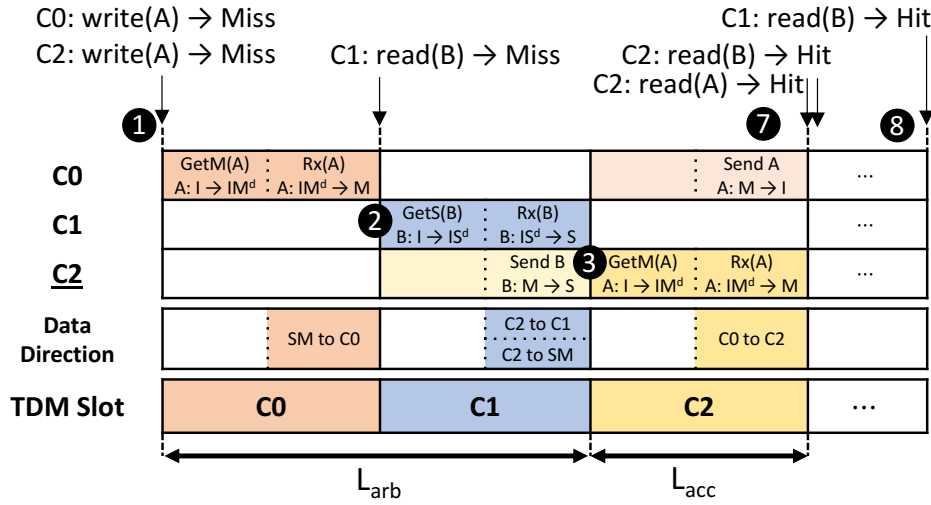
For Scenario 3, there are two sub-scenarios as follows. Scenario 3(a) is the case where the owner core C_j needs to send the data to C_i only and not the shared memory. This is the case for example if $R_i(A)$ is a *getM(A)* request, while A is modified in C_j 's private cache. This scenario is similar to Scenario 2 since it requires a single data transfer, while the source of the data is now a core and not the shared memory. Therefore, same argument applies similar to Scenario 2 such that $R_i(A)$ finishes without any interruption.

Scenario 3(b) is a bit more involved as it requires two data transfers and not one. In this scenario, C_j needs to send the data to both C_i and the shared memory. This occurs for instance if $R_i(A)$ is a *getS(A)* request, while A is modified in C_j 's private cache. Leveraging the observation about the bus architecture in Section 4, PCC is able to conduct these two transfers in parallel; and hence, enables $R_i(A)$ to also finish in this case without any interruption. ◀

5.1 Illustrative Example

Now, we show the operation of PCC in action by applying it to the same example in Figure 2 and discuss how PCC offers predictability with tight bounds and no required modifications to the coherence protocol.

Figure 4 shows the latencies of PCC in a system that utilizes TDM arbiter and *MSZ* coherence protocol. At ❶, C_0 and C_2 have write requests to A , and each request is broadcast, afterwards, to the bus at the beginning of its core's TDM slot. At ❸, C_2 , the core under analysis, is granted access to the bus and it issues its request. During the same slot of C_2 ,



■ **Figure 4** The memory latency of a write request to A from core C₂ in a 3-core system that uses PCC along with TDM arbiter and MSZ coherence protocol. This figure follows the same scenario, initial conditions, and the numbering as in Figure 2.

C₀, the previous owner of A, responds with the data. Accordingly, this example contains only two type of latencies L_{arb} and L_{acc} , which they are the only types that a single request can incur in case of PCC. Unlike PMSI, PCC does not suffer from $L_{inter-coh}$ or $L_{intra-coh}$ because of the fact that requests are served in a single access slot to the bus (Theorem 1). Additionally, a core cannot have a pending write-back and a request during its access slot because the write-backs are handled during the requestor’s access slot, and core’s access slots are for its demanding requests only.

PCC leverages COTS coherence protocols which ensures high average-case performance. This can be observed from the example in two instances. First at ②, where C₁ requests reading B which is owned by C₂. Consequently, C₂ responds by sending B to C₁ as well as to the shared memory; hence, both C₁ and C₂ are allowed to keep B in S state, unlike PMSI*. The second instance at ③, after C₂ completes its write to A, PCC allows C₂ to keep A in M state, unlike DISCO. This results in access hits for the read request to A at ⑦ and the read requests to B at ⑦ and ⑧.

6 Timing Analysis and Predictability Guarantees

In this section, we show that PCC solution satisfies the predictability invariants proposed in [15]. Besides, we derive PCC’s per-request WCL and the total worst-case memory latency incurred by a task.

6.1 Satisfying Predictability Invariants

According to [15], a system that utilizes COTS coherence protocol along with a predictable arbiter cannot guarantee a predictable memory latency. Hassan et al., also, provided in [15], 6 invariants to test the predictability of cache memory systems. Accordingly, we show in this section that PCC satisfies all the invariants, which means COTS protocols with any predictable arbiter can be predictable if cache-to-cache data transfers are allowed according to the described system model.

17:12 Predictable Cache Coherence

Invariant 1. A predictable bus arbiter must manage coherence messages on the bus such that each core may issue a coherence request on the bus if and only if it is granted an access slot to the bus.

► **Lemma 2.** *PCC satisfies Invariant 1.*

Proof. Allowing a core to issue a request on the bus is the bus arbiter's responsibility, and according to the system model defined in Section 4, PCC utilizes predictable arbiters. Predictable arbiters allow cores to issue requests only in their own dedicated access slots. ◀

Invariant 2. The shared memory services requests to the same line in the order of their arrival to the shared memory.

► **Lemma 3.** *PCC satisfies Invariant 2.*

Proof. Let two requests, $R_i(A)$ and $R_j(A)$ from two different cores C_i and C_j , respectively. Both requests target the same cache line which is owned by the shared memory. If $R_i(A)$ and $R_j(A)$ appear on the bus at t_i and t_j , respectively, where $t_i < t_j$, then according to Theorem 1, $R_i(A)$ once its message is broadcast at t_i , it will not be interrupted by any other request including $R_j(A)$ until it is serviced, which implies here that the shared memory finishing sending the data to Core C_i . Therefore the shared memory services the request according to the order of their appearance on the bus which is the same order of their arrival to the shared memory. ◀

Invariant 3. A core responds to coherence requests in the order of their arrival to that core.

► **Lemma 4.** *PCC satisfies Invariant 3.*

Proof. Let two requests, $R_i(A)$ and $R_j(B)$ from two different cores C_i and C_j , respectively. The requests target different cache lines which are both owned by C_k . Similar to the shared memory, if R_i appears first on the bus, C_k has to respond with the data to C_i before any other request can broadcast its message (Theorem 1). In conclusion, cores respond immediately to requests once they appear on the bus, therefore the requests' arrival order is respected. ◀

Invariant 4. A write request from C_i that is a hit to a non-modified line in C_i 's private cache has to wait for the arbiter to grant C_i an access to the bus.

► **Lemma 5.** *PCC satisfies Invariant 4.*

Proof. The coherence protocols dictate how the cache controllers deal with the writes to the non-modified lines. According to the system model in Section 4, PCC incorporates COTS protocols that necessitate a modification request (**GetM** or **UpgM**) to be broadcast before writing to a non-modified line (i.e., a line in S state). (1)

According to Lemma 2, cores issue requests only during their access slot. (2)

From (1) and (2), a write request to a non-modified line in the core's private cache should wait for the core's access slot to the bus. ◀

Invariant 5. A write request from C_i that is a hit to a non-modified line, say A , in C_i 's private cache has to wait until all waiting cores that previously requested A get an access to A .

► **Lemma 6.** *PCC satisfies Invariant 5.*

Proof. Assume that C_i owns non-modified cache line A in its private cache, and at time t_i , it has a write request, $R_i(A)$, to A . In addition, assume that C_j requested access (read or write), and its request $R_j(A)$, to A is broadcast on the bus at time t_j , where $t_j < t_i$. Invariant 5 breaks if PCC allows serving $R_i(A)$ before $R_j(A)$. (1)

Whereas, Lemma 5 enforces C_i to wait until it is granted access to the bus before proceeding with $R_i(A)$. Let this to happen at time $t_{i+\delta}$, where by construction $t_{i+\delta} > t_i$ (2)

Finally, Theorem 1 dictates that once a request is broadcast, it will be serviced before any other request can be broadcast or serviced (3)

From (1) – (3) and since $t_j < t_{i+\delta}$, it necessitates that $R_j(A)$ will be serviced before $R_i(A)$; thus, PCC satisfies Invariant 5. ◀

Invariant 6. Each core has to deploy a predictable arbitration between its own generated requests and its responses to requests from other cores.

► **Lemma 7.** *PCC satisfies Invariant 6.*

Proof. From Theorem 1, cores will never have pending responses to others' requests during their own access time. The reason for that is because of the principle of non-preemptively servicing the requests once they are granted access to the bus. Accordingly, the arbitration between a core's own generated requests and its responses always chooses the requests due to the absence of pending responses. Therefore, PCC has the effect of this arbitration layer intrinsically without implementing it. ◀

6.2 Per-Request WCL Analysis

According to the example in Section 5, it is clear that the latency of a single request depends mainly on the type of the bus arbiter.

► **Lemma 8.** *The per-request latency is calculated as in Equation 1, where WCL_{arb} is the worst-case arbitration latency.*

$$WCL_{perReq} = WCL_{arb} + L_{acc} \quad (1)$$

Proof. In worst-case, such request has to wait for WCL_{arb} before it can be granted access to the bus by the arbiter. Once it is granted access to the bus, according to Theorem 1, a core's request is non-preemptively serviced. Based on the system model, this service consumes a maximum latency of L_{acc} . Therefore, the per-request worst-case latency is as depicted in Equation 1. ◀

► **Lemma 9.** *The processing latency of any request under PCC is calculated as in Equation 1 regardless of the pipeline architecture of the cores in the system (whether in-order, out-of-order or a combination of both).*

Proof. For an in-order core, the proof is straightforward. An in-order core can have a maximum of one request in-flight at any given time. Therefore, such request do not suffer any queuing delay from requests of the same core (it is always the head of the queue). Such request suffers a worst-case processing latency as proven by Lemma 8. Additionally, such core causes a maximum interference of one request on other cores since it cannot have several requests simultaneously requiring service per construction.

An out-of-order core, in contrast, can have several simultaneously outstanding requests. We now prove that this behavior does not impact the processing latency of requests from such core, nor impacts other cores. First, for requests from the core itself, we are bounding the processing latency of any request, which is the latency suffered by such request once it becomes at the head of the queue of its corresponding core. This is because this processing latency is the considered one to be used when calculating the overall's task WCET []. As a result, no queuing delay needs to be added as a component to the latency in Lemma 8. Second, for the interference impact from this core on other cores, we prove that it still adheres to Lemma 8 as follows. 1) Under any of the predictable arbiters considered in the system model, each core gets a guaranteed turn to access the bus regardless of the behavior of other cores or the number of their outstanding requests. As a result, the arbitration latency component, WCL_{arb} remains the same. 2) By construction of PCC and as proven by Theorem 1, once a request is granted access to the bus, it entertains a non-preemptive service until it is fulfilled. This is regardless of the behavior of other cores. As a result, the access latency component, L_{acc} remains the same. From 1) and 2), we finish the proof for the out-of-order core case. ◀

6.3 Total Task's Worst-Case Memory Latency Analysis

A task's WCET can be computed as follows: $WCET = WCCT + WCML$, where the $WCCT$ is the worst-case computation time of the task executing on the core, and the $WCML$ is the total worst-case memory latency suffered by the task upon accessing the memory. We now show how to compute the $WCML$ using the per-request WCL derived in Section 6.2.

WCL can be simply calculated using the per-request WCL calculated in Equation 2 as follows, where R_T is the total number of memory request issued by the task under analysis.

$$WCML = R_T \times WCL_{perReq} \quad (2)$$

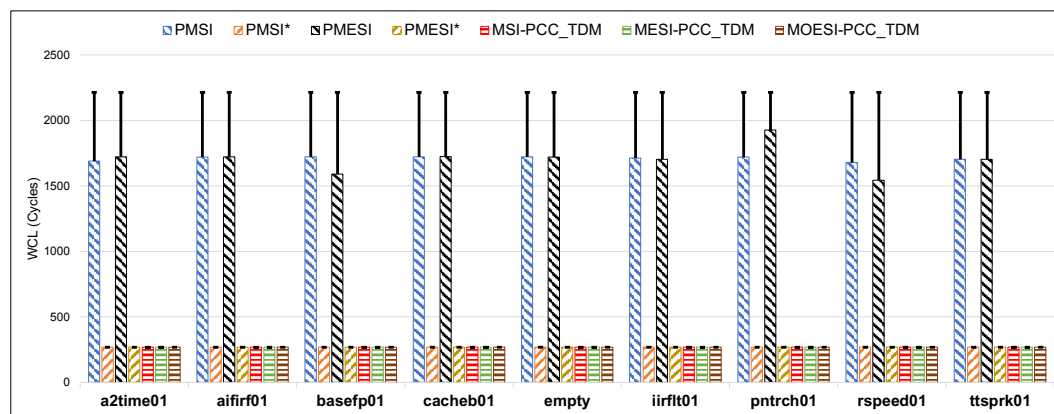
Effect of Dirty Line Replacements. Another latency component that should be considered is the effect of the write back requests due to L1 cache evictions or replacement. In worst-case, every request can trigger an eviction of a dirty cache line; and hence, creates a write back request that it has to wait for. As a result, The WCML in Equation 2 changes to the value calculated in Equation 3. This is because every request now has to wait for an additional write back request that also susceptible to WCL_{perReq} latency in worst case.

$$WCML = 2 \cdot R_T \times WCL_{perReq} \quad (3)$$

In some systems, the number of dirty line evictions can be constrained to the number of write requests. Accordingly, calculating WCML can be carried out using Equation 4, where R_W is the number of write requests.

$$WCML = R_T \times WCL_{perReq} + R_W \times WCL_{perReq} \quad (4)$$

This is the case for instance for the *MSI* protocol. However, COTS protocols that implement E state (e.g. *MESI* or *MOESI*) require PutM message for the lines in the E state, which are clean (non-modified) lines. As a result, using Equation 4 with such protocols is not sufficient and can lead to unsafe bounds. In that case, using a more conservative bound such as the one in Equation 3 is the safe decision.



■ **Figure 5** Per-request WCL of running EEMBC benchmarks, where T-bars represent the analytical value and solid bars are for the observed WCL among all the requests.

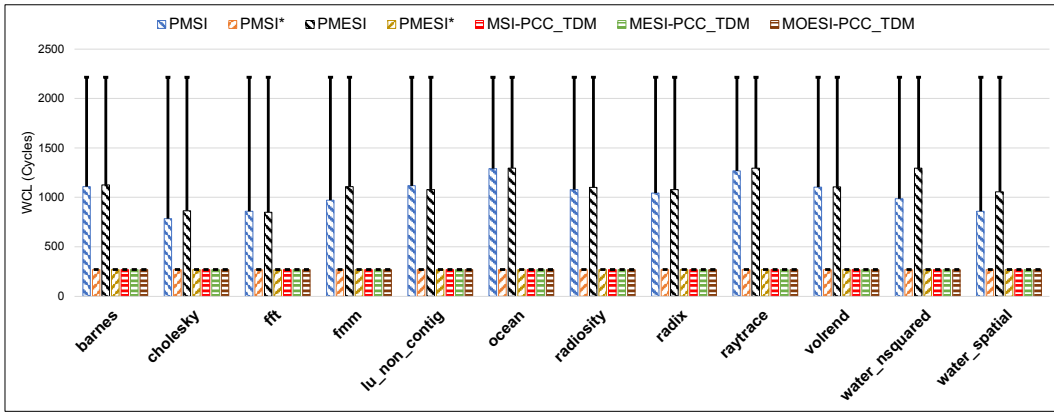
Effect of Hit/Miss Cache Analysis in the Presence of Coherence. Equation 3 is safe and sufficient to be used in the case of lack of more available information from the task analysis; however, WCML can be tightened using information from running the task under analysis in isolation (i.e., run the task on a core while turning off the other cores). For instance, the assumption that all demanding requests from a task incur WCL_{perReq} is rather pessimistic, since in most typical cases number of L1 hits is higher than the misses, and the hit latency (L_{Hit}) is much smaller than the WCL_{perReq} . However, it is not feasible to infer the number of hits from the isolation analysis due to the absence of coherence interference which decreases the hit rate. Moreover, the effect of coherence interference is not limited only to the shared data, but it can affect the hit rate of private data as well unless the target system offers data isolation between private and shared data in L1. Therefore, systems without data isolation should adhere to $WCML$ calculated by Equation 3. Whereas, if the system is capable of isolating shared data from private data (for example by providing a separate cache partition for each, $WCML$ can be more tightened by Equation 5. Equation 5 is based on the fact that if private and shared data are isolated from each other, they will not be able to evict each other, and an isolation analysis is performed to calculate the number of requests for private data that hit in L1 ($R_{privHit}^{iso}$), the number requests for private data that miss in L1 ($R_{privMiss}^{iso}$), the number of write-backs due to replacements to private data (R_{Repl}^{iso}), and the number requests for shared data (R_{Shared}). Since no assumption can be made about requests to shared data, all of them are assumed to be misses, and further suffering from write-back replacement delays.

$$WCML = (R_{privHit}^{iso} \times L_{Hit} + (R_{privMiss}^{iso} + R_{Repl}^{iso}) \times WCL_{perReq}) + (2 \times R_{Shared} \times WCL_{perReq}) \quad (5)$$

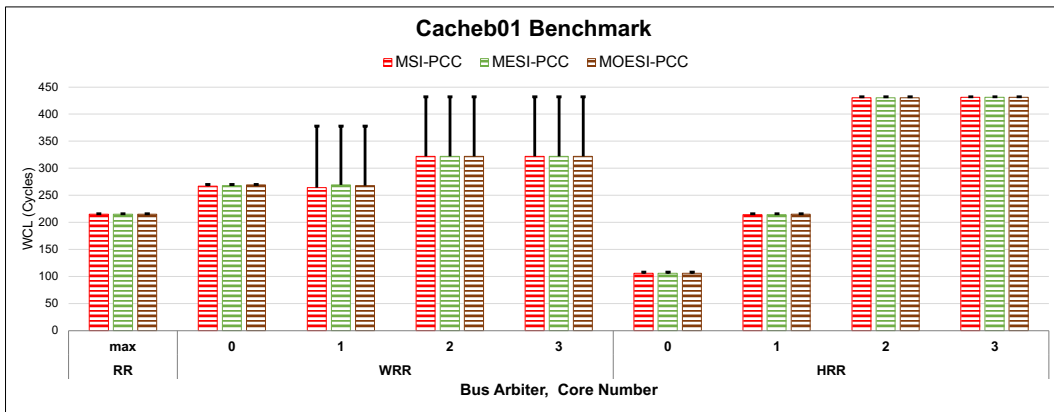
7 Evaluation

In order to evaluate our proposed solution, we performed a number of experiments to compare between PCC and state-of-the-art solutions. Additionally, we conducted other experiments to explore the effects of different combinations of bus arbiters with coherence protocols along with different execution modes (in-order and OoO). Throughout the experiments, we used a

17:16 Predictable Cache Coherence



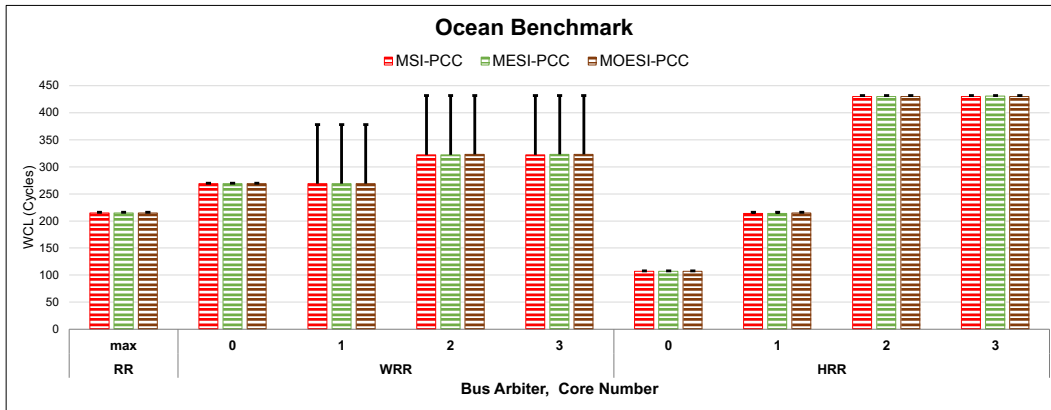
■ **Figure 6** Per-request WCL of running SPLASH-3 benchmarks, where T-bars represent the analytical value and solid bars are for the observed WCL among all the requests.



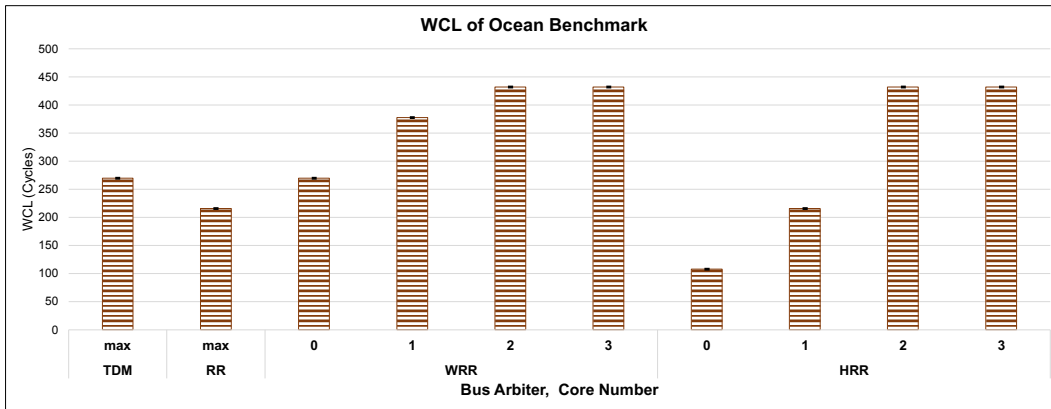
■ **Figure 7** The observed WCL (solid bars) and the analytical WCL (T-bars) for Cacheb01 benchmark from EEMBC suite.

4-core system where each core has 16KB direct-mapped L1 cache, and the cores are connected to each other and to a last level cache by a snooping unified bus. The access latency of L1 (L_{Hit}) is 1 cycle, and the complete access slot to the bus is 54 cycles: 4 cycles for the request latency and 50 for the data transfer. To avoid the unnecessary large latency of the off-chip memory, we used a perfect last level cache which fits the whole data of the running application. All the experiments were carried out using an open-source cache simulator¹, which enables us to run trace-based simulations on memory traces collected using Intel's PINtool upon executing the benchmarks. We used SPLASH-3 benchmarks [26], which were configured to execute in 4 threads where each thread runs on a separate core. Moreover, we used benchmarks from the EEMBC [25] suite to simulate the extreme case of data sharing and coherence interference by running four instances of the same benchmark trace on the four cores, simultaneously. Accordingly, all the cores issue almost the same sequence of requests and share 100% of their data.

¹ <https://gitlab.com/FanosLab/pcc-sim>



■ **Figure 8** The observed WCL (solid bars) and the analytical WCL (T-bars) for Ocean benchmark from SPLASH-3 suite.



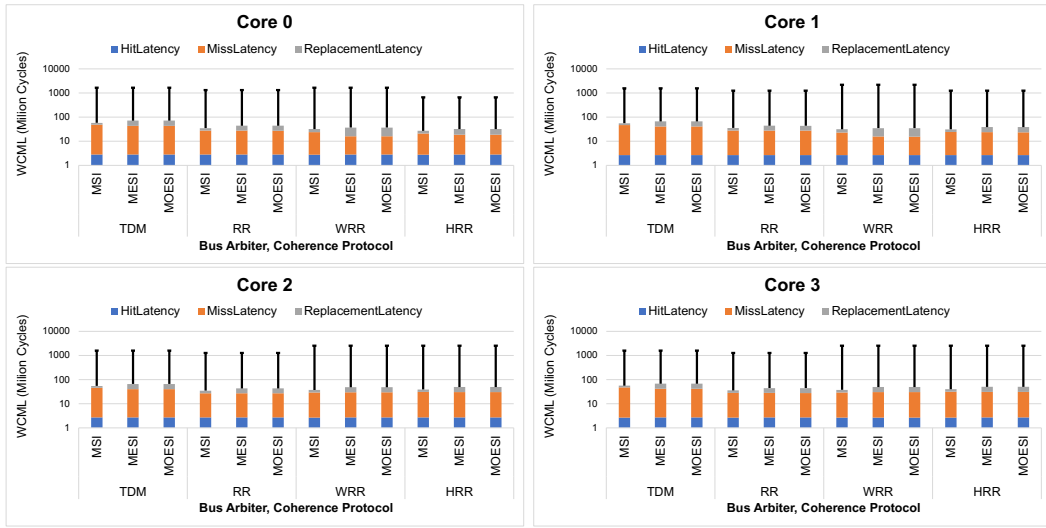
■ **Figure 9** The observed WCL (solid bars) and the analytical WCL (T-bars) for Ocean benchmark from SPLASH-3 suite running in a 4-core system with OoO pipeline and *MOESI* coherence protocol.

7.1 Per-Request WCL

In this group of experiments, we measure the latency that is incurred by each memory request of the running benchmark, and compare it to the analytical WCL to ensure the safety and the tightness of the bounds. Figure 5 shows the results of running EEMBC benchmarks, and it compares between PMSI [15], PMESI [18], PMSI*, and PMESI*[19]. Besides, the figure includes three variants of PCC with the protocols *MSI*, *MESI*, and *MOESI* along with TDM bus arbiter. TDM is chosen for this experiment to have a fair comparison with the other solutions which adopt a TDM arbiter. Similarly, Figure 6 shows WCL of running the SPLASH benchmarks.

► **Observation 1.** *The observed WCLs of the solutions that implement cache-to-cache data transfer are much tighter compared to PMSI and PMESI. Further, the analytical bounds of PMSI and PMESI are quite pessimistic, which is clear from Figure 6 where hardly 50% of the bound is reached. Contrarily, the other solutions, including PCC, show notably tight bounds with both benchmarks suites.*

Figures 7 and 8 delineate WCL of chosen benchmarks from EEMBC and SPLASH, respectively. The results of the other benchmarks are consistent with Figures 7 and 8; hence, we include the results of a single benchmark from each suite for conciseness. In this experiment, a



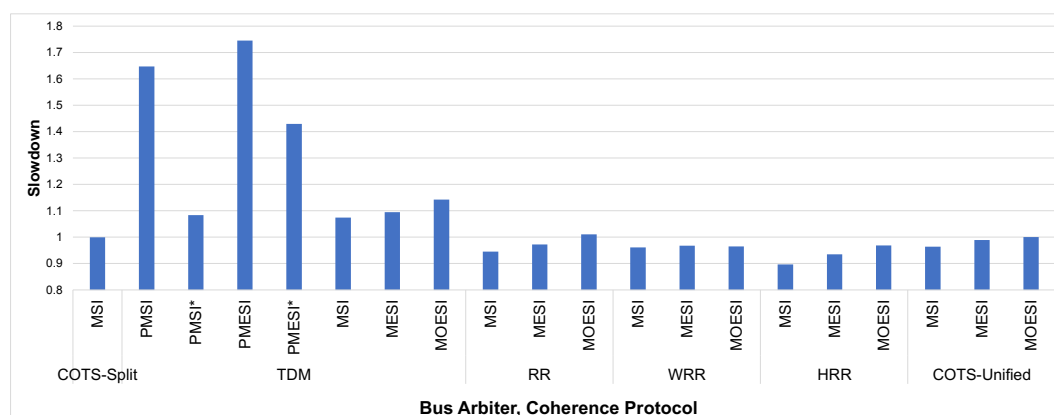
■ **Figure 10** The total memory latency of running Ocean benchmark from SPLASH-3 suite. The T-bars represent the analytical WCML calculated using Equation 3 and the solid bars are for the observed total memory latency. The vertical axis is in a logarithmic scale.

comparison is held between a combination of configurations that contain MSI , $MESI$, and $MOESI$ protocols along with RR, WRR, and HRR arbiters. WCLs of WRR and HRR are reported per core due to their unfair arbitration scheme; therefore, each core has a different bound based on its weight. The weights of the cores, in this and the following experiments, are $\{4, 2, 1, 1\}$, and the numbers are chosen arbitrarily to show the effect of the different weights. On the other hand, RR arbiter treats all the cores similarly; thus, the maximum value of WCL among the cores is reported.

► **Observation 2.** *There is no noticeable difference between results of Figures 7 and 8, although EEMBC simulates a synthetic effect of all the data is shared. This indicates the tightness and safety of WCL bounds in the case of a synthetic benchmark, like in Figure 7, and in a normal case, like Figure 8. Regarding the arbiters, RR shows a better WCL compared to WRR, whereas its WCL is average compared to the smallest and largest WCLs of HRR. The effect of the weights of the cores in WRR is not highly reflected into the WCLs, and the analytical WCL appears to be loose. On the other side, WCLs of the cores in HRR are rather tight.*

To complete the study of WCL, the analytical bounds are tested in the case of OoO execution. Figure 9 shows WCL of running a chosen benchmark from SPLASH-3 suite (the same benchmark used in Figure 8.) The execution of the benchmark is held using cores with OoO pipeline with a maximum of 8 outstanding requests. Additionally, TDM, RR, WRR, and HRR are used along with $MOESI$ coherence protocol for the comparison. $MOESI$ is chosen in this set of experiments since it is considered the most advance protocol among MSI and $MESI$.

► **Observation 3.** *The analytical bounds are still respected in the OoO execution; however, WRR shows tighter WCLs compared to the in-order experiment shown in Figure 8. The tightness of the WCLs is due to the ability of the OoO cores to issue multiple requests and benefit from their high weights; therefore, the cores with lower weights are pushed to their bounds.*



■ **Figure 11** The average slowdown of the performance of the predictable cache solutions compared to COTS-Split and COTS-Unified. All the values of running EEMBC benchmarks are normalized to COTS-Split performance.

7.2 Total Task's WCL

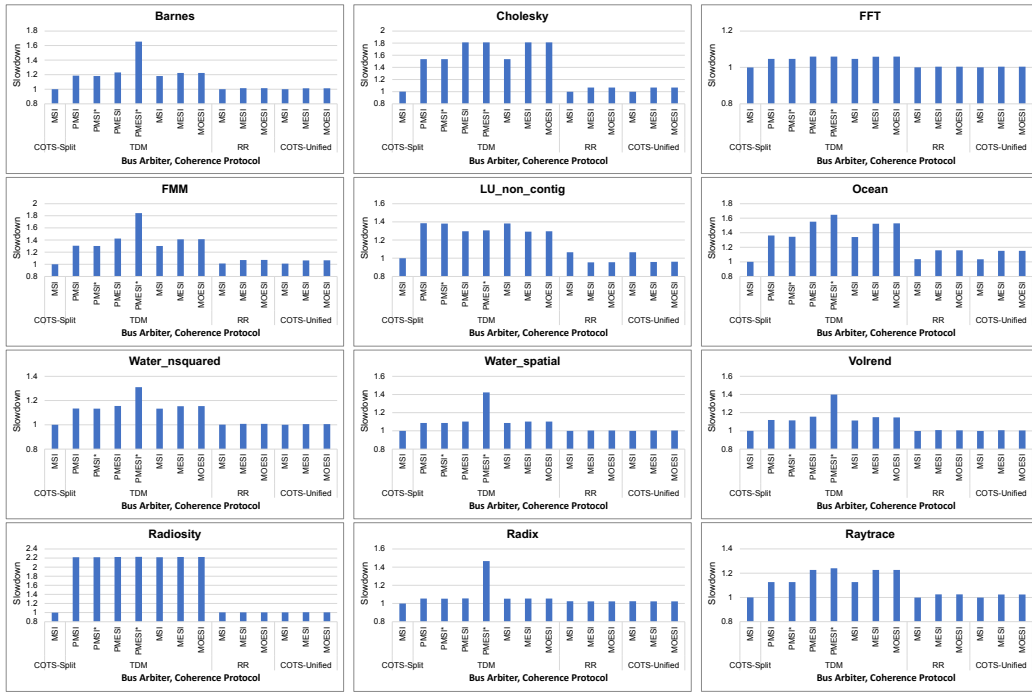
In this experiment, we are concerned with the total memory latency that is incurred by each task. Similar to the previous experiment, a single benchmark from SPLASH-3 is run using MSI , $MESI$, and $MOESI$ protocols along with TDM, RR, WRR, and HRR bus arbiters. Figure 10 shows WCML and the memory latency per each core. The latency is broken down into the three contributors to the latency: hit, miss, and replacement latencies. No information is assumed about the private vs shared data; and hence, WCML is calculated using the most conservative equation (Equation 3). Results are reported per core as each thread of the benchmark is mapped to a single core.

► **Observation 4.** *The gap between WCML and the observed latency is very large because of the pessimism in Equation 3. Regarding the protocols, MSI shows better latencies compared to $MESI$ and $MOESI$, and by looking at the replacement latency part, it is clear that replacements are the reason for the performance degradation of $MESI$ and $MOESI$. The justification of the higher number of replacements in $MESI$ and $MOESI$ can be returned to the addition of E and O states as they require a $PutM$ message, unlike S state, before the eviction of a cache line. An optimization that can mitigate the latency of replacements is the empty $PutM$ message (a $PutM$ message that is not followed by a write-back) in the case of E state; however, this optimization has to be coupled with a dynamic arbiter, such as RR, to show an impact on the latency.*

7.3 Average-case Performance

In this section, we compare the average-case performance between different solutions including the solutions that implement COTS arbiters which favor performance only. Figure 11 delineates the average slowdown of each solution relative to a system that deploys an MSI protocol along with a split bus architecture and FCFS arbiter which is denoted by COTS-Split. This experiment runs the synthetic benchmarks of EEMBC to compare the performance of all the previous configurations and adding to them COTS-Unified configurations. COTS-Unified denotes a FCFS bus arbiter on top of a unified bus architecture. Similarly, Figure 12 shows the performance results of running SPLASH-3 benchmarks.

17:20 Predictable Cache Coherence

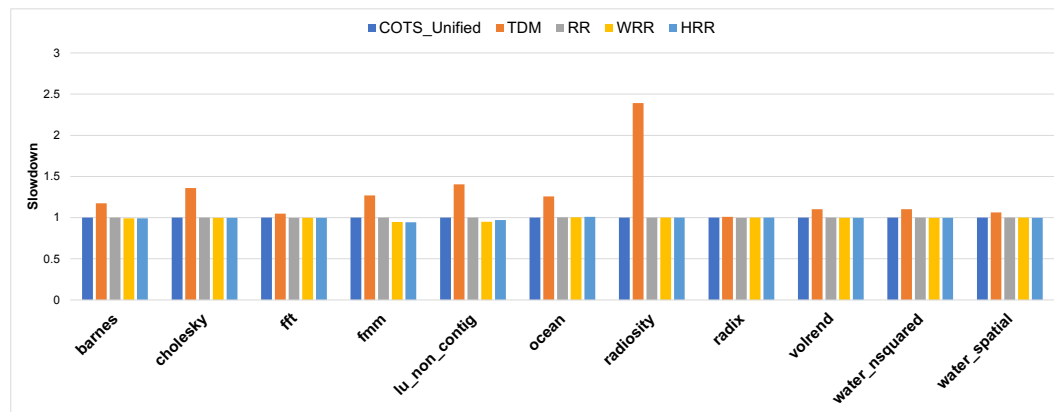


■ **Figure 12** The slowdown of the performance of the predictable cache solutions compared to COTS-Split and COTS-Unified. All the values of running SPLASH-3 benchmarks are normalized to COTS-Split performance.

► **Observation 5.** *The relative performance of MSI protocol compared to MESI and MOESI, shown in Figures 11 and 12, is consistent with their total memory latency which is discussed Observation 4. PMSI in Figure 11 shows a large performance degradation compared to MSI with TDM arbiter, while in Figure 12 the gap between them is minimal. The reason for this is the nature of the benchmarks as the percentage of the shared data in SPLASH is smaller than that in EEMBC; EEMBC has 100% of data shared to impose high coherence interference between cores. Also, we can make the same observation for PMESI and MESI with TDM. Moreover, PMESI* shows the worst performance in most of the benchmarks in Figure 12, and this is because PMESI* limits the cases that different cores can have the same cache line in S state. Finally, PCC deploying MSI with RR arbiter shows the best performance in Figure 12 and it is the closest to COTS-Split performance.*

The last experiment is to compare the performance of different arbiters that are coupled with MOESI protocol and OoO cores. Figure 13 shows the results of executing SPLASH-3 benchmarks, where the performance values of the arbiters are normalized to the values of COTS-Unified. The other predictable solutions are excluded from this experiments as none of them supports OoO execution.

► **Observation 6.** *The average gain in performance of MOESI with RR OoO execution compared to in-order execution is 5%. TDM shows the worst performance among the other arbiters, while WRR shows a slightly better performance than RR and HRR.*



■ **Figure 13** The slowdown of the performance of MOESI-PCC compared to COTS-Unified. All the values of running SPLASH-3 benchmarks are normalized to COTS-Unified performance.

8 Conclusion

We propose PCC: a solution to integrate COTS coherence protocols into legacy predictable real-time arbitration schemes without requiring any modifications to either of them. Doing so has several benefits. 1) PCC achieves tight latency bounds with minimal performance degradation. 2) It does not impose any burden on designing or verifying new protocols, which facilitates adoption by industry. 3) It uses legacy arbitration schemes that have been studied for a long time, which in turn carries forward the credit of their analyzability making the proposed solution more appealing from a certification perspective. 4) It enables the integration of any coherence protocol with any predictable arbiter in a plug-and-play fashion. In this paper, we leveraged this capability to implement 3 different detailed coherence protocols as well as 4 commonly-used real-time arbiters. This allowed us to carry exploratory experiments for 12 different architectural configurations. Finally, we release the source code of the cycle-accurate implementation of such architectures for the community to use and expand.

References

- 1 ARM. ARM CoreLink CCI-550 Cache Coherent Interconnect, Technical Reference Manual, 2015. URL: https://static.docs.arm.com/100282/0001/corelink_cci550_cache_coherent_interconnect_technical_reference_manual_100282_0001_01_en.pdf.
- 2 ARM. Arm Cortex-A53 MPCore Processor Technical Reference Manual r0p4, 2016. URL: <https://developer.arm.com/documentation/ddi0500/j/Level-2-Memory-System/Snoop-Control-Unit>.
- 3 ARM. Arm Cortex-A9 MPCore Technical Reference Manual r4p1, 2016. URL: <https://developer.arm.com/documentation/100486/0401/snoop-control-unit>.
- 4 ARM. ARM Cortex-R82, Technical Reference Manual, 2021. URL: <https://developer.arm.com/documentation/101548/0002/?lang=en>.
- 5 Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Cache where you want! reconciling predictability and coherent caching. *arXiv preprint*, 2019. arXiv:1909.05349.
- 6 Daniel Casini, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Latency analysis of i/o virtualization techniques in hypervisor-based real-time systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 306–319. IEEE, 2021.

- 7 Chun Chang. A Deep Dive on the QorIQ T2080 Processor, NXP, 2014.
- 8 M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2016.
- 9 David Kruckemyer Craig Forrest. Arteris Ncore™ Cache Coherent Interconnect, Technology Overview, 2006.
- 10 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–23, 2018.
- 11 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- 12 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77. IEEE, 2009.
- 13 Mohamed Hassan. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 495–505. IEEE, 2018.
- 14 Mohamed Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 15 Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246. IEEE, 2017.
- 16 Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018.
- 17 Salah Hessian and Mohamed Hassan. The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–230. IEEE, 2020.
- 18 Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 2020.
- 19 Anirudh Mohan Kaushik and Hiren Patel. A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–117. IEEE, 2021.
- 20 Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432. IEEE, 2019.
- 21 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared data caches conflicts reduction for wcet computation in multi-core architectures. In *18th International Conference on Real-Time and Network Systems*, page 2283, 2010.
- 22 MILO MK MARTIN, MARK D HILL, and DANIEL J SORIN. Why on-chip cache coherence is here to stay. *Communications of ACM*, 2012.
- 23 Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.
- 24 Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

- 25 J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, September 2009. doi:10.1109/MM.2009.74.
- 26 Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.
- 27 Nivedita Sritharan, Anirudh Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445. IEEE, 2019.
- 28 Mohamed Younis and Mohamed Aboutabl. Communication handling in integrated modular avionics, October 3 2002. US Patent App. 09/821,601.

RT-DFI: Optimizing Data-Flow Integrity for Real-Time Systems

Nicolas Bellec  



Univ Rennes, Inria, CNRS, IRISA, France

Guillaume Hiet  

CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, France

Simon Rokicki  

Univ Rennes, Inria, CNRS, IRISA, France

Frederic Tronel  

CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, France

Isabelle Puaut  

Univ Rennes, Inria, CNRS, IRISA, France

Abstract

The emergence of Real-Time Systems with increased connections to their environment has led to a greater demand in security for these systems. Memory corruption attacks, which modify the memory to trigger unexpected executions, are a significant threat against applications written in low-level languages. Data-Flow Integrity (DFI) is a protection that verifies that only a trusted source has written any loaded data. The overhead of such a security mechanism remains a major issue that limits its adoption. This article presents RT-DFI, a new approach that optimizes Data-Flow Integrity to reduce its overhead on the Worst-Case Execution Time. We model the number and order of the checks and use an Integer Linear Programming solver to optimize the protection on the Worst-Case Execution Path. Our approach protects the program against many memory-corruption attacks, including Return-Oriented Programming and Data-Only attacks. Moreover, our experimental results show that our optimization reduces the overhead by 7% on average compared to a state-of-the-art implementation.

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

Keywords and phrases Real-time system, Software security, Data-flow integrity, Worst-case execution time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.18

Supplementary Material *Software*: <https://gitlab.inria.fr/nbellec1/rt-dfi>

Acknowledgements We want to warmly thank AbsInt for providing the aiT WCET estimator.

1 Introduction

Real-time system (RTS) security has emerged as a growing concern with the increasing development of connected systems such as autonomous vehicles and the Internet-of-things [33, 23]. With many real-time systems still written in memory unsafe languages such as C and C++, the threat of memory corruption bugs remains important [37]. Previous research proved that such vulnerabilities have been used to attack these systems [31, 18].

Protections for RTS have to consider the *Worst-Case Execution Time (WCET)* in their design. In particular, the overhead of the protection on the WCET must be predictable to ensure that the estimation of the WCET remains a safe upper-bound of any execution time.

Previous works on protecting RTS programs against memory corruption have explored the adaptation of *Control-Flow Integrity (CFI)* to real-time constraints [32]. CFI ensures that the protected program execution conforms to the statically computed program's *Control-Flow*



© Nicolas Bellec, Guillaume Hiet, Simon Rokicki, Frederic Tronel, and Isabelle Puaut; licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 18; pp. 18:1–18:24



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Graph. At each branch executed by the program, CFI verifies that the branch’s target belongs to a set of valid targets. The protection considers that any branch to an invalid target corresponds to an attack and responds accordingly (e.g., by stopping the execution). However, CFI cannot protect against advanced attacks that corrupt non-control data. To protect against such threats, we have to rely on protections, such as *Data-Flow Integrity (DFI)* [10, 15], which guards the program against a broader spectrum of attacks.

DFI protects the program by ensuring that every data loaded from memory has been written by a trusted source. Software implementations of this protection are deterministic for a WCET estimator since the whole protection resides in the program’s instructions. However, DFI can have a significant time overhead on the protected program, despite the optimizations that have already been proposed [10]. Furthermore, these optimizations have not been designed for RTS and are unaware of the WCET.

In this paper, we present RT-DFI, a new software implementation of DFI that aims at improving the WCET specifically, in contrast to state-of-the-art DFI protections. Our approach uses *Integer Linear Programming (ILP)* to reduce the overhead on the estimated *Worst-Case-Execution Path (WCEP)*. Since our objective is to reduce the overhead on the WCET, we focused our work on a single task in the system. We evaluated RT-DFI using aiT [21], the industry standard for static timing analysis. The results show that our approach can reduce the overhead of DFI by up to 18% on the estimated WCET. The contributions of this paper are the following:

- We present RT-DFI, a new method to optimize software DFI for real-time systems. Compared to the state-of-the-art, RT-DFI reduces the DFI overhead on the estimated WCET.
- We implemented RT-DFI and a state-of-the-art DFI protection for a RISC-V processor within the LLVM compilation toolchain.
- We evaluated our method on various tasks of the TACLeBench benchmark suite [17] and showed that we obtained WCET improvements over the state-of-the-art technique, of 7% on average.

The rest of this paper is organized as follows. Section 2 presents some background on real-time systems and DFI. Then, we detail our contribution in Section 3. Section 4 formalizes the constructed ILP used to optimize DFI. Section 5 contains all information on how we conducted our experiments and presents experimental results. Section 6 presents the related work. Finally, Section 7 concludes this paper and proposes some perspectives.

2 Background

Memory corruption attacks aim at modifying the memory of a program to break its security properties, i.e., its confidentiality, integrity, or availability. Nowadays, many attacks have been developed to exploit potential memory corruptions in a program, such as *Return-Oriented Programming (ROP)* [9], *Control-Flow bending* [16], or *Data-Flow bending* [30]. With real-time systems being more connected than ever, malicious actors can exploit vulnerabilities in real-time systems to modify their behavior and break the guarantees of these systems, potentially resulting in significant economic and safety issues.

2.1 Memory Corruption

Since the infamous Morris Worm [3], memory corruption has been a recurring problem. Many countermeasures have been proposed to protect the programs. Some approaches prevent the attacker from accessing crucial information for exploiting program vulnerabilities, such

as Address Space Layout Randomization [35]. In contrast, others aim at detecting and preventing abnormal behavior of the program. Among this second class of protection, many previous works have studied CFI [2, 40], which detects and stops abnormal control-flow of the program.

In practice, CFI detects branches to invalid locations in the program, and it ensures that the return address of a function has not been hijacked. As ROP has become one of the main techniques to exploit memory corruption [9], CFI approaches aim at increasing the difficulty of using this technique. However, Shen et al. has shown that even in the presence of CFI, attackers can hijack the program's data flow to leak confidential data or modify some security-sensitive variables [13]. Castro et al. proposed to enforce the Data-Flow Integrity (DFI) to protect applications from such non-control data attacks [10]. However, the overhead induced by DFI is high. Thus, reducing the cost of DFI for RTS is crucial for the adoption of this protection.

2.2 Data-Flow Integrity

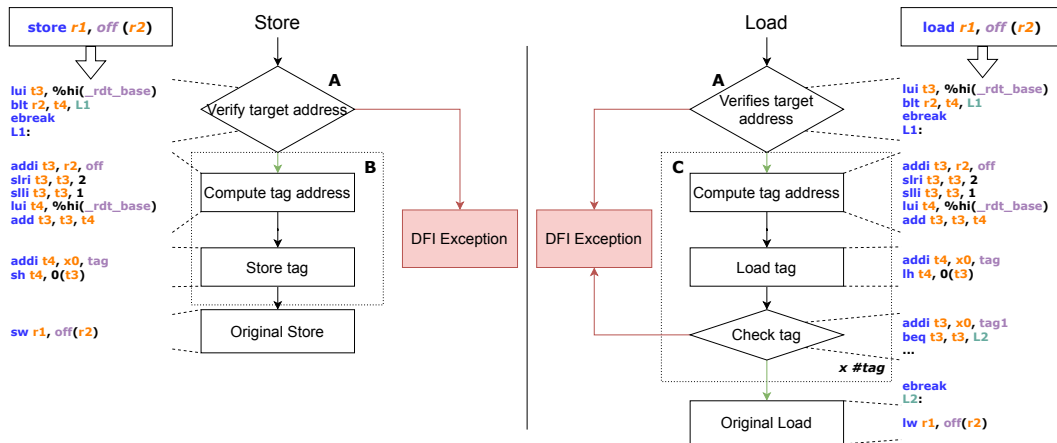
DFI as defined by Castro et al. [10] consists in instrumenting the original program code to assign a unique tag for each store instruction of the program and taint each written byte in memory with the tag of the corresponding store instruction. When the program loads the data later in its execution, DFI checks the data tag to ensure that it belongs to the *valid tag set* of the load. This set contains the tag of all the store instructions that can write the value read by the load. The valid tag set is built at compile-time, using data-flow information. The program is also instrumented to:

- (A) Prevent the original application code from accessing the memory area where the tags are stored.
- (B) Store the associated tag at each store instruction.
- (C) Load and verify the tag at each load instruction, using the *valid tag set*.

Figure 1 presents the steps that the instrumentation code executes at each load and store. We also provide a RISC-V example of an implementation for these steps. When the program executes a load or a store, it first verifies that the target address is neither in the *.text* segment nor in the memory part that contains the tags. The instrumentation code then computes the tag's address to either store or load the tag. If the protected instruction is a store, the instrumented code stores the tag before executing the original store instruction. If the protected instruction is a load, the tag is loaded and then checked against every tag in the *valid tag set* of the load. The initial load is considered legitimate and executed if the tag belongs to the *valid tag set*. Otherwise, DFI detects a data-flow error, considered as an attack.

A naive implementation of DFI can generate a code with more than 100 times the initial execution time [10]. Thus, optimizations have been proposed to reduce the overhead of DFI. The original paper [10] presents three optimizations: **equivalent classes optimization**, **redundant elimination**, and a greedy **tag check optimization**.

The **equivalent classes optimization** reduces the overall number of tags by detecting tags that always appear together in the same *valid tag sets* and regrouping them as a unique tag. This optimization reduces the overall number of tags used and, more importantly, reduces the number of checks required when a load occurs. In the original paper by Castro et al. [10], this optimization is the most important one, reducing the overhead of DFI below ten times the original execution time.



■ **Figure 1** DFI protection instrumentation. The left part represents the instrumentation of store instruction. The right part represents the instrumentation of load instructions. For each, we give an example of RISC-V implementation.

The **redundant check elimination** statically analyses the original program code to find successive loads or stores in the same basic block which target the same memory cell. When it discovers such successive instructions, it removes redundant protections. In the case of two successive loads, it removes the protection on the second load if the *valid tag set* of the first load is a subset of the *valid tag set* second. In the case of two consecutive stores with the same tag, we can remove the instrumentation for the second store since it would just overwrite the tag with itself. This optimization removes parts or the complete protection on loads and stores when it can statically ensure that the protection is redundant.

The **tag check optimization** improves the checking of tags for each load by checking multiple tags simultaneously. For this optimization, we differentiate between the tag (t), an identifier of the store that writes into a memory location, and the **tag representation** (r_t), an unsigned number encoding the tag, which the optimization can modify. Rather than verifying each tag of *valid tag set* one after the other in a *valid tag set*, we can regroup consecutive tag representations to form intervals. In this case, we can compare the tag of the loaded data with the limits of intervals defining the *valid tag set*. Checking intervals is particularly efficient for the biggest valid tag sets. Thus, the tag check optimization modifies the tag representations to reduce the number of intervals to check.

In the original article by Castro et al. [10], this last optimization uses a greedy algorithm that sorts the *valid tag sets* by decreasing value of the following metrics: $use \times size$ with use the number of load instructions that use the *valid tag set* and $size$ the size of the *valid tag set*. The algorithm then iterates on all the *valid tag sets*, in the order given by the metric, and assigns consecutive tag representations to the tags in each *valid tag set*, skipping the tags that already received a tag representation in a previous *valid tag set*.

The first issue of this algorithm is that it does not use any information about execution paths. In particular, loop bounds or branch frequencies could help to refine the representation allocation to focus on sets that are checked more frequently and not just that appear on more distinct loads. The second issue is that all the tags in a given set are assigned a new representation (except the tags that already have a new one) before analyzing another set. This local optimization prevents improvement based on multiple sets. For example, given the sets $\{A, B, C\}$ and $\{A, D, E\}$, the optimization may treat the first set and assign 1, 2, 3 to

respectively A , B , C and then consider the second set and assign 4 and 5 to respectively D and E . In this case, it would be more interesting to assign 3 to A such that the second *valid tag set* can also correspond to an interval.

■ **Table 1** Valid tag sets optimizations.

Valid tag sets		Tag representations
$S_1 = \{A, B, C, E\}$		$A \rightarrow 1 \quad D \rightarrow 4$
$S_2 = \{A, B, E\}$		$B \rightarrow 2 \quad E \rightarrow 5$
$S_3 = \{A, B, D\}$		$C \rightarrow 3$
----- Equivalent classes $A \sim B$ -----		
$S_1 = \{A, C, E\}$		$A \rightarrow 1 \quad D \rightarrow 4$
$S_2 = \{A, E\}$		$C \rightarrow 3 \quad E \rightarrow 5$
$S_3 = \{A, D\}$		
----- Tag check optimization -----		
$S_1 = \{A, C, E\}$	$\sim \llbracket 1, 3 \rrbracket$	$A \rightarrow 1 \quad D \rightarrow 4$
$S_2 = \{A, E\}$	$\sim \llbracket 1, 1 \rrbracket \cup \llbracket 3, 3 \rrbracket$	$C \rightarrow 2 \quad E \rightarrow 3$
$S_3 = \{A, D\}$	$\sim \llbracket 1, 1 \rrbracket \cup \llbracket 4, 4 \rrbracket$	
----- Optimal tag check -----		
$S_1 = \{A, C, E\}$	$\sim \llbracket 1, 3 \rrbracket$	$A \rightarrow 3 \quad D \rightarrow 4$
$S_2 = \{A, E\}$	$\sim \llbracket 2, 3 \rrbracket$	$C \rightarrow 1 \quad E \rightarrow 2$
$S_3 = \{A, D\}$	$\sim \llbracket 3, 4 \rrbracket$	

Table 1 presents the effect of the optimizations on *valid tag sets* on a simple example. The first column describes the *valid tag sets*, and the second column gives the tag representations. Dashed lines separate each optimization. We start with an example of three sets S_1 , S_2 , and S_3 . The *equivalent classes optimization* finds that tags A and B are equivalent and regroups them into tag A without modifying the tag representations, which creates a gap between the tag representations of A and C in S_1 . Then, the *tag check optimization* modifies the tag representations to remove this gap but, as it operates set by set, it fails to notice that A belongs both to S_1 and S_3 . Thus, the optimization cannot close the gap between the tag representation of A and D when it passes on S_3 . Finally, the figure presents a more sophisticated optimization that we would like to obtain.

Our work uses an ILP solver combined with information on the WCEP to tackle the two issues of *tag check optimization*. Notice that our goal is to improve the WCET rather than the average runtime.

2.3 Real-Time systems

RTS are computer systems with timing constraints, often due to their interaction with physical components. To ensure the respect of these constraints, we rely on analysis that estimates the WCET of each task. The WCET is then used with other scheduling data to verify that every task in the system has enough time to complete its execution. WCET estimation is performed on the executable file of the task to have all the low-level data available (e.g., processor instructions executed and the address of these instructions) and for a specific architecture to consider micro-architectural effects. As commonly accepted in the RTS community, WCET estimation is performed *in isolation*, leaving estimation and consideration of interferences between tasks to the schedulability analysis step.

Multiple analyses results are combined to perform WCET estimation. In particular, symbolic analyses of the cache, memory, and registers improve the precision of the WCET analysis. The value analysis safely determines the targeted memory area(s) of the instruction for each load and store of the program and the value that this instruction writes/reads in this memory area. This analysis results depend on the **context** of the instruction (e.g., the state of the call stack or the number of iterations of the loop containing the instruction). The analyses may over-approximate the results to avoid consuming too many resources (memory and time) and always provide a sound result. For instance, the value analysis provides a superset of actual memory locations. The returned set may also contain values that do not appear at run-time. The WCET is estimated using the results of these analyses for each basic block in the program. A path called WCEP, which maximizes the sum of the time of the basic blocks, is computed with a technique based on ILP called *Implicit Path Execution Technique (IPET)*.

We can retrieve much data from the WCET estimation as a byproduct of these analyses. In particular, knowing the WCEP allows targeting optimizations along this path, thus reducing the WCET. Such WCEP-oriented optimizations may result in a change of the WCEP, motivating iterative solutions to tackle WCEP-changes. RT-DFI uses path information to iteratively reduce the cost of the tag check of DFI .

3 Overview of RT-DFI, a WCET-directed DFI scheme

Decreasing the WCET of tasks improves the schedulability of a task set. Thus, we give in this section an overview of our approach named RT-DFI, which reduces the WCET of individual DFI-protected tasks. We designed RT-DFI such that it has the same level of protection as the original DFI. RT-DFI reduces the cost of tag check verifications using WCEP-oriented optimization. The optimization is iterative to tackle WCEP changes.

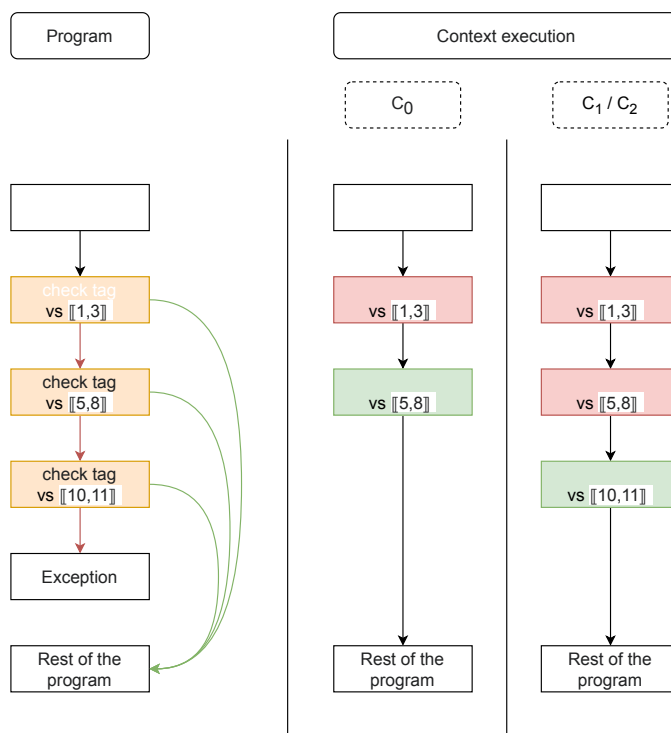
3.1 Using WCEP information to optimize tag checks

We focus our work on reducing the overhead of the tag check part of DFI, as this is the primary source of overhead [20]. The tag check overhead can be decomposed into two factors. The first factor is the number of checks required to cover the whole valid tag set of the load. Since the tags are checked using intervals, we can improve this factor by modifying the representation of the tags to reduce the number of intervals that cover the *valid tag set*. The second factor is the order in which we check these intervals. The tag check verifies if the tag belongs to each interval covering the *valid tag set*, and jumps to the rest of the code as soon as it finds an interval containing the tag. Thus, we can reduce the tag check overhead by first checking the most frequent intervals. In the rest of this paper, we use *interval order* to describe the order of the checks against the intervals.

RT-DFI uses the context data (number of executions in the WCEP, value analysis results) to optimize these two factors. To improve the WCET, we perform the optimization specifically on the WCEP. Thus, we only focus on contexts present in the WCEP. This approach also decreases the number of loads/contexts that we consider, reducing the complexity of our optimization problem.

In the rest of this paper, we assume that the WCET solver performs a symbolic value analysis and that we have access to its results. The WCET solver uses this value analysis to improve the IPET, avoiding paths in each context where the value analysis detects that these paths are infeasible. For example, the value analysis may establish that a condition is always satisfied in some context C_0 . In this case, the IPET only considers the path when the

condition holds for the context C_0 , even if the alternative path is more costly for the IPET in general. The same principle applies to the tag checks, which are a series of conditions. Thus, although all the tag checks are present in the program code, the value analysis may refine the IPET to bypass some checks in some contexts. When the value analysis cannot restrict the possible tag values for a given load, it still considers that the possible values are restricted by the **valid tag set** of the load since it reasons only on the legitimate executions of the program.



■ **Figure 2** Example of how checks can be bypassed in the WCEP.

For example, we present in Figure 2 a tag check with three intervals, $\llbracket 1, 3 \rrbracket$, $\llbracket 5, 8 \rrbracket$, and $\llbracket 10, 11 \rrbracket$. We also present the path considered by the IPET for three contexts C_0 , C_1 , and C_2 without optimization. We want to show how we can use the information from the value analysis to improve DFI overhead. Suppose the value analysis infers that the loaded tag is equal to 7 in context C_0 , is either 1 or 10 in context C_1 , and can take any value in context C_2 . For C_0 , the IPET considers the path that performs the first two checks and then jumps to the rest of the code, since the value analyses inferred that the tag belongs to the interval of the second check. Thus, the check of $\llbracket 1, 3 \rrbracket$ and $\llbracket 5, 8 \rrbracket$ count for the time of this context but not the check of $\llbracket 10, 11 \rrbracket$. In this case, it would be more optimal to place the check of $\llbracket 5, 8 \rrbracket$ as the first check for this context, as the IPET would skip the two other checks for the context C_0 . For C_1 , the IPET has to take into account the two possible values of the tag and can only jump to the rest of the code after the check of $\llbracket 10, 11 \rrbracket$. In this context, one way to optimize the WCET is to modify the interval orders to place $\llbracket 1, 3 \rrbracket$ and $\llbracket 10, 11 \rrbracket$ as the two first intervals (whatever the order of these two) to skip the check of the last interval. Another optimization is to modify the tag representations such as the new representations of 1 and 10 belong to the same interval. In this case, we can place this new interval as the first one to check, effectively skipping all other intervals. For the last context C_2 , the IPET

cannot skip any check, as it has no information on the tag value (except that it should be inside the *valid tag set*). Thus, the only way to optimize C_2 is to reduce the number of intervals that cover the *valid tag set*, by modifying the tag representations.

This example shows that different optimizations are possible in function of the information provided by the value analysis. We can change the interval orders, we can modify the tag representations, or do both. Modifying the interval orders only has a local impact on the WCEP. On the other hand, modifying the tag representations has consequences not only on the load we are focusing on, but also on every other load that has any of these tags in its *valid tag set*. Thus, knowing if a tag representation modification is interesting is more complex than for an interval order modification. Furthermore, multiple contexts for the same load can have conflicting optimizations. In our example, contexts C_0 and C_1 can both be optimized by modifying the interval orders, but the optimal order of C_0 conflicts with the optimal order of C_1 . To deal with this problem, we use an ILP solver that provides a good solution along the whole WCEP, taking into account global effects when modifying the tag representations.

To deal with the three possibilities seen in the previous example, we divide the contexts of the value analysis into three types, depending on the result of the value analysis:

- (a) The value analysis has determined exactly a single tag for this context (**known tag context**). In this case, we know that only one test will be true and once this test is executed, we jump to the rest of the program. Thus, we want to execute this test as soon as possible. This case corresponds to context C_0 in our previous example.
- (b) The value analysis found multiple possible tags, which are a strict subset of the *valid tag set* (**partially known tag context**). In this case, checks are skipped only when all the tags of the context are tested. Thus, we are interested in placing the tests for these tags as soon as possible, but the order between these tests is of no importance (for this context). Modifying the tag representations to reduce the number of intervals is also interesting. This case corresponds to context C_1 in our previous example.
- (c) The value analysis was unable to refine the possible values (i.e., the whole *valid tag set* is possible) (**unknown tag context**). In this case, no test can be skipped. Thus, the only way to reduce the cost of such context is to reduce the number of tests by finding a better tag representation for this *valid tag set*. This case corresponds to context C_2 in our previous example.

These different types of context represent how much information we have about if and when tests are skipped. The goal is then to use the context data to improve the tag representation and the interval orders on the WCEP. To do so, we use an ILP model that searches for a minimal number of tests along the WCEP. In contrast with the greedy algorithm of the first *tag check optimization* described in section 2.2, an ILP can provide better solutions that use all the information we have (see Section 5). In particular, while the greedy algorithm only performs local improvement load per load, an ILP can find optimizations that have a global impact on multiple loads. Furthermore, the current ILP solvers are very efficient and propose to set a timeout to stop the resolution after a given time, which can be used to obtain a good solution while having a bound on the time consumed by the solving algorithm.

The optimization flow is the following: the program is compiled the first time with DFI and all the optimizations present in the original paper by Castro et al. [10]. We then use the WCET analysis to construct an ILP that we optimize to find a better solution on the WCEP. We modify the program with this new solution. We then repeat the optimization process by creating a new ILP based on the new program executable, while still maintaining the same level of optimization on the previous paths. This allows RT-DFI to converge as the WCET reduces or stays the same. The process stops when the WCET does not improve anymore or after a given number of iterations.

3.2 Principle of the ILP

To optimize the WCEP, we use an ILP solver to minimize the number of checks on the current WCEP. We use the ILP to optimize the tag representation and, for each load on the WCEP, to find an optimal interval order. To do so, we use the result of the value analysis for each load in the WCEP. In particular, for each load, we segregate the contexts into known/partially known/unknown tag contexts, and we retrieve the **weight** of each context in the WCET, i.e., the number of times each context is executed in the WCEP. We also assume that check costs are the same on the WCET, whether they check against an interval or a single tag. This allows us to model single tags as singleton intervals in our ILP. This assumption is true in our implementation (see Section 5), where we generate code that checks intervals as fast as a single tag using only one branch instruction (with a method explained in [10]).

We construct the ILP with three steps:

1. We find a valid tag representation for each tag in the WCEP. We represent each tag as a variable, and we add constraints to prevent two tags from having the same value.
2. We find an optimized interval order for each load in the WCEP. For each load, we generate one variable per tag in the *valid tag set* of this load instruction, this variable representing the interval order of the interval containing the tag. We add constraints to these variables such that tags that must be in the same interval have the same order and tags that are in different intervals have different orders.
3. We aggregate the order variables of each load into a weighted sum that represents the objective function to optimize. For every context of a given load, the number of checks is the maximum of the order variables of the possible tags of the context (i.e., how much interval must be checked before we can skip the rest) multiplied by the **weight** of the context.

For example, in Table 2, we present four tags with their tag representations (for the entire program) and their interval orders (for a given load l). We see that for this load, tags A and B are tested first, then tag C and finally tag D . We also present 3 contexts C_0 , C_1 and C_2 with their possible tags and the cost associated with each context as well as the overall cost. The context C_0 has B as its only possible tag, and the WCEP only checks against the first interval before jumping to the rest of the code. Thus, the cost of C_0 is 1 (the interval order of B) times w_0 (the weight of C_0). For context C_1 , which has more possible tags (C and D), the WCEP passes by the three checks before jumping to the rest of the code. Thus, the cost of this context is 3 times w_1 . Finally, for an unknown context such as C_2 , the WCEP is forced to check against all the intervals. Thus, it has a cost of 3 times w_2 . The objective function is the sum of each context cost for each load.

To deal with changes of the WCEP we add new constraints to our ILP that prevent new optimizations from destructing the previous ones, which allows RT-DFI to converge. These new constraints have the following shape: '*previous objective function*' \leq '*previous objective function value*'. The '*previous objective function*' is constructed the same way as the current objective function, but with the context of the previous optimization. The '*previous objective function value*' is just the minimal value of the objective function found by the previous optimization. These constraints force the ILP to optimize the current WCEP while maintaining the same level of optimization on the previous path. Of course, if we have multiple previous WCEPs, we can add one constraint of this shape per previous WCEP.

■ **Table 2** An example of how the cost of the contexts are computed by the ILP for an arbitrary load l , knowing a tag representation and the interval order.

(a) Tag representations and interval order.

Tag	Representation	Interval Order (for load l)
A	2	1
B	3	1
C	5	2
D	9	3

(b) Contexts and associated costs. w_0 , w_1 and w_2 represent the *weight* of the contexts (the number of time they appear in the WCEP).

Context	Tags	Cost
C_0	B	$\max(1) \cdot w_0$
C_1	C,D	$\max(2, 3) \cdot w_1$
C_2	A,B,C,D	$\max(1, 2, 3) \cdot w_2$
All	A,B,C,D	$\max(1) \cdot w_0 + \max(2, 3) \cdot w_1 + \max(1, 2, 3) \cdot w_2$

4 Formal definition of the ILP for WCET-oriented tag check optimization

In this section, we present a formal definition of the ILP we use to optimize DFI on the WCEP. We first give a notation table and describe which data is available to construct the ILP in Subsection 4.1. We explain in Subsection 4.2 how to compute the number of checks for a load when the tag representations and the order of the intervals are known. We then present in Subsection 4.3 the ILP that minimizes the number of checks on the WCEP by optimizing the tag representations and the order of the intervals at the first iteration of the algorithm. Finally, we explain in Subsection 4.4 the constraints we add to the ILP to handle potential WCEP changes.

4.1 Notation table and problem formal definition

■ **Table 3** Notation table for the mathematical terms.

Notation	Type	Signification
$\llbracket a, b \rrbracket$	Interval	Integer interval between a and b
l	Load	A protected load
L	Set[Load]	Set of loads in the WCEP
t	Tag	A tag
T	Set[Tag]	Set of tags checked in the WCEP
r_t	\mathbb{N}	The tag representation of t (fixed)
s_l	Set[Tag]	Valid tag set of load l
$I_{l,t}$	Interval	Interval specific to l containing r_t
$\phi_{l,t}$	\mathbb{N}	Order of $I_{l,t}$ (check order) (fixed)
C_l	Context	A context for l in the WCEP
T_{C_l}	Set[Tag]	Possible tags for the context C_l
w_{C_l}	\mathbb{N}	Number of occurrence of C_l in the WCEP
N_l	\mathbb{N}	Number of checks of l in the WCEP
$Succ(t)$	Tag	t' such as $r_{t'} = r_t + 1$

Two tables are used to formally describe the ILP. Table 3 contains the mathematical terms we use. Note that when we use r_t or $\phi_{l,t}$, we consider them already fixed. Table 4 lists the ILP variables and constants used in the ILP formulation. A few notations represent values of the mathematical domain. The difference with the ones listed in Table 3 is that they are determined by the ILP solver and not fixed. We note them as *free*.

■ **Table 4** Notation table for ILP variables and constants.

Notation	Type	Signification
M	Constant	Number greater than any factor in the ILP (see big-M notation [22])
$start$	Constant Tag	Special tag used as the start of tag representation
end	Constant Tag	Special virtual tag used as the end of tag representation
V	Set[Tag]	$T \cup \{start, end\}$
v_t	\mathbb{N}	Vertex representing tag t
$entry_t$	\mathbb{N}	Number of edges entering v_t
$exit_t$	\mathbb{N}	Number of edges exiting v_t
$e_{t,t'}$	\mathbb{B} (boolean)	There is a directed edge from t to t' (or not)
R_t	\mathbb{N}	Represent r_t (free)
$\lambda_{l,t}^+$	\mathbb{B}	Represents if $Succ(t) \in s_l$ or not
$\Lambda_{l,t,t'}^+$	\mathbb{B}	$\lambda_{l,t}^+$ if $R_t < R_{t'}$ else $\lambda_{l,t'}^+$
$\Phi_{l,t}$	\mathbb{N}	Represent $\phi_{l,t}$ (free)
$\Phi_{l,t}^+$	\mathbb{N}	$\Phi_{l,t'}$ for t' such as $t' = Succ(t)$
$\Delta_{l,t,t'}$	\mathbb{N}	Represents $\ \Phi_{l,t} - \Phi_{l,t'}\ $
$\Delta_{l,t,t'}^+$	\mathbb{N}	Represents $\ \Phi_{l,t}^+ - \Phi_{l,t'}^+\ $
$\Gamma_{l,t,t'}$	\mathbb{N}	$\Delta_{l,\alpha,\beta}^+$ if $Succ(\alpha) \in s_l$ else 0 with $\alpha, \beta \in \{t, t'\}$, $R_\alpha < R_\beta$

We first recall the problem at hand and we explicit which data we have before the ILP. We then dive into the formal construction of the ILP in the next subsections. Our goal is to construct an ILP that can select the tag representations (globally) and interval orders (for each load on the WCEP) to minimize the number of checks on the current WCEP. To do so, we have data on all the contexts of each load in the WCEP. For each context C_l of the load l , present on the WCEP, we have the result of the value analysis (in the worst-case, the result is the valid tag set s_l of l) as well as the number of occurrences of C_l in the WCEP. When we want to iterate the optimization after a change of WCEP, we also consider that we have the same kind of data for the previous WCEP (as we just optimized it) as well as the value of the objective function of the last iteration. These data are used in subsection 4.4 to never undo previous optimizations.

4.2 Computing the number of checks

For this part, we consider that we know every tag representation, written r_t . We regroup the tag representations into intervals for each load l and we assign an arbitrary order to these intervals. Note that we can map s_l to a minimal number of intervals containing only valid tags. To do so, we consider all tags are single-element intervals and we merge adjacent intervals until there is no more fusion possible. We note $I_{l,t}$ the interval specific to l that contains r_t . As the intervals can be checked in an arbitrary order, we assign to each interval $I_{l,t}$ an index $\phi_{l,t}$ (starting at 1) which represents the order of the checks of the intervals (the interval with index 1 is checked first then the one with index 2, etc.). Note that for two tags t and t' , if $r_t \in I_{l,t'}$ then $I_{l,t} = I_{l,t'}$ and $\phi_{l,t} = \phi_{l,t'}$.

► **Example 1.**

t	r_t	$I_{l,t}$	$\phi_{l,t}$
A	1	$\llbracket 1, 2 \rrbracket$	1
B	2	$\llbracket 1, 2 \rrbracket$	1
C	4	$\llbracket 4, 4 \rrbracket$	2

We provide an example with 3 tags A , B and C in Example 1. As the tag representations are assigned for the entire program, the tag representations of A , B and C have no reason to be contiguous. We provide the interval of each tag and an arbitrary index for each interval.

Let C_l be a context for the load l with T_{C_l} the set of values provided by the value analysis and w_{C_l} the number of occurrences of C_l in the WCEP. We obtain the following number of checks for C_l in the WCEP:

$$\max_{t \in T_{C_l}} (\phi_{l,t}) \cdot w_{C_l} \quad (1)$$

This formula appears because the IPET only bypasses checks once they have covered all the possible tags of the context, and because the context appears w_{C_l} times in the WCEP.

The number of checks for a given load is an aggregation of the number of checks for every context of this load in the WCEP (2). The number of checks over the whole WCEP is the sum over all the loads (3).

$$N_l = \sum_{C_l} (\max_{t \in T_{C_l}} (\phi_{l,t}) \cdot w_{C_l}) \quad (2)$$

$$\sum_{l \in L} N_l \quad (3)$$

4.3 Transformation into an ILP problem

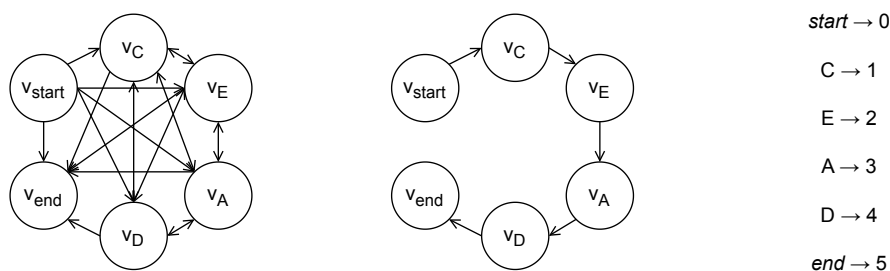
In the previous part, we described how to compute the number of checks once the tag representations and the interval orders are known. Thus, to construct the ILP, we only need to construct these two components. As we describe the ILP, we use a few notation shortcuts, $(x < y)$, $(x \cdot b)$ and $\max(x, y, \dots)$ to represent variables with the same value as these functions. For more information on how to construct such variables, we refer to [22].

We choose the tag representations to form a contiguous interval since this mapping requires the lesser space to store the tags during the program execution, and it maintains the space overhead of DFI.

In this case, we can sort the tags by their representation. Moreover, every tag, except the first and last tags of the interval, has a successor and a predecessor. These are precisely the properties of a path in a graph, which is easily expressible as an ILP. Furthermore, since, in principle, every tag can have any representation, we need a complete graph to allow all paths. Thus, we use a complete directed graph to construct the tag representations, whose vertices v_t represent the tags t .

In this graph, we want to select a vertex-cover path, which provides us with the tag representations. For a given path, the edge $e_{t,t'}$ from v_t to $v_{t'}$ is present if $r_{t'} = r_t + 1$. To ease the construction of the ILP, we introduce two virtual tags, $start$ and end , which are the start and end of the path.

In Figure 3, we present an example of the graph for 4 tags, plus $start$ and end . We also give an example of a path in this graph from v_{start} to v_{end} , and the corresponding tag representation mapping. We obtain this mapping by following the path and assigning consecutive representations to each tag.



■ **Figure 3** Example of the tag representation by the ILP.

We introduce $entry_t$ (resp. $exit_t$), which counts the number of edges entering (resp. exiting) the vertex v_t as well as R_t the variable containing the tag representation of t . The constraints for the path are the following¹²:

$$\sum_{t,t' \in V} e_{t,t'} = Card(V) - 1 \quad (4)$$

$$\forall t \in V, entry_t = \sum_{t' \in T \setminus \{t\}} e_{t',t} \quad (5)$$

$$\forall t \in V, exit_t = \sum_{t' \in T \setminus \{t\}} e_{t,t'} \quad (6)$$

$$\forall t \in T, entry_t = 1 \quad (7)$$

$$\forall t \in T, exit_t = 1 \quad (8)$$

$$\forall t, t' \in V, (R_{t'} + 1) - Card(T) \cdot (1 - e_{t',t}) \leq R_t \leq (R_{t'} + 1) + Card(T) \cdot (1 - e_{t',t}) \quad (9)$$

$$entry_{start} = 0, exit_{end} = 0, R_{start} = 0, R_{end} = Card(V) - 1 \quad (10)$$

Constraint (4) forces the path to have no more edges than necessary for a path passing by each vertex only once. Constraint (5) (resp. (6)) defines $entry_t$ (resp. $exit_t$). Constraint (7) (resp. (8)) forces each vertex except v_{start} (resp. v_{end}) to have only 1 entry edge (resp. 1 exit edge). Constraint (9) forces $R_t = R_{t'} + 1$ if and only if v_t is the vertex next to $v_{t'}$ in the path. Finally, constraint (10) deals with the special cases of tags $start$ and end . The overall design of this part of the ILP is a classic directed graph representation [14] combined with constraints (9) and (10).

We now explain how the ILP computes the interval orders. We create for each tag t and each load l a variable $\Phi_{l,t}$ that contains the interval order of $I_{l,t}$. In the case t does not belong to s_l , we assign an arbitrary number to $\Phi_{l,t}$ such that two different tags (both not belonging to s_l) have different indexes and such that these indexes are higher than the maximum interval index of l (i.e., greater than $Card(s_l)$). As the ILP computes the tag representations R_t , it must also compute the intervals and their orders, as the number of intervals and the interval themselves depends on the tag representations. We implicitly define the intervals with the following lemma:

► **Lemma 2.** $\forall t, t' \in T, I_{l,t} = I_{l,t'} \iff \phi_{l,t} = \phi_{l,t'}$

Lemma 2 expresses that if two tags are in the same interval, then they have the same index. Thus, we can encode in the ILP that two tags t and t' are in the same interval for the load l if and only if $\Phi_{l,t} = \Phi_{l,t'}$.

¹ We use the classic encoding of boolean variable in ILP with $false = 0$ and $true = 1$

² $Card(S)$ the cardinal of S

Writing constraints that represent if two tags t and t' are in the same interval (for a given l) is complex in the ILP, as it requires checking that every tag with a representation in between R_t and $R_{t'}$ belongs to s_l . To handle this problem, we use Lemma 3.

► **Lemma 3.** $\forall a \leq b, \forall S \subset \mathbb{N}, \llbracket a, b \rrbracket \subset S \iff (a \in S) \wedge (\llbracket a + 1, b \rrbracket \subset S)$

Rather than verifying that all the tags in between R_t and $R_{t'}$ belongs to s_l , we just verify that the $Succ(t)$ (considering that $R_t < R_{t'}$) belongs to s_l and let another part of the ILP handle the verification of a smaller interval. We can then recursively use the same Lemma until we have no more tags in between R_t and $R_{t'}$ to check.

We can thus rewrite these two lemmas to obtain the following relation:

$$\forall t, t' \in s_l, t \neq t', r_t < r_{t'}, \phi_{l,t} = \phi_{l,t'} \iff \phi_{l,Succ(t)} = \phi_{l,t'} \quad (11)$$

with $Succ(t)$ the tag t' such that $r_{t'} = r_t + 1$. Relation 11 explains that we can infer that two tags belong to the same interval by knowing if the successor of one of the tags belongs to this interval, as long as a few conditions are met. As we do not know whether $R_t < R_{t'}$ before executing the ILP, we use ILP variables that represent $\|R_{l,t} - R_{l,t'}\|^3$ and we build constraints that enforce Relation 11.

$$\forall t \in s_l, \Phi_{l,t}^+ = \sum_{t' \in T} (e_{t,t'} \cdot \Phi_{l,t'}) \quad (12)$$

$$\forall t \in V, \lambda_{l,t}^+ = \sum_{t' \in s_l} e_{t,t'} \quad (13)$$

$$\forall t, t' \in s_l, \Lambda_{l,t,t'}^+ = (R_t < R_{t'}) \cdot \lambda_{l,t}^+ + (R_{t'} < R_t) \cdot \lambda_{l,t'}^+ \quad (14)$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} = (\Phi_{l,t'} < \Phi_{l,t}) \cdot (\Phi_{l,t} - \Phi_{l,t'}) + (\Phi_{l,t} < \Phi_{l,t'}) \cdot (\Phi_{l,t'} - \Phi_{l,t}) \quad (15)$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'}^+ = (\Phi_{l,t'} < \Phi_{l,t}^+) \cdot (\Phi_{l,t}^+ - \Phi_{l,t'}) + (\Phi_{l,t}^+ < \Phi_{l,t'}) \cdot (\Phi_{l,t'} - \Phi_{l,t}^+) \quad (16)$$

$$\forall t, t' \in s_l, \Gamma_{l,t,t'} = (R_t < R_{t'}) \cdot \lambda_{l,t}^+ \cdot \Delta_{l,t,t'}^+ + (R_{t'} < R_t) \cdot \lambda_{l,t'}^+ \cdot \Delta_{l,t',t}^+ \quad (17)$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} \leq \Gamma_{l,t,t'} + (1 - \Lambda_{l,t,t'}^+) \cdot M \quad (18)$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} \geq \Gamma_{l,t,t'} + (1 - \Lambda_{l,t,t'}^+) \quad (19)$$

Constraint (12) defines $\Phi_{l,t}^+$, a variable that represents $\phi_{l,Succ(t)}$. Constraint (13) defines $\lambda_{l,t}^+$, a binary variable equal to 1 if and only if $Succ(t) \in s_l$. Constraint (14) defines $\Lambda_{l,t,t'}^+$, a binary variable equals to $\lambda_{l,a}^+$ with a the tag with the lowest tag representation between t and t' . Constraint (15) (resp. (16)) defines $\Delta_{l,t,t'}$ (resp. $\Delta_{l,t,t'}^+$) which represents $\|\phi_{l,t} - \phi_{l,t'}\|$ (resp. $\|\phi_{l,Succ(t)} - \phi_{l,t'}\|$). Constraint (17) defines $\Gamma_{l,t,t'}$, which represents $\Delta_{l,\alpha,\beta}^+$ if $Succ(\alpha) \in s_l$ else 0, with $\alpha \neq \beta \in \{t, t'\}$ s.a. $R_\alpha < R_\beta$. Finally, constraint (18) (resp. (19)) provide an upper bound (resp. lower bound) on $\Delta_{l,t,t'}$ that enforces the relation expressed in (11) and handles the case where $Succ(t)$ and/or $Succ(t')$ are not in s_l .

All these constraints organize the $\Phi_{l,t}$ variables such that we obtain the intervals and their orders. We can then use the $\Phi_{l,t}$ variables to build the objective function (3) (seen in Subsection 4.2) that we want to minimize.

4.4 Handling WCEP changes

When optimizing for the WCET, we must handle changes of the WCEP. In particular, incremental optimizations must ensure that an iteration does not destroy the work of a previous iteration. To do so with our ILP, we use additional constraints that prevent

³ $\|x\|$ being the absolute value of x

increasing the number of checks on the previous WCEPs. These constraints are of the form $objective_i \leq result_i$ with $objective_i$ being the objective function of the previous i^{th} iteration and $result_i$ the minimal value of this objective function, found by the previous iteration of the ILP solver.

As we use the same kind of variable as the real objective function, all the constructions explained previously remain the same. This approach also allows us to reduce the complexity of the ILP when the same tags/loads appear in two distinct WCEP (be it the previous or current one) as we can use the same variables and constraints to represent both.

5 Experimental results

To validate RT-DFI, we implemented it for a RISC-V LLVM compiler toolchain and applied it to a series of benchmarks from the TACLeBench benchmark suite. In subsection 5.1, we present how we implemented RT-DFI⁴ and the baseline DFI protection it is compared against, the benchmarks we used and the methodology. In subsection 5.2, we provide and analyze the results of this experiment. We also comment in subsection 5.4 on some DFI violations that we encountered in the benchmarks.

5.1 Experimental setup

We implemented RT-DFI using LLVM [28] as a compilation chain. PhASAR [34] produces the *valid tag set*. The compilation process is performed in the following way:

1. We transform the source code into *LLVM Intermediate Representation (IR)* with **O1** optimization switch (to avoid unnecessary load/store instructions that greatly increase the overhead of DFI).
2. We use *PhASAR* to produce the *valid tag sets* used by DFI and integrate them into the *LLVM IR* as annotations.
3. We apply the optimizations described in Section 2 on the *LLVM IR* to obtain the state-of-the-art DFI presented in [10].
4. We compile the *IR* into a *RISC-V* 32-bits executable for the *RudolV*⁵ processor. We integrated a last pass to the *RISC-V* backend of *LLVM* that transforms DFI annotations into assembly code. This prevents any separation between the protection and the protected instruction and handles the case where the instruction was not present in the *IR* (register scavenging, entry/exit of functions).
5. We execute RT-DFI, which produces a new IR that we can feed to the previous step.

This required to change $\sim 5,000$ lines of code in *LLVM* and $\sim 1,000$ lines of code in *PhASAR*. We use the industry standard for static timing analysis **aiT** [21] to compute the WCET, obtain the WCEP and perform value analysis. *CPLEX* [5] 20.1 is used to solve the ILP problems described in section 4. The rest of the implementation is written in Python 3.8. It orchestrates the programs (aiT, CPLEX, LLVM, PhASAR), it aggregates and unifies data retrieved from aiT and the compilation chain (in particular, the *valid tag sets*), it generates the ILP and finally, it modifies the *LLVM IR* to handle the new optimized tag representations and interval orders. This represents $\sim 8,000$ lines of code in Python.

For our experiments, we use TACLeBench [17]. This benchmark suite is composed of five groups of benchmarks:

⁴ <https://gitlab.inria.fr/nbellec1/rt-dfi>

⁵ <https://github.com/bobbl/rudolv>

1. **kernel** contains small kernel functions such as a binary search or an md5 hash.
2. **sequential** contains large function blocks (e.g. cryptographic or compression algorithms).
3. **test** contains three synthetic programs designed to challenge the WCET analysis tools.
4. **parallel** contains two modified real-world parallel applications, *Debie* and *PapaBench*.
5. **app** contains two real applications, *lift* and *powerwindow*.

A full benchmark description can be found in [17]. Out of the five groups, we did not run our experiments on the **parallel** group as we consider a bare-metal execution of the programs without relying on a Real-Time Operating System (RTOS), which defeats the purpose of the **parallel** group. We also excluded the *bitonic*, *bitcount*, *fac*, *quicksort*, *recursion*, *ammunition*, *anagram* and *huff_enc* benchmarks as it was harder to obtain the recursion bounds for the aiT WCET solver. Furthermore, *rijndael_dec* and *rijndael_enc* fail to pass aiT as they violate DFI, as explained in subsection 5.4.

We focus our experiment on the **aiT estimated WCET** after executing RT-DFI compared to the state-of-the-art DFI implementation. We perform two optimizations based on the data provided by aiT. The first optimization uses the value analysis of aiT to improve the *valid tag set* of our protection. In some cases, we can further reduce the *valid tag sets* provided by PhASAR by using the value analysis of aiT. In particular, the data-flow analysis used with PhASAR is field-insensitive (as [10]), and thus fails to distinguish between the cells of an array, which forces the analysis to keep some tags that could be removed. On the other hand, the value analysis of aiT is performed at the memory level and can help to refine the *valid tag set* of PhASAR. However, we can not use only aiT to obtain the *valid tag sets*, as their construction use information only available in the LLVM IR. This improvement of the *valid tag sets* is a byproduct of using the value analysis of a WCET solver to optimize the WCEP. We only present it in this section as the same result could be obtained by improving our data-flow analysis. Since it provides interesting results, we ought to present it. The second optimization is RT-DFI, presented in Section 4. For this second optimization, we executed four iterations for each benchmark, to address potential WCEP changes. However, we did not see improvement past the first iteration, so we just provided the improvement after the first iteration in our analysis of the results. We did not bound the ILP runtime, as all ILP were solved in less than 40 seconds.

5.2 Results

Figure 4 shows the normalized overhead factor of the state-of-the-art DFI on the WCET, with 1 the WCET of the program without DFI. 20 out of the 47 benchmarks have an overhead between $\times 1.03$ and $\times 1.6$ while the remaining 27 benchmarks have an overhead between $\times 1.9$ and $\times 5$. The mean overhead is $\times 2.38$. As observed in [10], DFI can incur a high overhead on the protected program, depending on the number of store/load instructions of the program. This confirms the need to reduce the impact of DFI on the WCET.

Figure 5 shows for each benchmark the improvement on the WCEP in percentage compared to the state-of-the-art when improving *valid tag sets* with the value analysis (*value analysis improvement* in the figure) and then with RT-DFI (*RT-DFI* in the figure). The mean improvement with both optimizations is 7.6% with a standard deviation of 4.4 percentage points. We note that for 29 benchmarks out of the 47, RT-DFI is the most impactful, while 18 of the benchmarks are more impacted by the value analysis improvements. However, among these 18 benchmarks, 10 have no improvement by RT-DFI because every load on the WCEP has a *valid tag set* with a single tag, thus preventing any further optimization. These 10 benchmarks are *adpcm_dec*, *complex_updates*, *cover*, *deg2rad*, *filterbank*, *fir2dim*, *iir*, *isqrt*, *prime* and *rad2deg*. All these benchmarks have either a very small WCET without protection

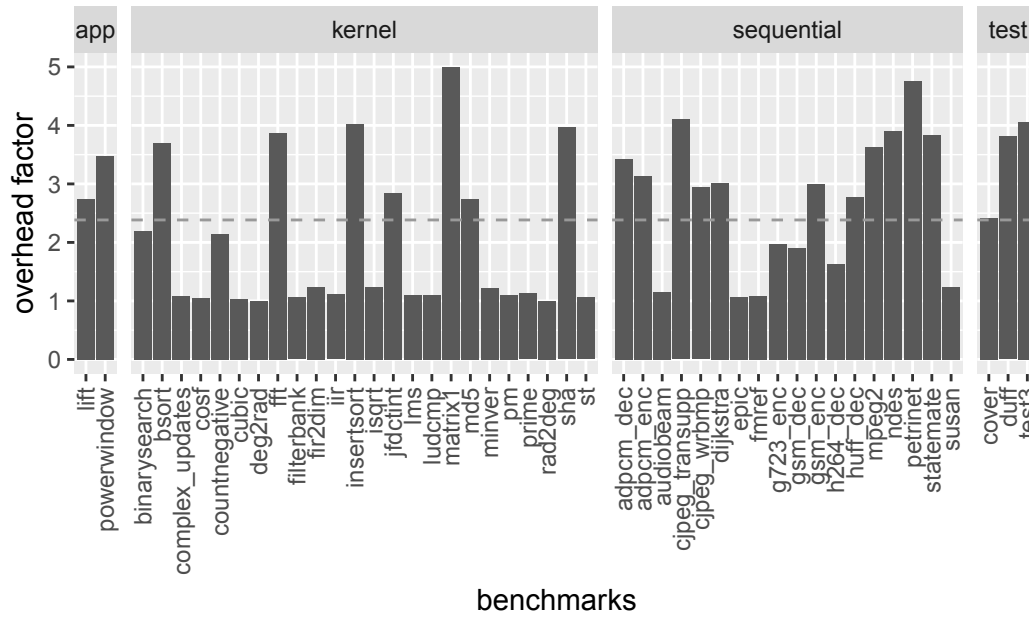


Figure 4 Overhead factor of DFI using the state-of-the-art DFI of [10] with the mean as a gray dashed line.

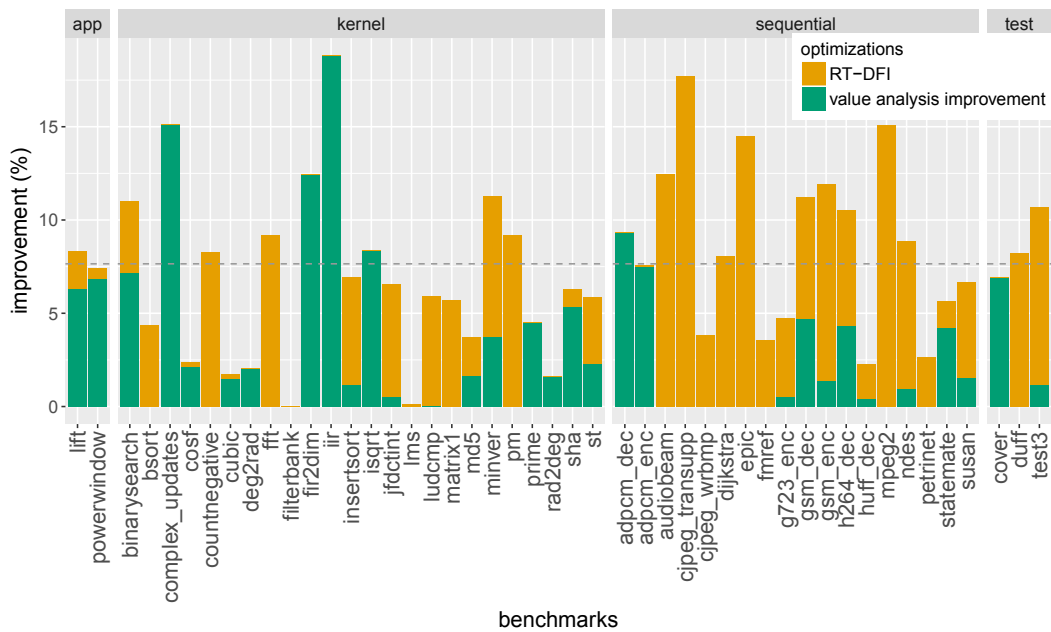


Figure 5 Improvement on the overhead of DFI with value-analysis optimization and RT-DFI. The cumulated mean is represented as a gray dashed line.

or a small DFI overhead (less than $\times 1.24$ if the WCET without protection is higher than 10,000 cycles). This tends to prove that having very small *valid tag sets* does reduce the overhead to an acceptable level. The overhead of the smallest benchmark skyrockets as they are mostly composed of load/store instructions.

Our optimization process spent most of its runtime executing the WCET analysis ($\sim 66\%$ on average) or compiling the new executable ($\sim 29\%$ on average). For all the benchmarks, the ILP solver part of RT-DFI took less than 40 seconds per problem to solve. Thus, the main runtime cost of RT-DFI is due to our iterative process that requires to re-launch a WCET analysis and re-build an executable at each step. As the number of steps remains low, we do not think this is an issue for the application of RT-DFI. Furthermore, as iterative optimization does not seem efficient, we only need two WCET analysis to perform RT-DFI.

5.3 Notes on iterative optimization

As explained before, we did not include the results for more than one iteration of RT-DFI, as more iterations do not further reduce the WCET. We have two hypotheses as to why iterations do not provide more improvement:

1. The constraints of the previous WCET prevent improvement on the new WCEP.
2. The WCEP has only small changes that have almost no impact on the final WCET⁶.

To test hypothesis 1, we relaxed the constraints of the previous WCEPs ($objective_i \leq 1.05 \cdot result_i$) to give more freedom to the solver. There was no overhead reduction past the first iteration of RT-DFI. While we can argue if the relaxation is enough and that it should depend on the WCET improvement, this tends to reject hypothesis 1. However, we must remain prudent and more experiments are required to fully reject this hypothesis.

Hypothesis 2 is hard to prove or reject because it is hard to compute path differences between two binaries, where instruction addresses and control-flow graphs may change due to the optimization. We manually examined and saw this problem arise for two benchmarks, namely *lift* and *powerwindow*, by visually examining the Control-Flow Graphs and WCEPs in aiT. However, this method is time-consuming and error-prone. The automatization requires handling changes in the tag representations/interval orders that are equivalent in terms of ILP and/or WCET but may change many parts of the program. As this is a complex issue, we do not have enough data to judge this hypothesis at the moment.

5.4 Notes on the security

Even without mounting a real attack scenario, RT-DFI detected errors on three benchmarks. The first two errors were detected by aiT in *rijndael_dec* and *rijndael_enc*. They are due to an off-by-one read to a buffer that loads a stack-saved register and triggers DFI. As aiT detects this in its value analysis, it considers that DFI exception is triggered and that its WCET analysis is unsafe. This means that providing aiT with a DFI-protected program can help find bugs that can be corrected before reaching the market. The third error appeared when executing *sha* using Qemu to test our DFI implementation. DFI detected a read to a saved register when executing the function *sha_wordcopy_fwd_aligned*. This function writes, at each iteration, a word into a buffer and then reads the word for the next iteration. Thus, at the last iteration, a word is read past the buffer, which triggers DFI. Note that

⁶ in particular, due to the nature of the benchmarks that contains few paths close to the WCEP

this error is detected at the execution by DFI, but not when using the analysis of *aiT*. This shows that even simple benchmarks can have non-trivial errors that are not detected, and the requirement for improved security protection in RTS.

6 Related work

The security of RTS has gained importance in the last decade. Closely related to DFI, CFI protection has been studied for real-time and embedded systems [1, 24]. Mishra *et al.* have written a survey on CFI techniques for RTS [32]. Using timing information to protect the program has also been studied [4, 41]. With the advancement of attack techniques that bypass CFI protection [8, 13], we wanted to focus on stronger protection.

Various implementations and variations of DFI have been studied. Song *et al.* presented HDFI [36], a hardware variation of DFI that uses 1-bit tags to isolate private data. Liu *et al.* studied TMDFI [29], a hardware DFI that modifies a lowRISC core with tag memory to reduce the overhead. Both these approaches modify the hardware and add specific DFI instructions to the ISA. Furthermore, the overhead is hard to predict, which does not fit real-time systems. Feng *et al.* [20] presented a hardware DFI protection that uses processor-in-memory combined with on-the-fly optimizations of memory reads/writes to reduce the memory contention of DFI. As with the previous approaches, specific hardware supports (in particular, processor-in-memory) are required and the overhead is not easily predictable. However, it does not require ISA modification and instead uses standard load/store instructions. Bresch *et al.* presented Trustflow [6], a hardware support for partial DFI with dedicated fast lookup tables. However, it only protects a subset of the data that are selected by hand, it requires hardware support and specific instructions. On the other hand, only protecting a small part of the data remove the overhead. In our work, we have full DFI protection, we do not need hardware support and the overhead is directly obtainable with a WCET solver.

Other methods to protect real-time systems have been proposed. Many researchers have explored vulnerabilities in scheduling methods [11, 27] and mitigations to these vulnerabilities [12, 42, 38, 26, 39, 33, 25]. Fellmuth *et al.* [19] proposed WCET-aware block-level artificial diversity to harden programs in RTS. Burow *et al.* [7] analyzed the impact of current moving target defenses (e.g., ASLR) on the WCET. Kadar *et al.* [24] studied syscall instrumentation to detect attacks in the context of embedded mixed-criticality systems.

7 Conclusion

The security of real-time systems has become an important subject in the last decade. DFI mitigates many potential memory corruption attacks. However, current DFI protections have an overhead that is either very high or hardly predictable. In this paper, we present a method to reduce the overhead of software-based DFI on the WCET. We present an ILP formulation that uses information retrieved from the WCEP to optimize the number of checks of DFI. Our experiment shows that this method helps to reduce the DFI overhead by a mean factor of 7.6%.

Our main limitation is that our method is designed for bare-metal applications. For an independent set of tasks, we could use the same method, with a few tweaks to ensure different tags for each task. However, dealing with tasks sharing resources and *RTOS* still requires more work. In particular, shared resources make the data-flow information used for DFI much harder to obtain precisely.

While our result shows that optimizing the tag checks on the WCET can reduce the overhead by a fair amount, other sources of overhead remain. In the future, we would like to study the overhead due to the imprecision of the data-flow analysis. Another important part of the overhead comes from computing at each store and load the address of the tag. Studying how to reduce this overhead could further reduce the global overhead of DFI. Studying the impact of hardware DFI on WCET also looks promising. Finally, we would like to study the execution of the benchmark applications on a real hardware.

References

- 1 Fardin Abdi Taghi Abad, Joel van der Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso, and Sibin Mohan. On-chip control flow integrity check for real time embedded systems. In *1st IEEE International Conference on Cyber-Physical Systems, Networks, and Applications, CPSNA 2013, Taipei, Taiwan, August 19-20, 2013*, pages 26–31. IEEE Computer Society, 2013. doi:10.1109/CPSNA.2013.6614242.
- 2 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS '05*, pages 340–353, New York, NY, USA, 2005. ACM. event-place: Alexandria, VA, USA. doi:10.1145/1102120.1102165.
- 3 anonymous. Morris worm, November 2021. Page Version ID: 1053313243. URL: https://en.wikipedia.org/w/index.php?title=Morris_worm&oldid=1053313243.
- 4 Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPICs*, pages 8:1–8:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ECRTS.2020.8.
- 5 Christian Blielikú, Pierre Bonami, and Andrea Lodi. Solving mixed-integer quadratic programming problems with ibm-cplex: a progress report. In *Proceedings of the twenty-sixth RAMP symposium*, pages 16–17, 2014.
- 6 Cyril Bresch, David Hély, Stéphanie Chollet, and Ioannis Parissis. TrustFlow: A Trusted Memory Support for Data Flow Integrity. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 308–313, July 2019. ISSN: 2159-3469. doi:10.1109/ISVLSI.2019.00063.
- 7 Nathan Burow, Ryan Burrow, Roger Khazan, Howard E. Shrobe, and Bryan C. Ward. Moving target defense considerations in real-time safety- and mission-critical systems. In Hamed Okhravi and Cliff Wang, editors, *Proceedings of the 7th ACM Workshop on Moving Target Defense, MTD@CCS 2020, Virtual Event, USA, November 9, 2020*, pages 81–89. ACM, 2020. doi:10.1145/3411496.3421224.
- 8 Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 161–176. USENIX Association, 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini>.
- 9 Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.
- 10 Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association. event-place: Seattle, Washington. URL: <http://dl.acm.org/citation.cfm?id=1298455.1298470>.
- 11 Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. A novel side-channel in real-time schedulers. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 90–102. IEEE, 2019. doi:10.1109/RTAS.2019.00016.

- 12 Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. Schedguard: Protecting against schedule leaks using linux containers. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 14–26. IEEE, 2021. doi:10.1109/RTAS52030.2021.00010.
- 13 Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.
- 14 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.
- 15 Irene Díez-Franco and Igor Santos. Data Is Flowing in the Wind: A Review of Data-Flow Integrity Methods to Overcome Non-Control-Data Attacks. In Manuel Graña, José Manuel López-Guede, Oier Etxaniz, Álvaro Herrero, Héctor Quintián, and Emilio Corchado, editors, *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16*, Advances in Intelligent Systems and Computing, pages 536–544, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-47364-2_52.
- 16 Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 901–913, New York, NY, USA, 2015. ACM. event-place: Denver, Colorado, USA. doi:10.1145/2810103.2813646.
- 17 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASISs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- 18 N. Falliere, L. O. Murchu, and E. Chien. W32. stuxnet dossier. *Whitepaper, Symantec Corp., Security Response*, 5:6, 2011.
- 19 Joachim Fellmuth, Paula Herber, Tobias F. Pfeffer, and Sabine Glesner. Securing real-time cyber-physical systems using wcet-aware artificial diversity. In *15th IEEE Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2017, Orlando, FL, USA, November 6-10, 2017*, pages 454–461. IEEE Computer Society, 2017. doi:10.1109/DASC-PiCom-DataCom-CyberSciTec.2017.88.
- 20 Lang Feng, Jiayi Huang, Jeff Huang, and Jiang Hu. Toward Taming the Overhead Monster for Data-Flow Integrity. *arXiv:2102.10031 [cs]*, February 2021. arXiv: 2102.10031. arXiv: 2102.10031.
- 21 Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.
- 22 Igor Griva, Stephen G Nash, and Ariela Sofer. *Linear and nonlinear optimization*, volume 108. Siam, 2009.
- 23 Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. *CoRR*, abs/1711.04808, 2017. arXiv:1711.04808.
- 24 Marine Kadar, Gerhard Fohler, Don Kuzhiyelil, and Philipp Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 292–305. IEEE, 2021. doi:10.1109/RTAS52030.2021.00031.

- 25 Kristin Krüger, Gerhard Fohler, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Improving security for time-triggered real-time systems with task replication. In *24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018, Hakodate, Japan, August 28-31, 2018*, pages 232–233. IEEE Computer Society, 2018. doi:10.1109/RTCSA.2018.00036.
- 26 Kristin Krüger, Marcus Völp, and Gerhard Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPICs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ECRTS.2018.22.
- 27 Jaeheon Kwak and Jinkyu Lee. Covert timing channel design for uniprocessor real-time systems. In Jong Hyuk Park, Hong Shen, Yunsick Sung, and Hui Tian, editors, *Parallel and Distributed Computing, Applications and Technologies, 19th International Conference, PDCAT 2018, Jeju Island, South Korea, August 20-22, 2018, Revised Selected Papers*, volume 931 of *Communications in Computer and Information Science*, pages 159–168. Springer, 2018. doi:10.1007/978-981-13-5907-1_17.
- 28 Chris Lattner and Vikram Adve. Llvn: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, CGO 2004.*, pages 75–86. IEEE, 2004.
- 29 Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu Zhang. TMDFI: Tagged Memory Assisted for Fine-Grained Data-Flow Integrity Towards Embedded Systems Against Software Exploitation. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 545–550, August 2018. ISSN: 2324-9013. doi:10.1109/TrustCom/BigDataSE.2018.00083.
- 30 Tingting Lu and Junfeng Wang. Data-flow bending: On the effectiveness of data-flow integrity. *Computers & Security*, 84:365–375, July 2019. doi:10.1016/j.cose.2019.04.002.
- 31 Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.
- 32 Tanmaya Mishra, Thidapat Chantem, and Ryan M. Gerdes. Survey of control-flow integrity techniques for embedded and real-time embedded systems. *CoRR*, abs/2111.11390, 2021. arXiv:2111.11390.
- 33 Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 103–116. IEEE, 2019. doi:10.1109/RTAS.2019.00017.
- 34 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for C/C++. In *TACAS (2)*, pages 393–410, 2019.
- 35 Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security, CCS '04*, pages 298–307, New York, NY, USA, October 2004. Association for Computing Machinery. doi:10.1145/1030083.1030124.
- 36 Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, May 2016. ISSN: 2375-1207. doi:10.1109/SP.2016.9.
- 37 Victor van der Veen, Dennis Andriess, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 1675–1689, New York, NY, USA, 2017. ACM. event-place: Dallas, Texas, USA. doi:10.1145/3133956.3134026.

- 38 Marcus Völz, Claude-Joachim Hamann, and Hermann Härtig. Avoiding timing channels in fixed-priority schedulers. In Masayuki Abe and Virgil D. Gligor, editors, *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008, Tokyo, Japan, March 18-20, 2008*, pages 44–55. ACM, 2008. doi:10.1145/1368310.1368320.
- 39 Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. Minimizing side-channel attack vulnerability via schedule randomization. In *58th IEEE Conference on Decision and Control, CDC 2019, Nice, France, December 11-13, 2019*, pages 2928–2933. IEEE, 2019. doi:10.1109/CDC40024.2019.9030144.
- 40 Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-Flow Integrity for Real-Time Embedded Systems. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:24, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.2.
- 41 Julian Wolf, Bernhard Fechner, Sascha Uhrig, and Theo Ungerer. Fine-grained timing and control flow error checking for hard real-time task execution. In *7th IEEE International Symposium on Industrial Embedded Systems, SIES 2012, Karlsruhe, Germany, June 20-22, 2012*, pages 257–266. IEEE, 2012. doi:10.1109/SIES.2012.6356592.
- 42 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 111–122. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461362.

A Full experimental data

■ **Table 5** Experimental results. **Vanilla** is the WCET of the program without protection. **SotA** is the overhead for state-of-the-art DFI protection. **VA++** is the improvement compared to SotA using the value analysis to improve the valid tag sets. **R1** (resp. **Best**) represents the improvement of RT-DFI with 1 iteration (resp. 4 iterations) compared to SotA.

Bench	Vanilla	SotA	VA++	R1	Best
adpcm_dec	2,306	+243.24%	-9.34%	-9.34%	-9.34%
adpcm_enc	3,306	+213.22%	-7.52%	-7.59%	-7.59%
audiobeam	5,653,355	+15.66%	-0.01%	-12.45%	-12.45%
binarysearch	94	+119.15%	-7.14%	-10.71%	-10.71%
bsort	167,333	+269.44%	0.00%	-4.35%	-4.35%
cjpeg_transupp	10,321,821	+310.02%	-0.00%	-17.72%	-17.72%
cjpeg_wrbmp	165,680	+193.86%	0.00%	-3.83%	-3.83%
complex_updates	23,941	+7.94%	-15.14%	-15.14%	-15.14%
cosf	794,173	+4.77%	-2.10%	-2.36%	-2.36%
countnegative	4,228	+114.50%	0.00%	-8.26%	-8.26%
cover	51	+141.18%	-6.94%	-6.94%	-6.94%
cubic	34,348,748	+3.77%	-1.49%	-1.73%	-1.73%
deg2rad	304,652	+0.03%	-2.04%	-2.04%	-2.04%
dijkstra	3,624,311,302	+200.99%	-0.00%	-8.09%	-8.09%
duff	372	+280.91%	0.00%	-8.23%	-8.23%
epic	11,032,999,977	+7.40%	0.00%	-14.53%	-14.53%
fft	90,043,546	+286.45%	0.00%	-9.18%	-9.18%
filterbank	99,168,970	+5.74%	-0.00%	-0.00%	-0.00%
fir2dim	33,301	+23.43%	-12.42%	-12.42%	-12.42%
fnref	18,678,944	+8.05%	-0.01%	-3.55%	-3.55%
g723_enc	717,091	+96.79%	-0.52%	-4.72%	-4.72%
gsm_dec	1,910,861	+90.82%	-4.71%	-10.90%	-10.90%
gsm_enc	3,719,804	+200.28%	-1.37%	-11.75%	-11.75%
h264_dec	48,385	+62.63%	-4.34%	-10.25%	-10.25%
huff_dec	625,471	+177.46%	-0.42%	-2.28%	-2.28%
iir	5,907	+11.07%	-18.81%	-18.81%	-18.81%
insertsort	1,148	+302.26%	-1.15%	-6.86%	-6.86%
isqrt	698,732	+23.93%	-8.38%	-8.38%	-8.38%
jfdctint	1,593	+184.18%	-0.55%	-6.54%	-6.54%
lift	696,735	+174.71%	-6.33%	-8.22%	-8.22%
lms	3,524,882	+9.65%	0.00%	-0.12%	-0.12%
ludcmp	127,564	+10.44%	-0.03%	-5.92%	-5.92%
matrix1	8,764	+399.36%	0.00%	-5.71%	-5.71%
md5	13,888,069	+174.65%	-1.65%	-3.70%	-3.70%
minver	46,554	+21.97%	-3.74%	-10.97%	-10.97%
mpeg2	2,677,246,923	+263.49%	-0.00%	-15.09%	-15.09%
ndes	116,203	+290.20%	-0.95%	-8.77%	-8.77%
petrinet	2,577	+376.06%	0.00%	-2.66%	-2.66%
pm	197,879,495	+10.17%	-0.00%	-9.18%	-9.18%
powerwindow	2,717,332	+247.30%	-6.86%	-7.37%	-7.37%
prime	1,755	+12.65%	-4.50%	-4.50%	-4.50%
rad2deg	306,691	+0.04%	-1.61%	-1.61%	-1.61%
sha	1,933,236	+296.43%	-5.34%	-6.22%	-6.22%
st	2,991,897	+5.78%	-2.31%	-5.78%	-5.78%
statemate	124,111	+283.22%	-4.21%	-5.56%	-5.56%
susan	388,703,083	+24.24%	-1.55%	-6.57%	-6.57%
test3	235,916,292	+305.47%	-1.17%	-10.57%	-10.57%

Foundational Response-Time Analysis as Explainable Evidence of Timeliness

Marco Maida ✉

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Sergey Bozhko ✉

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
Saarbrücken Graduate School of Computer Science, Universität des Saarlandes, Germany

Björn B. Brandenburg ✉

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

The paper introduces *foundational* response-time analysis (RTA) as a means to produce strong and independently checkable *evidence of temporal correctness*. In a foundational RTA, each response-time bound calculated comes with an auto-generated *certificate* of correctness – a short and human-inspectable sequence of *machine-checked proofs* that formally show the claimed bound to hold. In other words, a foundational RTA yields *explainable* results that can be independently verified (e.g., by a certification authority) in a rigorous manner (with an automated proof checker). Consequently, the analysis tool itself does *not* need to be verified nor trusted. As a proof of concept, the paper presents POET, the first foundational RTA tool. POET generates certificates based on PROSA, the to-date largest verified framework for schedulability analysis, which is based on COQ. The trusted computing base is hence reduced to the COQ proof checker and its dependencies. POET currently supports two scheduling policies (*earliest-deadline-first*, *fixed-priority*), two preemption models (*fully preemptive*, *fully non-preemptive*), arbitrary deadlines, periodic and sporadic tasks, and tasks characterized by arbitrary arrival curves. The paper describes the challenges inherent in the development of a foundational RTA tool, discusses key design choices, and reports on its scalability.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Formal software verification

Keywords and phrases hard real-time systems, response-time analysis, uniprocessor, Coq, Prosa, fixed priority, EDF, preemptive, non-preemptive, verification

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.19

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.8.1.7>

Funding This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 803111), and from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 391919384.

Acknowledgements We thank Pierre Roux for introducing us to COQEAL and the members of the joint ANR-DFG project RT-PROOFS for fruitful discussions.

1 Introduction

The purpose of a *response-time analysis* (RTA) is to obtain safe bounds on the worst-case response times of all critical tasks in a real-time system. To this end, the system is described with a mathematical model, which typically comprises a workload model, a resource model, and a scheduling policy. The model is then analyzed to derive response-time bounds, which requires (i) a *theory* that rigorously justifies that the RTA correctly characterizes the worst-case scenario, and (ii) an RTA *tool* that executes the concrete calculations.



© Marco Maida, Sergey Bozhko, and Björn B. Brandenburg;
licensed under Creative Commons License CC-BY 4.0
34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio; Article No. 19; pp. 19:1–19:25



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Both aspects are equally critical. An error in (i), such as an invalid over-generalization or a missed corner case, leads to a flawed theory. An error in (ii), which can be any significant bug in the tool, leads to a flawed implementation. Given the number of documented analysis mistakes in the real-time literature (e.g., [13, 18, 28, 34]), and the reality that complex tools are rarely bug-free, both the research community and industry have shown a growing interest in the application of computer-assisted formal verification (e.g., [11, 21, 23]), with safety standards also advising its use (e.g., ISO 26262, DO-178C).

However, applying the currently available proof assistants and verification tools to mechanize RTAs and formally verify RTA tools is neither an easy nor a cheap task. Once a theory – or the behavior of a program – is encoded in a proof assistant’s specification language, it usually needs to be augmented with additional information (e.g., step-by-step proofs, program invariants) before the verification procedure can succeed. This process requires human intervention and usually takes a considerable amount of time. Moreover, developing and maintaining a verified tool requires advanced programming skills and specialized expertise. As a consequence of such cost and knowledge barriers, RTA tools in use today are typically not verified, and this is unlikely to change in the foreseeable future.

In this paper, we therefore explore the challenge of obtaining trustworthy response-time bounds without having to verify the RTA tool that produces them. To this end, we propose *foundational RTA* as means to produce evidence of timeliness that can be independently checked in a rigorous manner by an untrusting third party. As elaborated in Section 2, a foundational RTA tool must produce *proof-carrying response-time bounds*, that is, bounds that come with *certificates of correctness* that can be automatically checked by a proof assistant such as COQ. The key advantage of a foundational RTA is that the *trusted computing base* (TCB) is reduced to only the proof checker and its dependencies: the analysis tool itself does *not* need to be trusted because the correctness of its claims can be easily and independently verified. Therefore, the RTA tool’s source code can be updated, modified, ported, and optimized like any other non-critical software.

The advantages of foundational RTA are obvious; its practical realizability and scalability, however, much less so. As a proof of concept, we present the design and implementation of the first foundational RTA tool. Our RTA tool, called POET (*Prosa Obsigned Evidence of Timeliness*), works in conjunction with the formally verified PROSA framework for real-time schedulability analysis [11], and in particular with its *abstract RTA* library [8]. When a problem instance (i.e., a concrete task set, scheduling policy, and preemption model) is given as input, POET generates, along with the usual response-time bounds, a set of formal proofs of correctness that can be automatically machine-checked by the COQ proof assistant [42]. The process is completely automated (i.e., users do *not* need to write definitions nor to prove theorems) and does not require any expertise with formal verification on behalf of the user. Consequently, POET is the first RTA tool that can assert the correctness of its results in a trustworthy manner without any need for the tool itself to be verified.

To summarize, this paper makes the following **contributions**:

- We introduce the notions of foundational RTA and proof-carrying response-time bounds (Section 2), along with the first foundational RTA tool, POET, and discuss key design and implementation challenges (Section 4);
- describe how mechanisms in POET and the overall workflow ensure the trustworthiness of the certificates (Section 5),
- discuss how POET computes the search space for each task’s response-time bound in a manner that is both computationally efficient and verifiable (Section 6);
- explain how we enabled POET to scale to numerically large task parameters, despite PROSA’s proof-oriented (but computationally inefficient) number representation (Section 7); and finally

- report on an empirical evaluation of POET on synthetic task sets of realistic complexity in terms of task count, utilization, and numerical magnitude of the parameters (Section 8). Last but not least, POET makes it possible for users unfamiliar with COQ to benefit from the power of formal verification. POET will be released as an open-source project.

2 Design Space of Verified RTAs

To motivate the advantages of foundational RTA and to provide context, we begin with a brief survey of the space of possible alternatives. Given that the literature on real-time systems in general, and on RTAs in particular, targets mission- and safety-critical systems – such as cars, trains, aircraft, and spacecraft – RTA tools are an obvious candidate for verification. However, while the ultimate goal is clear – the computed bounds must never be optimistic – it is less obvious what it means to actually “verify an RTA tool” in practice. In fact, there are many ways in which formal verification could improve the trustworthiness of RTA tools, with differing trade-offs in terms of required effort and verification coverage.

Common flaws. To better understand the space of possible solutions, first consider the following (non-exhaustive) list of flaws threatening the validity of response-time bounds. The most obvious issue is that the *underlying RTA theory may be incorrect* (**F1**). Sadly, the published record shows that this is far from just a theoretical concern: many cases of flawed reasoning have been documented [12, 18, 28, 34, 49], Chen et al. [13] and Cerqueira et al. [11] give more examples of incorrect analyses of processor scheduling, Indrusiak et al. [31] list *eight* refuted RTAs for on-chip networks (NoCs), and new corrections still continue to appear on a regular basis (e.g., [29, 52]). At this point in time, it has become painfully clear that engineers would be foolish to trust an analysis to be correct just because it was peer-reviewed and published at a well-reputed venue. Formal verification is arguably the best, and perhaps the only way, to overcome this trustworthiness gap, especially in critical systems.

Moving on, even if the underlying analysis is not at fault, then a particular implementor may still *misinterpret the published analysis* (**F2**), or simply overlook *logic errors in the implementation* (**F3**) such as incorrect loop conditions or incomplete search-space traversals. From first-hand experience, such issues do not appear to be uncommon. In a similar vein, easy-to-overlook *programming mistakes resulting in silent errors* (**F4**), such as over- or underflow during unchecked integer arithmetic or floating-point precision issues, plague RTA tools just like they do applications in virtually all other domains.

Likely somewhat less common, but a real concern nonetheless, is the possibility that *software defects in other parts of the tool* (**F5**) corrupt the state of the analysis implementation, such as a dangling pointer in an XML or JSON parsing library, or a concurrency issue. This can also affect tools programmed in otherwise memory-safe languages (such as OCAML or JAVA) that link with native C libraries for common functionality.

Another concern related to I/O is that the *parsing of the workload and system description* (**F6**), especially if it is presented in a difficult-to-process format, may silently alter the input such that an otherwise correct analysis is run on a slightly, but critically different problem instance than intended. This is a particular concern for tools that do not produce *explainable* results, that is, for tools that produce output from which it is not possible for a human to understand how, or based on exactly which model, the result was obtained.

Last, and also decidedly least, is the concern that *transient errors*, such as bit flips due to cosmic background radiation or thermal noise, could affect the RTA tool during its execution (or the verifier, for that matter). While not unheard of, the likelihood of occurrence on an

engineer’s workstation or laptop is extremely low. Furthermore, it can be safely assumed that any critical system is analyzed not just once, and not only on just one machine, at which point the risk of repeated transient errors at design time becomes negligible.

Clearly, an ideal “verified RTA” should mitigate all of these flaws (and more). Additional desiderata include that it should be reasonably fast and scale to workloads of industrially relevant size and parameter magnitudes (e.g., expressed in processor cycles or nanoseconds). Furthermore, an ideal RTA tool should be explainable in the sense that it justifies its results in a way that can independently be checked by a third party (e.g., government regulators) for use in (non-formal) certification processes (e.g., as commonly encountered in the avionics and medical domains, and increasingly also in the automotive domain). How can these goals be achieved? While a full survey of the verification literature is beyond the scope of this paper, we can generally identify three fundamental approaches: *verify the tool* itself, augment an unverified tool with a verified *results validator*, and the proposed *foundational approach*.

Verified implementation. In principle, one could verify the entire RTA tool, in particular if it is implemented in a verification-friendly language such as OCAML or HASKELL. To our knowledge, this has not been done to-date, and whether it is even worthwhile in the context of RTA tools is a point open for debate. Nonetheless, it is interesting to consider different levels of *specifications* that such an effort could target.

The weakest specification would be that the tool computes a particular mathematical expression, or finds a fixed-point of a certain structure, without any formal claims about the semantical implications of this. In other words, the tool would be verified to correctly compute a number, but no formal claim is made about the meaning of this number (i.e., that it is a bound). The advantage of this is that the underlying RTA does *not* have to be verified, i.e., this approach could be used with any RTA in the literature. While this requires proving the absence of F3- and F4-type flaws, it fails to mitigate F1- and F2-type flaws.

A step up would be to use the same specification for the implementation (of just the computation, without semantics), while also verifying the employed RTA *theory* in a separate effort (or using an already-verified one, e.g., [8, 11]). The resulting assurance is much improved, as this approach prevents the historically very problematic F1-type flaws. However, without an end-to-end verification, F2-type flaws still cannot be ruled out.

Ideally, the tool’s specification should include *semantic guarantees*: if the tool claims a computed number to be a response-time bound, then there does not exist a schedule (i.e., a trace of the analyzed model) in which the bound is exceeded. Clearly, this would be the most desirable level to target as it completely eliminates the risk of any flaws in categories F1–F4.

The limiting factor is the implied amount of work: not only would such a semantically rich specification require a complete verification of the underlying theory (e.g., as done in PROSA [11]), but to also categorically avoid F5- and F6-type issues, a deep specification and verification of all “plumbing” code (such as the input format parser) would have to be carried out. With the contemporary software ecosystem and verification tools, this would be a daunting task of questionable feasibility. On top of this, a verified RTA tool would not inherently produce any additional evidence of correctness besides the computed bounds (i.e., its output is still not explainable). On the plus side, there is no reason to doubt that such a tool could be fast. However, to the best of our knowledge, there have been no attempts to-date to verify an entire RTA tool, likely due to the enormous effort that would be required.

Results validator. A different approach that side-steps the issue of having to verify the entire RTA tool with all its real-world complexities is to instead *check only the analysis results* [23, 36]. The high-level idea is to leave the actual RTA implementation unverified

(with all the software-engineering flexibility this brings), and to develop a *second*, verified tool that independently validates the results of the RTA tool. If it is fundamentally easier to check a solution than to find one, then this approach is beneficial in principle.

While much more attainable than a fully verified RTA implementation – as successfully demonstrated on industrially relevant network analyses [23, 36] – it comes at the price of having to develop the formally verified results validator, which is subject to all the aforementioned challenges related to verified code. Case in point, Fradet et al.’s CERTICAN [23] checking procedure is first verified with COQ and then automatically extracted from COQ as an OCAML file. However, to actually obtain a useful, runnable tool, the extracted OCAML code must still be combined with non-verified support code to handle “plumbing” tasks such as I/O and parsing. As a result, F5- and F6-type issues remain plausible concerns.

Proof-carrying bounds. In this paper, we seek to develop a way of obtaining trustworthy response-time bounds that avoids verifying or trusting the RTA tool altogether. To this end, inspired by Appel’s seminal work on *foundational proof-carrying code* [3], we transfer the foundational approach to the RTA setting. To elaborate on the original concept, proof-carrying code is a (usually) low-level program, such as a sequence of instructions emitted by a compiler, that comes with an independently checkable proof establishing its properties [38]. A foundational proof-carrying code [3] has a proof that relies only on the foundations of mathematical logic, hence minimizing the number of additional components (ad-hoc type systems, verification condition generators, one-off output checkers, etc.) to be trusted.

Likewise, we require a *foundational RTA* to produce response-time bounds that come with a proof of correctness relying only on the mathematical logic implemented by the underlying proof assistant, i.e., a foundational RTA yields *proof-carrying response-time bounds*.

To demonstrate the practicality of this idea, we present the first such tool, POET, as a proof of concept. By analogy, POET behaves like a successful student during an exam: when faced with a tricky computational problem (i.e., finding the response-time bound of a task), POET not only yields the final solution (i.e., a number), but also justifies with a sequence of proof steps *why* the solution is correct. In other words, the solution is *explainable* in the sense that its derivation can be understood without interviewing the student about their thought processes (i.e., knowing POET’s source code). POET thus does not have to be trusted at all, since its answer can be independently verified.

As a foundational proof, by definition, depends only on the foundations of mathematical logic, it necessarily includes a complete justification of both the underlying RTA and how it applies to the specific workload. POET hence categorically avoids all flaws of types F1–F5, and with light supervision (Section 5) sidesteps F6-type issues altogether, while inherently producing independently checkable evidence of timeliness.

The *Achilles’ heel* of the foundational approach, however, is that its runtime is closely tied to that of the underlying proof assistant that checks the generated certificates, which, as we discuss in Sections 6–8, was a serious challenge in the realization of POET.

3 Background

POET relies on abstract RTA [8], which has been integrated into PROSA [40], a COQ [42] framework. We briefly review each of these building blocks in turn.

COQ [42] is a widely-used, mature interactive theorem prover based on the *calculus of inductive constructions* [15, 16, 39] that provides rich support for the *mechanization* (i.e., the formalization and machine-checked verification) of both nontrivial mathematical theories (e.g., [25, 26]) and large software systems (e.g., [35]). COQ provides primarily two languages:

GALLINA, a formal, dependently typed specification language used to write mathematical definitions, state theorems, and implement functional programs, and LTAC, an untyped macro language used to steer the proof engine.

COQ is not a fully automatic theorem prover: while proof *checking* is automatic, proof *authoring* typically requires human intervention. Once a theorem is stated, it is necessary to provide a sequence of LTAC *tactic* applications (each of which can be seen as a single step of the proof) that, starting from the stated hypotheses, allows the proof engine to reach the claimed conclusion. As COQ allows the user to create new tactics via the LTAC language, it is possible to introduce domain-specific automation. POET heavily relies on this feature to completely automate the proof generation process.

Once a COQ source file (.v) has been written, it can be compiled into a lower-level representation (generating a .vo file) and finally verified by the standalone proof checker COQCHK. The COQ compiler, COQCHK, and their dependencies, form POET's entire TCB.

Prosa [40], the schedulability analysis framework underlying POET's certificates, uses COQ and its popular extension SSREFLECT [37]. Starting from classic real-time systems concepts, such as *job*, *task*, *processor*, and *arrival curve*, the contributors of PROSA both mechanized classical results (e.g., the optimality of the *earliest-deadline-first* scheduling policy) and developed new theories (e.g., [10, 22, 24]).

For our purposes, the most relevant theory in PROSA is *abstract RTA* (ARTA) [8], which formalizes the well-known busy-window principle to derive a generic RTA applicable to different types of workloads, scheduling policies, and preemption models. ARTA has been instantiated for two scheduling policies (*earliest-deadline-first*, *fixed-priority*) and four preemption models (*preemptive*, *non-preemptive*, *limited-preemptive*, *floating non-preemptive*) in every possible combination, yielding eight different fully verified RTAs [8].

Given that all proofs have been mechanized in PROSA, a high degree of trust may be placed upon the correctness of ARTA and its instantiations. However, a mechanized RTA theory, much like its traditional pen-and-paper counterpart, only describes and justifies *under which conditions* a claimed response-time bound is valid, but it is not, per se, an executable program that can yield numerical results given a concrete task set. Rather, the main ARTA proof follows an axiomatic approach by treating the scheduler, the tasks, and the claimed response-time bounds as abstract variables on which a number of assumptions are made. The abstract result is then derived from these assumptions.

The key idea at the heart of POET is that, by providing instantiations for all variables (i.e., by assigning concrete values), along with a proof that all of ARTA's assumptions are satisfied, ARTA can be put to practical use to verify precomputed response-time bounds. POET thus inherits the system model from ARTA, which we summarize next.

System model. ARTA assumes a discrete time model, where $\varepsilon \triangleq 1$ represents the smallest, indivisible unit of time (e.g., a processor cycle). The system is comprised of a set of n independent tasks $\tau = \{\tau_1, \dots, \tau_n\}$ scheduled on a uniprocessor. Each task τ_i is characterized by its *worst-case execution time* C_i , its *relative deadline* D_i , and an *arrival bound* $\alpha_i(\Delta)$ that upper-bounds the number of new *jobs* (i.e., task activations) that arrive in any interval of length Δ . The two considered preemption models are expressed through each task's *run-to-completion threshold* RCT_i , where $RCT_i = \varepsilon$ in the case of fully non-preemptive tasks, and $RCT_i = C_i$ in the case of fully preemptive tasks. In the case of fixed-priority scheduling, each task also has a fixed priority π_i . As usual in schedulability analysis, any scheduling overheads are presumed to be negligible or already integrated into each task's cost C_i .

Over time, each task τ_i produces an infinite sequence of jobs $\{J_{i,0}, J_{i,1}, \dots\}$. We let \mathbb{J} denote the set of all jobs of all tasks. Each job $J_{i,j} \in \mathbb{J}$ has an arrival time $a_{i,j}$, an *execution time* $c_{i,j} \leq C_i$, and an *absolute deadline* $d_{i,j} = a_{i,j} + D_i$. The number of job arrivals in any interval is constrained by the arrival bound: $\forall \tau_i, \forall t, \forall \Delta, |\{J_{i,j} \mid t \leq a_{i,j} < t + \Delta\}| \leq \alpha_i(\Delta)$.

Finally, for ease of reference, the *arrival sequence* $a(t) \triangleq \{J_{i,j} \mid \forall i, j : a_{i,j} = t\}$ is a function that maps each instant t to the (possibly empty) set of jobs released at t . The arrival sequence and the jobs it contains are the only non-instantiated variables in POET's certificates, meaning that the response-time bounds are proven for all possible arrival sequences respecting the arrival curve and worst-case execution time (WCET) of each task.

Response-time bound. By design, ARTA is independent of specific scheduling policies and preemption models. This is achieved by formulating the RTA problem in an abstract way that captures the essential relationships between the task set, the scheduling policy, the preemption model, and the worst-case response time of a task under analysis [8]. We omit these general technical details here and instead focus on the specific cases relevant to POET.

Intuitively speaking, ARTA analyzes points in the schedule at which the system is *quiet*, which means that no potentially interfering workload is pending (w.r.t. the task under analysis τ_i). A job's *busy window* is the interval between the two closest quiet times enclosing both the job's arrival and completion (by definition, no quiet time occurs while the job is pending). The core of ARTA revolves around a worst-case analysis of the busy window of an arbitrary job $J_{i,j}$ of the task under analysis τ_i . As $J_{i,j}$'s busy window ends only when $J_{i,j}$ completes, a finite busy window implies a response-time bound.

To this end, an ARTA instantiation must provide (and prove correct) two essential inputs. First, there must exist a finite bound L on the maximum busy-window length of any job of τ_i , as otherwise the busy-window principle is not applicable. Second, ARTA requires a policy-specific *interference bound function* $IBF_i(A, \Delta)$ to be defined, with the following semantics [8]: if a job $J_{i,j}$ is released A time units after the beginning of its busy window (where $A \geq 0$, i.e., A is $J_{i,j}$'s relative release offset), then $IBF_i(A, \Delta)$ is an upper bound on the maximum amount of potentially interfering workload arriving in any interval of length Δ .

Given a task under analysis τ_i with a maximum busy-window length L and a policy-specific $IBF_i(A, \Delta)$, ARTA proceeds by considering every possible arrival offset $A \in [0, L)$. For each such offset A , a solution F to the fixed-point equation

$$A + F = RCT_i + IBF_i(A, A + F) \quad (1)$$

is required to exist. Intuitively, F is the maximum time it takes for a job with arrival offset A to receive sufficient service to certainly reach the run-to-completion threshold RCT_i , at which time, by definition, it cannot be preempted anymore. Therefore, the response time of a job with arrival offset A can be bounded by $F + (C_i - RCT_i)$. Task τ_i 's overall response-time bound R is given by the maximum F encountered solving Equation (1) for each offset $A \in [0, L)$.

Search space. Practically speaking, it is impossible to check every possible arrival offset $A \in [0, L)$ since, for task sets specified with nanosecond resolution, L easily reaches magnitudes in the order of trillions. Fortunately, it is not necessary to check every single point in the interval, since the only varying term in Equation (1) is $IBF_i(A, A + F)$. Hence, ARTA defines the *search space* of the task under analysis τ_i as

$$\mathcal{A}_i \triangleq \{0\} \cup \{A \in (0, L) \mid \exists \Delta, IBF_i(A - \varepsilon, \Delta) \neq IBF_i(A, \Delta)\} \quad (2)$$

■ **Listing 1** An example POET input file (in YAML format) specifying two tasks scheduled under the fully-preemptive fixed-priority policy. The lower-priority task (with ID 2) is periodic (Line 10) and has a deadline exceeding the period (Line 11). The higher-priority task (with ID 1) is a sporadic task characterized by an arrival-curve prefix (Line 5), which is specified by the length of the prefix (220) and the list of steps of the curve in the prefix: $\alpha_1(\Delta) = 1$ for $\Delta \in [1, 105)$ and $\alpha_1(\Delta) = 2$ for $\Delta \in [105, 220)$. The initial value $\alpha_1(0) = 0$ can be omitted by convention.

```

1  scheduling policy: fixed-priority
2  preemption model: fully-preemptive
3  - id: 1
4    worst-case execution time: 50
5    arrival curve: [220, [[1,1],[105,2]]]
6    deadline: 100
7    priority: 2
8  - id: 2
9    worst-case execution time: 10
10   period: 30
11   deadline: 100
12   priority: 1

```

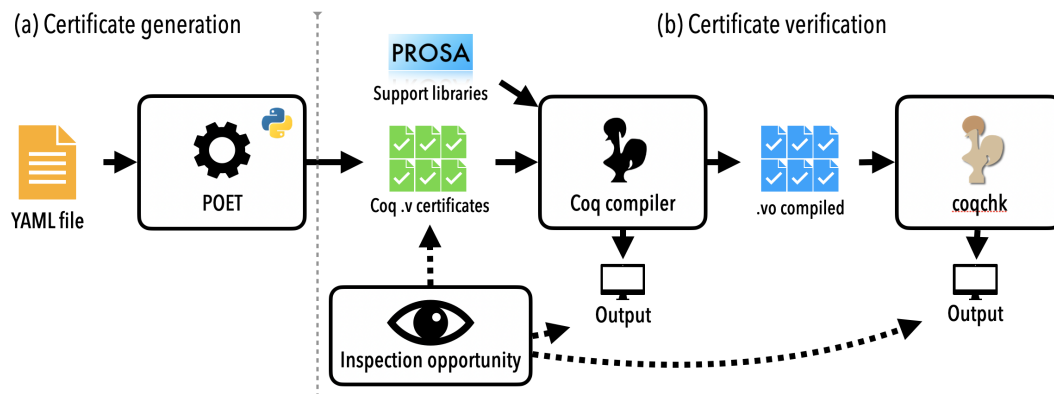
The search space \mathcal{A}_i restricts the analysis to such offsets A at which the interference bound function IBF_i changes in value, hence excluding all the plateaus of the function. In practice, this restriction results in a sparse search space, which is key to obtaining a practical runtime.

4 POET: Design and Workflow

POET is the first foundational RTA tool: it generates formal COQ proofs, i.e., *certificates*, establishing the correctness of its computed response-time bounds. As a proof of concept, it supports four concrete ARTA instantiations. Specifically, it supports real-time workloads comprised of recurrent, independent, arbitrary-deadline tasks under *fixed-priority* (FP) and *earliest-deadline-first* (EDF) scheduling with both the *fully-preemptive* (FP) and *fully non-preemptive* (NP) preemption models. Task activations may be *periodic* or *sporadic*, or defined by an arbitrary arrival curve. Due to POET’s novel and unique combination of objectives and features, its design and implementation posed some unusual challenges. We begin with an overview of these challenges, key design decisions, and the resulting workflow, and then discuss central issues in more detail in the subsequent sections.

The first major requirement is **usability**. For foundational RTA to be successful it must have a low barrier to adoption, which means that POET must remain accessible to a general audience without any expertise in formal verification. In particular, users must *not* be expected to be proficient in authoring COQ proofs. We therefore designed POET to require only a human-readable YAML file specifying the task set, the scheduling policy, and the preemption model. Listing 1 shows an example. From this simple input, which does not differ significantly from that of other, unverified RTA tools, POET generates verified RTA results fully automatically, without any human interaction or need for verification expertise.

The second major requirement is **transparency**. Since the process of calculating the response-time bounds, generating formal proofs of their correctness, and then machine-checking the proofs is entirely automated, it is essential to prevent silent failures. It is thus necessary to ensure explainability of the results, that is, it must be readily possible



■ **Figure 1** The POET workflow: (a) the YAML input file is used by POET to instantiate one certificate per task; (b) each certificate is compiled and verified using Coq. The procedure is fully automated, but open to human inspection at each step.

for a human to scrutinize in detail and fully understand the certificates. The certificates hence must be optimized for readability and any generated artifacts must be limited to a comprehensible size and scope. Moreover, there are ways in which machine-checked proofs might still not justify the intended conclusions. This issue is further discussed in Section 5.

A major challenge arises from POET’s support of **arbitrary arrival curves**. By supporting not only the classic periodic and sporadic task models, but also any workload that can be characterized with arbitrary arrival curves, POET gains broad applicability to real-world workloads (e.g., bursty workloads, workloads with jitter, irregular arrival processes, trace-based empirical arrival bounds, etc.). However, as defined in Section 3, an arrival curve is a function on an infinite time domain. In the input to POET, such an arrival curve necessarily needs to be truncated to a finite *arrival-curve prefix* (e.g., Line 5 in Listing 1). Unfortunately, in the context of a COQ proof, representing and computing with arrival-curve prefixes is far more tricky than one might initially suspect. In Section 6, we discuss the strategies adopted in POET and the resulting accuracy *vs.* efficiency trade-offs.

Last but not least, as already mentioned in Section 2, the **computational efficiency** of the certification process proved to be a major hurdle in the implementation of POET, ultimately affecting its design. Specifically, numerically large computations are infeasible with the unary representation of numbers employed in SSREFLECT, and hence by extension also PROSA and ARTA. However, to be practical, POET’s certificates need to support large-magnitude parameters because in real-world task sets periods, costs, and deadlines are often expressed in nanoseconds or processor cycles. We discuss the strategies that we adopted in POET to realize nonetheless acceptably fast calculations in Section 7.

4.1 Implementation and Workflow

The usability and transparency considerations lead to the workflow illustrated in Figure 1. The entire procedure comprises two phases, namely (a) the generation of certificates starting from an input file provided by the user, and (b) the COQ compilation and proof-checking.

The input file contains all necessary information about the task set, the scheduling policy, and the preemption model (recall Listing 1). Given an input file, POET produces one certificate (.v file) per task by instantiating a template specific to the given scheduling policy and preemption model. During this phase, COQ itself is not involved in any way.

POET is implemented as a simple PYTHON tool and acts for the most part as a straightforward template engine. Certificate generation begins with a *template* similar to Listing 2 (discussed in Section 4.2 below) that, instead of containing specific data (e.g., task declarations, the value of L and R , etc.), has a uniquely named placeholder at each relevant position. POET simply replaces each placeholder with concrete data taken either from the YAML input file (e.g., task parameters) or computed on the fly (e.g., the value of R), and for each task stores the result as a new `.v` file containing the final proof script.

Using the same principle, POET also generates *data-dependent proof scripts*. For instance, each certificate needs a proof script showing that each point of the input-dependent search space can be paired with a fixed-point solution of Equation (1). However, the generation of data-dependent proof scripts is kept as simple as possible by implementing as much case-analysis logic as possible as generic LTAC tactics that automate the proof. Overall, the lion’s share of the development effort has gone into the certificate templates and the supporting COQ libraries, whereas the PYTHON component is relatively small and mundane in comparison.

Once the certificates are in place, POET triggers first the COQ compiler, which will produce compiled files (`.vo`) containing low-level *proof terms*, and finally COQCHK, which verifies the proof terms. By design, phases (a) and (b) are independent and performed by different tools (POET and COQ). In particular, the second phase – compilation and verification of the certificates – may be performed repeatedly and on different machines.

The generated certificates establish the *soundness* of the bounds computed by POET, but do not make any *tightness* claims. A bug in POET could thus lead to the rejection of feasible task sets, or might cause pessimism in the claimed bounds, but it cannot result in incorrect bounds, as COQCHK will reject any flawed certificates claiming unsafe bounds.

We chose to implement POET in PYTHON because of our familiarity with the language and its convenient facilities for basic file handling and text manipulation. By design, a foundational RTA does not have to be verified itself, so the fact that PYTHON is notoriously difficult to verify (being a dynamically typed scripting language) did not hinder us.

4.2 The Structure of a Certificate

Listing 2 shows the certificate generated by POET for the second task of the input file of Listing 1. The certificate starts by importing the correct support library in Line 1, which here is the fully-preemptive FP instantiation of ARTA in the PROSA open-source framework.

The actual task-specific content starts in Line 8, which opens a scope for the subsequent declarations. The certificate continues with straightforward declarations of the tasks in the task set (Lines 12–23), the task set (Line 25), the identity of the task under analysis (line 26), and the analysis results (Lines 28 and 29). Recall that L is a bound on the maximum busy-window length and R is the claimed response-time bound. Both L and R are computed by POET; the goal of the certificate is to prove that R is indeed an upper bound on the actual response time of any job of the task under analysis.

The first four lemmas proven by POET are auxiliary facts documenting that the user-provided parameters are valid (e.g., periods and task costs are positive, arrival curves are monotonic, etc.). Listing 2 shows one such example lemma in Lines 36–37. The next lemma in Lines 87 and 88 establishes that the bound on the maximum-busy interval calculated by POET is correct. Specifically, ARTA [8] requires L to be the solution to a fixed-point equation that depends on the scheduling policy and preemption model.

Next, in lines 96-101, the arrival sequence is introduced. Note that no concrete definition is given: the certificate treats the arrival sequence as a *variable*, i.e., the result is proven *for all possible arrival sequences* respecting the hypotheses H_1, \dots, H_5 stated in lines 97-101. The

■ **Listing 2** A simplified version of the certificate generated by POET for the second task of the workload given in Listing 1. All proofs and comments, and some intermediate lemmas and auxiliary definitions, have been omitted for brevity, and some details (e.g., lemma names) have been simplified. The notation $[::a;b;c]$ is SSREFLECT syntax for a list comprised of three elements a , b , and c .

```

1  Require prosa.results.fixed_priority.rta.fully_preemptive.
   [...]
8  Section Certificate.
   [...]
12 Let tsk01 := { |
13   task_id := 1;
14   task_cost := 50;
15   task_deadline := 100;
16   task_arrival := Curve_Prefix (220, [(1, 1);(105, 2)]);
17   task_priority := 2 |}.
18 Let tsk02 := { |
19   task_id := 2;
20   task_cost := 10;
21   task_deadline := 100;
22   task_arrival := Periodic 30;
23   task_priority := 1 |}.
24
25 Let ts := [::tsk01;tsk02].
26 Let tsk := tsk02.
27
28 Let L := 80.
29 Let R := 60.
   [...]
36 Lemma valid_arrival_curve :
37    $\forall \text{task, task} \in \text{ts} \rightarrow \text{max\_arrivals tsk } 0 = 0 \wedge \text{monotone leq (max\_arrivals tsk)}$ .
   [...]
87 Lemma L_fixed_point :
88   total_hep_rbf ts tsk L = L.
   [...]
96 Variable arr_seq : arrival_sequence Job.
97 Hypothesis H1 : arrival_sequence_uniq arr_seq.
98 Hypothesis H2 : all_jobs_from_taskset arr_seq ts.
99 Hypothesis H3 : arrivals_valid_job_costs arr_seq.
100 Hypothesis H4 : consistent_arrival_times arr_seq.
101 Hypothesis H5 : respects_max_arrivals arr_seq ts.
   [...]
106 Definition sched := uni_schedule arr_seq.
   [...]
126 Definition F_solutions := [::60;40;20].
127
128 Lemma R_is_maximum :
129    $\forall A, \text{is\_in\_search\_space tsk L } A \rightarrow$ 
130    $\exists F, \text{task\_rbf tsk (A + 1) + total\_ohep\_rbf ts tsk (A + F) } \leq A + F \wedge F \leq R$ .
   [...]
143 Theorem R_bounds_response_time :
144   task_response_time_bound arr_seq sched tsk R.
145 Proof.
   [...]
148   apply arta_response_time_bound_fp_fp.
   [...]
167 Qed.
   [...]
172 Corollary R_respects_deadlines :
173   task_response_time_bound arr_seq sched tsk R  $\wedge R \leq \text{task\_deadline tsk}$ .
   [...]
181 End Certificate.
182
183 Section AssumptionLessExample.
184   Definition concrete_arr_seq := concrete_arrival_sequence ts generate_jobs_at.
   [...]
186   Theorem R_bounds_response_time_concrete:
187     task_response_time_bound concrete_arr_seq (sched concrete_arr_seq) tsk R.
   [...]
200 End AssumptionLessExample.

```

hypotheses state that the arrival sequence contains each job only once (H_1) and only contains jobs of tasks in the task set (H_2). Further, each job present in the sequence has a positive cost that does not exceed its task’s WCET (H_3) and its position in the arrival sequence is consistent with its arrival time (H_4). Finally, for each task in the task set, the cumulative number of job arrivals is bounded by the arrival curve of the task (H_5). Importantly, the definitions of H_1, \dots, H_5 reside in PROSA, from which the certificate derives its semantics.

In Lines 126–130, the certificate shows that Equation (1) for fully-preemptive FP scheduling, as specified by the user in Listing 1, holds for every A in the search space. To this end, a sequence of solutions – for each A , the corresponding F in Equation (1) – is provided by POET in Line 126. The data-dependent proof script (here omitted) performs a case analysis for each A in the search space and presents the corresponding solution to the proof engine.

The overall correctness claim is stated in Lines 143–167, which states that R is indeed a response-time bound for the task under analysis. Crucially, to prove this fact, POET applies the main ARTA theorem for fully-preemptive FP scheduling [8] in Line 148.

Again, the definition of the predicate `task_response_time_bound` used in Line 144 resides in PROSA and is not specific to (nor in any way influenced by) POET. This is an important point: any formally verified result is useful only as far as the *specification* that is shown to hold is semantically meaningful. POET therefore does *not* provide its own semantic specification. Instead, it delegates the semantic modeling of real-time scheduling entirely to prior work and reuses an established specification, namely PROSA [11].

Finally, the corollary in Lines 172–173 explicitly confirms the desired result: the deadline of the task under analysis is always respected. As a safeguard against accidental omission of the main proof, the corollary repeats the claim from Line 144, which ensures that the corollary cannot be proven without first completing the main proof.

The main certificate ends in Line 181, which closes the scope of the non-instantiated variable declared in Line 96, namely the arrival sequence `arr_seq`. The remaining section in Lines 183–200 is another safeguard: it repeats the main theorem of the certificate (Line 144) in an *assumption-less* context (i.e., without `arr_seq` in scope), for a concretely defined arrival sequence. The purpose of this section is to demonstrate that the hypotheses H_1, \dots, H_5 stated in lines 97–101 are free of contradictions, as we explain in more detail in Section 5.

Overall, the generated certificate resembles the flow of traditional pen-and-paper reasoning and is sufficiently short to be inspected manually (200 lines in total). In particular, all proofs, which are responsible for the bulk of the total line count, can be safely skipped since they are verified by COQCHK. Importantly, the certificate is intentionally readable and simple, in the sense that only very little experience with GALLINA is required to make sense of it.

5 Trustworthiness of the Procedure

By design, POET itself is not part of the TCB. In particular, any critical bugs in the tool that result in incorrect response-time bounds will not go unnoticed by COQ (which is the TCB) – any attempt to compile and check such corrupted certificates will result in obvious and unmistakable errors. In this section, we thus focus on issues that could lead to *silent failures*: how could auto-generated certificates still be misleading even if they are accepted as valid by the proof checker?

Incomplete proofs. Since POET is not assumed to be correct, it might conceivably generate an incomplete (or, in the extreme case, even completely empty) certificate that COQ would then successfully check: a cleanly truncated (or empty) file does not contain any incorrect proofs and hence does not give the proof checker any reason to reject it. COQ further lets the user *admit* theorems, i.e., to accept them as valid without proof, treating them as axioms.

Though these edge cases could easily be programmatically detected by POET itself, doing so would implicitly turn the tool into a trusted component. For the same reason, POET cannot be in charge of reporting the results of the verification attempt to the user. After the certificates have been generated, POET must not intervene in any way. Therefore, any action that needs to be executed after the creation of the certificates *is handled entirely in the COQ environment*. This includes printing of the results, which is done directly in COQ, and checking that no theorems have been *admitted* (using COQCHK).

To entirely eliminate any need to trust POET, the certificates and the output they generate are designed to be human-readable. Although POET is fully automated, the process is transparent and open to human supervision. In particular, a user may (i) observe the outputs of the COQ compiler and COQCHK to assess whether they succeeded, (ii) ensure that the input and the generated certificates match in terms of task set, task parameters, scheduling policy, and preemption model (e.g., the file in Listing 1 with Lines 1–25 of Listing 2), and (iii) to ensure that the response-time claim is included in the verified certificate (e.g., check whether the corollary in line 172 of Listing 2 is present).

Finally, the certificates themselves are completely transparent, too: it is easy for a user with COQ expertise to scrutinize (i.e., interactively step through) the complete list of proof steps that lead to the response-time bound. While we do not expect the typical user to inspect certificates that closely, striving for readability and making the certificates generally inviting increases their quality and ensures explainability of the results. Furthermore, it renders them suitable as *evidence of temporal correctness* since a third-party auditor or certification authority can independently study and dissect POET’s certificates down to their fundamental definitions (as provided by PROSA) and the axioms of COQ’s logic.

Although human supervision is always welcome, only step (i) – carefully reading the analysis outputs, a degree of supervision that *any* RTA tool requires – is essential. The rather hypothetical errors caught by steps (ii) and (iii) are unlikely to manifest accidentally, and are thus less relevant in practice. In any case, none of the steps is onerous, and each can be easily completed without in-depth verification expertise.

Contradictions. A second, more subtle type of error is related to the possibility of contradictory hypotheses in axiomatic theories such as ARTA: conclusions reached in a sound manner from contradictory premises may still be incorrect. This potential pitfall has been previously described in-depth by Cerqueira et al. [11]. As it is generally not possible for COQ to detect contradictory hypotheses automatically, it is necessary to establish the absence of contradictions on a case-by-case basis. Concretely, this requires demonstrating that it is possible to instantiate all variables such that all hypotheses are respected simultaneously.

POET’s certificates generalize over only one variable, namely the arrival sequence `arr_seq` (Lines 96–101 of Listing 2). Recall from Section 3 that the arrival sequence yields, for any given time t , the sequence of jobs (each with a specific cost) that arrive at time t . The arrival sequence is intentionally left uninstantiated in the certificate’s main section (Lines 8–181) as the response-time bound must hold for *any* possible pattern of arrivals that respects the workload constraints (e.g., that jobs arrive periodically).

Nonetheless, to formally prove the absence of contradictions, POET generates for each task, in addition to the general response-time bound (Lines 143–167), a *second* version of the theorem in a separate section devoid of *any* variables or hypotheses (Lines 183–200 in Listing 2). In this *assumption-less example*, the response-time bound is proven once more to hold by applying the general result to a concrete arrival sequence with fixed job costs, which establishes that the general result does not make any contradictory assumptions. Technically,

in CoQ terminology, the type of the main result is shown to be inhabited. At this point, as no variable or hypothesis is in scope, only a soundness flaw in COQCHK (which is part of the TCB) could allow an incorrect result to be proved.

To establish that the main result’s hypotheses are contradiction-free, it actually suffices to instantiate *any* valid arrival sequence, including pathological ones in which no job ever arrives. However, from a user’s perspective, it is more confidence-inspiring to use a nontrivial workload. POET therefore generates an arrival sequence that, at any time, greedily maximizes the number of arrivals of each task and their costs while respecting all workload constraints.

To summarize, neither POET nor the underlying RTA are part of the TCB, and hence do not have to be trusted, because **(1)** the results of the analysis are communicated from within the CoQ environment (and not by POET itself), **(2)** any program defects in POET or flaws in the underlying RTA that lead to incorrect response-time bounds will be caught by CoQ (the proof checker would fail), **(3)** the remote chance of mismatching assumptions in the certificate (e.g., wrong task parameters or a change in scheduling policy) or certificate truncation are immediately obvious to lightweight human supervision since POET’s certificates are short and designed for readability, and **(4)** the risk of contradictory hypotheses is mitigated by the inclusion of an assumption-less example that exercises the general bound in a verified manner.

6 Supporting Arbitrary Arrival Curves

A major challenge affecting POET is the *representation* and *extrapolation* of arbitrary arrival-curve prefixes, and the fast computation of the ARTA search space \mathcal{A}_i (recall Equation (2)), which is closely linked to it. To reiterate, support for arbitrary arrival curves in POET is desirable due to the flexibility it affords, enabling support for a wide range of real-world arrival processes that are neither perfectly periodic nor described well with a single, scalar minimum inter-arrival time. Fortunately, ARTA already supports arbitrary arrival curves [8], but only at an ideal mathematical, non-instantiated level (i.e., ARTA’s arrival curves are functions on an infinite domain, and not some finite representation thereof). In contrast, POET needs to compactly represent and efficiently compute with *concretely defined* arrival-curve *prefixes*.

As sketched in Listing 1, POET expects each arrival-curve prefix to be compactly expressed with a *horizon* h and a sparse sequence of m *steps* s_1, \dots, s_m . The horizon h defines the length of the prefix (i.e., the maximum Δ covered). A step $s_k = (\Delta_k, c_k)$ indicates that the arrival curve α_i “takes a step” at Δ_k : $\alpha_i(\Delta_k - \varepsilon) < c_k$ and $\alpha_i(\Delta_k) = c_k$.

Given an arrival-curve prefix with $h < L$, it becomes necessary to **extrapolate** during analysis. Let α^* denote the arrival curve extrapolated from the finite prefix, and let $s(t) \triangleq \max(\{0\} \cup \{c_k \mid 1 \leq k \leq m \wedge \Delta_k \leq t\})$ denote the result of looking up the number of arrivals in an interval of length $t \leq h$ in the given sequence of steps. Then:

$$\alpha^*(\Delta) \triangleq \lfloor \Delta/h \rfloor \cdot s(h) + s(\Delta \bmod h) \quad (3)$$

This choice represents a trade-off between the speed of extrapolation and analysis precision. Equation (3) does not guarantee an optimal extrapolation – depending on the given prefix, an extrapolation exploiting the arrival curve function’s sub-additivity may be less pessimistic. Nonetheless, it provides the key advantages of being simple and fast to compute. In fact, each time the proof engine has to evaluate Equation (1), several arrival curves are evaluated. Since traversing the ARTA search space is one of the most expensive steps of certificate checking (as we show in Section 8), it is desirable to keep Equation (3) as simple as possible.

As shown in Listing 1, POET has built-in support for **periodic and sporadic tasks**. However, as a task τ_i with period or minimum inter-arrival time T_i can be easily described with an arrival curve $\alpha_i(\Delta) \triangleq \lceil \Delta/T_i \rceil$, POET’s certificates work *exclusively* with arrival-curve prefixes. Specifically, a task with period or minimum inter-arrival time T_i is internally

represented with an arrival-curve prefix with horizon $h = T_i$ and a single step $s_1 = (1, 1)$. This conversion is lossless: in this important special case, Equation (3) does not introduce any pessimism since $\alpha^*(\Delta) = \lfloor \Delta/h \rfloor \cdot s(h) + s(\Delta \bmod h) = \lfloor \Delta/T_i \rfloor \cdot 1 + s(\Delta \bmod T_i)$, which is equal to $\frac{\Delta}{T_i} + 0 = \alpha_i(\Delta)$ if T_i divides Δ and equal to $\lfloor \Delta/T_i \rfloor + 1 = \lceil \Delta/T_i \rceil = \alpha_i(\Delta)$ otherwise.

Critically, the automatic conversion to arrival-curve prefixes is performed in POET's COQ libraries, and not in the PYTHON part, hence introducing no verification gap whatsoever while still freeing the user from having to think about this detail.

As already mentioned, the **computation of the search space** is the primary bottleneck of POET. Recall from Equation (2) that ARTA defines the search space \mathcal{A}_i for each task τ_i as the set of points at which the interference bound function IBF_i changes in value. Although ARTA correctly anticipates that a sparse search space will be necessary for practical use [8], it does *not* provide a way to compute it. However, the search space *must be computed by the proof engine* to validate POET's fixed-point solutions (e.g., in Line 126 of Listing 2).

To this end, we exploit the structure of Equation (3). Since the function IBF_i depends on the concrete scheduling policy, we discuss FP and EDF scheduling in sequence.

In the case of FP scheduling, Equation (2) reduces to the set of points at which the *extrapolated* arrival curve of the task under analysis τ_i changes in value in the interval $[0, L]$: $\mathcal{A}_i^{FP} = \{A \mid A < L \wedge \alpha_i^*(A) \neq \alpha_i^*(A + \varepsilon)\}$ [8]. As shown in POET's support libraries (which are checked as part of every certificate), \mathcal{A}_i^{FP} can be easily over-approximated by repeatedly concatenating task τ_i 's list of steps s_1, \dots, s_m .

$$\mathcal{A}_i^{FP} \subseteq \{lh + s_1, \dots, lh + s_m \mid 0 \leq l \leq \lfloor L/h \rfloor\} \quad (4)$$

Equation (4) defines a superset of the actual search space because some of the included points may exceed L (e.g., if $\lfloor L/h \rfloor \cdot h + s_m > L$). This does not affect the analysis's correctness as there is no harm in evaluating superfluous offsets [8].

In the case of EDF, the search space is far more complex since it involves every task in the task set: $\mathcal{A}_i^{EDF} = \{A \mid A < L \wedge \exists \tau_j \in \tau, \alpha_i^*(A + D_i - D_j) \neq \alpha_i^*(A + \varepsilon + D_i - D_j)\}$ [8]. For ease of computation, we decompose \mathcal{A}_i^{EDF} on a per-task basis such that $\mathcal{A}_i^{EDF} = \bigcup_{\tau_j \in \tau} \mathcal{A}_{i,j}^{EDF}$, where $\mathcal{A}_{i,j}^{EDF} = \{A \mid A < L \wedge \alpha_i^*(A + D_i - D_j) \neq \alpha_i^*(A + \varepsilon + D_i - D_j)\}$. Note that $D_i - D_j$ is a constant on both sides of the defining inequality. If removed from both sides, we obtain exactly \mathcal{A}_i^{FP} ; that is, $\mathcal{A}_{i,j}^{EDF}$ can be thought of as \mathcal{A}_i^{FP} *shifted* by $D_j - D_i$ time units. Analogously to Equation (4), POET thus computes $\mathcal{A}_{i,j}^{EDF}$ as

$$\mathcal{A}_{i,j}^{EDF} \subseteq \{lh + s_1 + D_j - D_i, \dots, lh + s_m + D_j - D_i\} \quad (5)$$

for $0 \leq l \leq \lfloor L/h \rfloor$. Again, this is an over-approximation; in particular, any negative points are simply ignored. Finally, Equations (4) and (5) demonstrate the key benefit of the fast extrapolation rule in Equation (3): the search space can be over-approximated easily and relatively quickly.

7 Scalability of the Certification Procedure

One of the major challenges we encountered during the development of POET is the poor computational performance of SSREFLECT's standard number representation. Without a working solution, this seemingly small detail can jeopardize the entire idea of foundational RTA – for which computation in the proof engine is essential.

POET's certificates depend on PROSA, and therefore implicitly on SSREFLECT, which uses a *unary* representation of natural numbers. The use of a unary number representation has clear advantages when writing proofs, as it simplifies inductive reasoning and case

Therefore, **(a)** a binary version of each unavoidable computation was implemented and integrated into POET’s COQ support libraries, and **(b)** each such alternative implementation was related to its unary counterpart via an additional corpus of proofs so that the computed results can be *substituted* into the proof. More technically, for the proof engine to accept a rewriting step (i.e., to perform a substitution), the two involved functions must be proven to be extensionally equivalent. To this end, we applied COQEAL [14], a COQ proof framework for changes in data representation.

To compute a unary-numbers function using a binary-numbers counterpart, COQEAL requires proving a so-called *refinement*. Given the sets of unary numbers \mathbb{N}_1 and binary numbers \mathbb{N}_2 , consider a conversion function $\Phi(x) : \mathbb{N}_2 \rightarrow \mathbb{N}_1$ that, given a binary number, yields its unary equivalent. Further, consider a unary-numbers predicate $p_1(x) : \mathbb{N}_1 \rightarrow \mathbb{B}$ and its binary-numbers counterpart $p_2(x) : \mathbb{N}_2 \rightarrow \mathbb{B}$. For the purposes of POET, a refinement can be seen as a proof that, for any binary number x , $p_1(\Phi(x)) = p_2(x)$. Analogous arguments can be made for predicates with multiple parameters and for predicates with higher-order parameter types based on unary numbers, such as lists and tuples over \mathbb{N}_1 . With a refinement in place, a proof step triggering a computation of p_1 can be replaced, similarly as in lemma *Ex2* in Listing 3b, with one involving a computation of p_2 , which can be performed quickly.

In conclusion, for each numeric function computed by COQ when compiling POET’s certificates, **(1)** an equivalent function defined on binary numbers was implemented in the support libraries, and **(2)** a refinement relating the two functions was proven. The total support code related to refinements is roughly 1800 LOC of definitions, proofs, and tactics, representing around 40% of the entire POET support code: enabling binary computations comes with an extra development cost. However, the switch to binary representation dramatically impacts runtime and memory needs and hence is essential.

Case in point, consider once again the task set in Listing 1 (expressed in milliseconds). With the binary representation in place, the total certification time is around five seconds on our testing machine (i.e., slightly slower than before), but stays roughly the same irrespective of whether task parameters are given in microseconds or nanoseconds.

Finally, it is worth emphasizing that COQEAL is *not* part of the TCB since, as a COQ library, it is itself subject to full verification by COQCHK. Therefore, speeding up the computation by translating to a binary encoding does not introduce any verification gaps.

8 Empirical Evaluation

To assess POET’s runtime characteristics, we conducted an empirical evaluation using synthetic workloads, which we generated as follows. For a given number of tasks n and total utilization u , we used the Dirichlet-Rescale algorithm [27] to draw n utilization values u_1, \dots, u_n summing to u . To exercise POET’s versatility, we considered five different types of workloads: **(i)** sporadic workloads as commonly encountered in automotive systems, **(ii)** sporadic workloads with a log-uniform distribution of minimum inter-arrival times, **(iii)** sporadic workloads subject to job bursts, **(iv)** sporadic workloads subject to jitter, and **(v)** sporadic arrivals distributed according to a Poisson distribution. This selection covers a wide range from relatively well-structured arrival processes to less regular ones. Types (i)–(iv) equivalently represent periodic workloads since the sporadic task model generalizes the periodic one. We hence use the terms “inter-arrival time” and “period” interchangeably.

The different arrival bounds were generated as follows. Let τ_i denote the task for which an arrival bound is being generated. In the case of (i), the period T_i was chosen uniformly at random from $P \triangleq \{1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$ ms, a set of periods commonly

encountered in automotive systems [33]. For (ii), T_i was drawn log-uniformly from $[1, 100]$ ms. For (iii), a bursty arrival process was defined by the maximum number of jobs in a burst $b_i = 4$ and the randomly chosen minimum intra-burst inter-arrival time $T_i^{in} \in [1, 100]$ μs and the minimum inter-burst inter-arrival time $T_i \in P$. In the case of (iv), τ_i 's period T_i was chosen from P and the maximum jitter was drawn uniformly at random from $[0.1T_i, 3T_i]$. Lastly, for (v), we considered a Poisson process with a mean arrival rate $r_i \triangleq \frac{1}{T_i}$, where T_i was drawn uniformly at random from P . For any m and Δ , a Poisson process has a non-zero (but rapidly diminishing) probability of yielding m arrivals in an interval of length Δ . To convert this to an arrival curve, we defined $\alpha_i(\Delta)$ such that, for any Δ , the probability of observing more than $\alpha_i(\Delta)$ job releases in an interval of length Δ is less than 10^{-3} .

Next, the generated arrival bounds were encoded in POET's input format. Cases (i) and (ii) are trivial as POET natively supports periodic and sporadic tasks. Cases (iii)–(v) require the arrival process to be expressed as a finite arrival-curve prefix, as discussed in Section 6. Recall that task τ_i 's finite arrival-curve prefix consists of two parts: a horizon h_i and a sequence of m_i steps. In the case of (iii), bursty arrivals, the arrival bound can be encoded in a lossless manner by setting $h_i \triangleq T_i$ and defining $m_i = b_i$ equidistant steps with a separation of T_i^{in} . For both (iv) and (v), we simply truncated the arrival curve by choosing the maximum h_i containing at most $m_i = k$ steps, where k differed across experiments.

Finally, the relative deadline D_i was drawn from $[0.3T_i, 3T_i]$, and the task's WCET C_i was set to $\lceil u_i \cdot p_i \rceil$ in cases (i) and (ii), and to $\lim_{t \rightarrow \infty} \lceil (u_i \cdot t) / \alpha(t) \rceil$ in cases (iii)–(v). All parameters were given to POET in nanosecond resolution (i.e., $\varepsilon = 1$ ns).

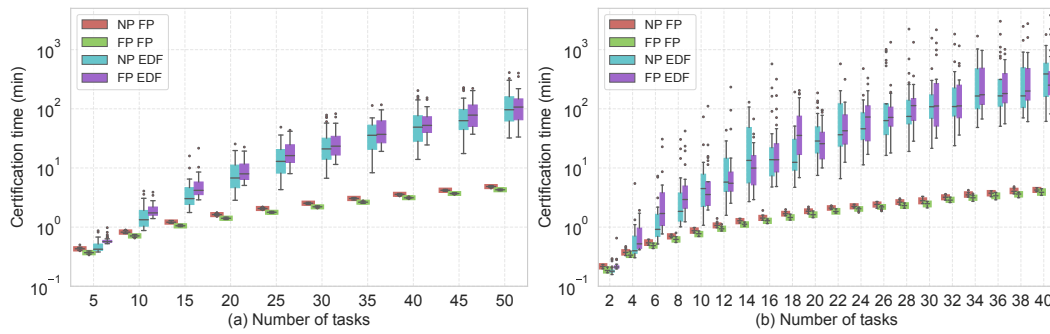
For each considered n , we let the total utilization u range from 0.5 to 0.9 in steps of 0.1. We evaluated each supported policy: fully-preemptive FP (FP-FP), fully non-preemptive FP (NP-FP), fully-preemptive EDF (FP-EDF), and fully non-preemptive EDF (NP-EDF). Under FP scheduling, tasks were assigned rate-monotonic priorities. We ran all experiments on a Linux host with two 2.50 GHz Intel Xeon "Platinum 8180" processors and 394 GiB RAM.

In the **first experiment**, we focused on classic sporadic tasks (i.e., type-(i) tasks). We varied the number of tasks n from 5 to 50 in steps of 5 and, for each combination of scheduling and preemption policy, generated 10 task sets for each cardinality ($\times 10$) and utilization ($\times 5$), resulting in 500 task sets per policy and 2000 in total. For each workload, we measured the end-to-end runtime of the entire workflow depicted in Figure 1 (including POET, the CoQ compiler, and COQCHK) while running *sequentially* on a single core.

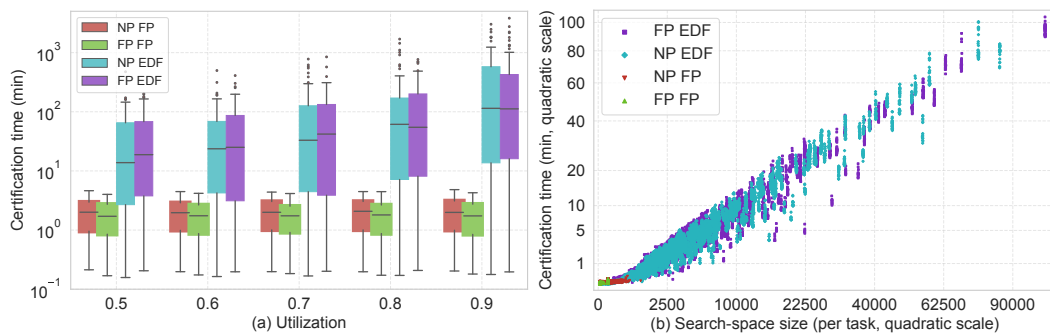
As can be seen in Figure 2a, these workloads can be easily certified by POET. There is a clear difference between EDF and FP analyses, but no major difference between the two preemption policies. Small workloads ($n \leq 10$) are typically solved in seconds for both EDF and FP; larger workloads take significantly more time, with FP analyses being clearly faster. For $n = 50$, the mean runtime per task set was 4.2 minutes under FP-FP, 4.8 minutes under NP-FP, 119 minutes under FP-EDF, and 122 minutes under NP-EDF. Overall, across all cardinalities, the mean runtime per task set was 2.1 minutes under FP-FP, 2.4 minutes under NP-FP, 38 minutes under FP-EDF, and 35 minutes under NP-EDF.

The growth in certification time dependent on n evident in Figure 2a is due to two factors: adding a task obviously increases the number of certificates that must be generated and checked, while also increasing the complexity of the certificates of all prior tasks.

In the **second experiment**, we challenged POET with more demanding workloads by randomly choosing each task's arrival model among types (i)–(v). The resulting workloads are hence less structured and more complex, characterized by many nontrivial arrival curves.



■ **Figure 2** The end-to-end runtime of POET and CoQ *vs.* the number of tasks in **(a)** for classic sporadic tasks (type (i)) and **(b)** mixed workloads (types (i)–(v)). Boxes range from the first to the third quartile; whiskers extend to 1.5 times the inter-quartile range (IQR).

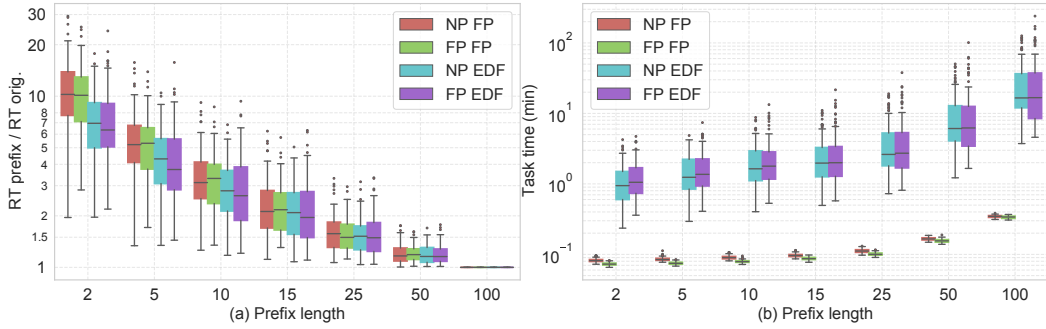


■ **Figure 3** Based on the same data set as shown in Figure 2b, **(a)** the total certification runtime *vs.* total utilization and **(b)** the *single-task* certification runtime *vs.* the task’s search-space size; note the quadratic scale of both axes in inset (b).

In this experiment, we varied the number of tasks n from 2 to 40 in steps of 2. For each combination of scheduling and preemption policy, we generated 5 task sets for each cardinality ($\times 20$) and utilization ($\times 5$), resulting in 500 task sets per policy and 2000 in total. The maximum length of arrival-curve prefixes was fixed to $k = 50$. As in the first experiment, we measured the end-to-end runtime of the entire workflow running sequentially on one core.

The results of the second experiment (depicted in Figure 2b) exhibit similar trends as in Figure 2a, but with an (expected) steeper growth in certification time. For $n = 40$, the mean runtime per task set was 3.9 minutes under FP-FP, 4.2 minutes under NP-FP, 568 minutes under FP-EDF, and 466 minutes under NP-EDF. Overall, across all cardinalities, the mean runtime per task set was 1.8 minutes under FP-FP, 2.1 minutes under NP-FP, 140 minutes under FP-EDF, and 145 minutes under NP-EDF. Clearly, POET’s scalability is substantially reduced for such challenging workloads, but it still manages to certify workloads of nontrivial size. Notably, the most time-consuming aspect of POET, certificate checking, can be easily parallelized since task certificates are independent of each other. High runtimes for large n can thus be alleviated by letting POET run COQCHK in parallel on up to n cores.

Next, to better understand the primary drivers of runtime growth, we plotted the data collected in the second experiment in two additional ways as shown in Figure 3. First, we explored the impact of the task set’s **total utilization**. As can be seen in Figure 3a, in contrast to task set cardinality, total utilization influences certification time only slightly, and



■ **Figure 4** Number of steps in the arrival-curve prefix *vs.* (a) certification time of the most-preempted task and (b) the increase in response time for arrival bounds of types (iv) and (v).

only in the case of EDF. In fact, while high utilization can impact the maximum busy-window length in pathological cases, for the considered workloads, it did not make a significant difference in the expected case.

The major driver of certification runtime is instead the **search-space size**. In fact, as can be seen in Figure 3b, there is a clear linear correlation between the size of the search space of a task and its certification time. Note that, in contrast to the prior figures, Figure 3b shows the individual *per-task* certification time relative to the size of its search space. Certification finished in at most 5 minutes for most of the tasks (84%), and in less than 10 minutes for 92% of the tasks regardless of the preemption model and scheduling policy, peaking at 104 minutes under FP-EDF for a task with a search space containing 104547 points.

The search space and certification time of FP-FP and NP-FP are clustered in the bottom-left part of the plot. In contrast, FP-EDF and NP-EDF cover the entirety of the space, which is a result of the fact that the EDF search space is inherently larger and dependent on all tasks, as evident from Equations (4) and (5). In fact, 99% of tasks scheduled under an FP policy had a search space containing fewer than 104 points, whereas the 99th percentile under an EDF policy is 57470 points. Hence, task sets usually take much less time to be verified under an FP policy, which explains the gap between the EDF and FP policies in Figure 2.

Finally, we performed a **third experiment** with the goal of evaluating the role of the number of steps in arrival-curve prefixes. For this experiment, we fixed the number of tasks to $n = 25$ and varied instead the prefix size of all tasks $k \in \{2, 5, 10, 15, 25, 50, 100\}$. For each combination of scheduling and preemption policy, we generated 10 task sets for each k ($\times 7$) and utilization ($\times 5$), considering task sets composed homogeneously of tasks with either arrival model (iv) or (v) ($\times 2$), resulting in 700 task sets per policy and 2800 in total. To arrive at a given prefix size k , we always started from an arrival curve with $m_i = 100$ steps, and then iteratively merged steps to gradually shrink the prefix. To maximize the effect of the loss of information that results from the shortening of the prefix, we focused on the *lowest-priority* task under FP scheduling, and on the task with the *largest deadline* under EDF. In both cases, we refer to this task as the *most-delayed* task.

First, we investigated the impact of k on RTA **accuracy**. Figure 4a shows the relative increase in the response-time bound of the most-delayed task relative to its *baseline response-time bound*, which was obtained using the full prefix with 100 steps. By reducing k from 100 to 2, the response-time bound for the most-delayed task reaches a staggering $20\times$ increase, which (as expected) drops quickly as k is increased.

The results in Figure 4a should be seen in context of the corresponding **task certification times** that, as can be seen in Figure 4b, grow substantially with increasing k (note the log scale). EDF-scheduled task sets are once again substantially more expensive to analyze.

Overall, Figure 4 shows that, for complex workloads with irregular arrival processes, the number of steps of the arrival-curve prefix represents a major trade-off between certification time and analysis accuracy. It should be noted, however, that POET always managed to complete the certification, even for $k = 100$ steps. Furthermore, it is important to realize that full certification is expected only at the *final* stage of development – at which point runtimes in the order of several hours can be acceptable – and not during day-to-day development.

Overall, we consider POET to be a successful proof of concept. While the computational efficiency of the underlying proof assistant and libraries is (as expected) a major bottleneck, the experiments overall showed POET to cope with complex workloads and to scale to practical workload sizes. Foundational RTA is thus not only theoretically desirable, but also practical, and therefore worthy of further study, extension, and optimization.

9 Related Work

As already discussed in Section 2, POET draws inspiration and adopts terminology from Appel’s classic work on foundational proof-carrying code [3], which has been highly influential. Since its publication two decades ago, it has been widely adopted in the area of program verification [9, 32], and continues to play a major role in state-of-the-art verification tools [41].

POET is closely linked to PROSA [11]. While PROSA is the to-date largest machine-checked framework for real-time schedulability analysis – and presently the only one with an implementation of ARTA [8] – it is neither the first nor the only attempt in this direction [7, 20, 21, 46, 51]. It is worth noting that a foundational tool like POET is not inherently related to COQ: some of the just-cited papers make use of different proof assistants, namely NQTHM [44], PVS [45], and ISABELLE/HOL [43]. Though some are more suited than others, conceptually speaking, a foundational approach could be realized with any of these, and each would likely pose different challenges and trade-offs. In particular, the LEAN proof assistant [19] is a modern alternative to COQ based on the same underlying logic [15, 16, 39]; LEAN would likely be a viable alternative for use in foundational RTA tools.

Closest to POET in terms of objectives and approach are Mabillet et al.’s results validator for network calculus [36] (using ISABELLE/HOL) and Fradet et al.’s results validator CERTICAN [23] (using COQ and PROSA) for the CAN RTA implemented in RTAW-PEGASE [6]. In contrast to POET, which generates proofs as explainable evidence, but is intentionally left unverified, these tools do not generate proofs nor other evidence, but are themselves verified.

Finally, an alternative way to approach the schedulability analysis problem is to validate the correctness of a bound with model-checking techniques. In this approach, the system under analysis is first reduced to a model comprising a network of discrete automata with timed semantics. Then, a model checker explores the state space of the model with the objective of covering every possible trace, including those in which the worst-case response time of a task is experienced. Generally speaking, model-checking has been a highly successful technique as shown by tools like UPPAAL [5] and KRONOS [50] (both based on timed automata [1]) as well as HYTECH [30] (based on linear hybrid automata [2]). Compared to foundational RTA, model checking is an orthogonal technique with fundamentally different trade-offs and challenges. As a major advantage, a model checker requires no RTA theory to be developed or verified, since the worst case response-time is implicitly found during exploration of the state space. However, when compared to foundational RTA, model-checking requires a significantly larger, much more complex TCB. The reason is that practical model-checkers are typically large, nontrivial pieces of software that, due to model checking’s well-known state-space explosion problem and the resulting scalability challenges (e.g., [48]), have large incentives to

be heavily optimized. This naturally leads to the development of advanced techniques to prune search trees [5], speed up computations via statistical techniques [17], and hardware acceleration [4]. Each optimization technique increases the size of the TCB and arguably renders it more fragile. While Wimmer and Lammich [47] developed a verified unreachability certificate checker for timed automata, they reported it to be an order of magnitude slower and significantly more memory intensive than the state-of-the-art tool UPPAAL, which limits its practical use in the schedulability analysis of realistically sized task sets.

Regarding the explainability of results, model-checkers are capable of providing a counterexample leading to a worst-case scenario (e.g., a deadline is violated), but typically do not produce *evidence* that a property is *not* violated. Foundational RTA tools yield exactly the opposite: they do not give counterexamples, but do provide a sequence of machine-checked proofs that show the response-time bounds to be correct. In conclusion, both model checking and proof automation are important research directions, with diverse advantages and limitations. From a tool user's point of view, currently neither clearly dominates the other.

10 Conclusion

We have proposed foundational RTA as a means to obtain explainable, trustworthy evidence of temporal correctness and discussed the design and implementation of POET, the first foundational RTA tool. A foundational RTA produces proof-carrying response-time bounds that can be independently verified by a proof checker. Consequently, a foundational RTA tool does not have to be trusted and can be developed like any other application, while *its results* are trustworthy: fully explainable and verifiably correct.

While POET is an important first step demonstrating feasibility of the approach, for practical use, it will be necessary to go beyond ideal uniprocessor systems. In particular, it would be desirable to extend POET to more complex workloads (e.g., synchronization and precedence constraints), to more realistic system models (e.g., scheduling overheads), and to multiprocessor platforms (e.g., semi-partitioned scheduling).

References

- 1 Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
- 2 Rajeev Alur, Thomas A Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
- 3 Andrew W Appel. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256. IEEE, 2001.
- 4 Jiri Barnat, Luboš Brim, Milan Češka, and Tomáš Lamr. CUDA accelerated LTL model checking. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 34–41. IEEE, 2009.
- 5 Gerd Behrmann, Alexandre David, Kim G Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 125–126, 2006.
- 6 Marc Boyer, Jorn Migge, and Marc Fumey. PEGASE—a robust and efficient tool for worst-case network traversal time evaluation on AFDX. Technical report, SAE Technical Paper, 2011.
- 7 Marc Boyer, Pierre Roux, Hugo Daigormte, and David Puechmaile. A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

- 8 Sergey Bozhko and Björn B. Brandenburg. Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle. In *32nd Euromicro Conference on Real-Time Systems (ECRTS'20), July 7-10, 2020, Virtual Conference, 2020*.
- 9 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1):367–422, 2018.
- 10 Felipe Cerqueira, Geoffrey Nelissen, and Björn B Brandenburg. On strong and weak sustainability, with an application to self-suspending real-time tasks. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 26–1, 2018.
- 11 Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284. IEEE, 2016.
- 12 Jian-Jia Chen and Björn B Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. *Leibniz Transactions on Embedded Systems*, 4(1):01–1, 2017.
- 13 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, et al. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1), 2019.
- 14 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.
- 15 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.
- 16 Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.
- 17 Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *International Conference on Computer Aided Verification*, pages 349–355. Springer, 2011.
- 18 Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3), 2007.
- 19 Leonardo De Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.
- 20 Daniel de Rauglaudre. Vérification formelle de conditions d’ordonnancabilité de tâches temps réel périodiques strictes. In *JFLA-Journées Francophones des Langages Applicatifs-2012*, 2012.
- 21 Bruno Dutertre. The priority ceiling protocol: formalization and analysis using PVS. In *Proceedings of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS)*, pages 151–160, 1999.
- 22 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. A generalized digraph model for expressing dependencies. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 72–82, 2018.
- 23 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. CertiCAN: A tool for the Coq certification of CAN analysis results. In *RTAS*, 2019.
- 24 Pascal Fradet, Maxime Lesourd, Jean-François Monin, and Sophie Quinton. A generic coq proof of typical worst-case analysis. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–229. IEEE, 2018.
- 25 Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
- 26 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, Lecture Notes in Computer Science, pages 163–179, 2013.

- 27 David Griffin, Iain Bate, and Robert I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium (RTSS'20)*, December 1-4, Houston, TX, USA, pages 76–88. IEEE Computer Society, 2020.
- 28 Arpan Gujarati, Felipe Cerqueira, Björn B Brandenburg, and Geoffrey Nelissen. Correspondence article: a correction of the reduction-based schedulability analysis for apa scheduling. *Real-Time Systems*, 55(1):136–143, 2019.
- 29 Mario Günzel and Jian-Jia Chen. A note on slack enforcement mechanisms for self-suspending tasks. *Real-Time Systems*, pages 1–10, 2021.
- 30 Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.
- 31 Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolic. Analysis of buffering effects on hard real-time priority-preemptive wormhole networks. *arXiv preprint arXiv:1606.02942*, 2016.
- 32 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.
- 33 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.
- 34 Karthik Lakshmanan, Dionisio de Niz, and Ragnathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.
- 35 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- 36 Etienne Mabilie, Marc Boyer, Loïc Fejoz, and Stephan Merz. Towards certifying network calculus. In *ITP*, 2013.
- 37 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2021. doi:10.5281/zenodo.4457887.
- 38 George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.
- 39 Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.
- 40 Prosa. <http://prosa.mpi-sws.org/>.
- 41 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*, 2021.
- 42 The Coq Proof Assistant. <https://coq.inria.fr>.
- 43 The Isabelle Proof Assistant. <https://isabelle.in.tum.de/>.
- 44 The Nqthm Theorem Prover. <https://www.cs.utexas.edu/users/moore/best-ideas/nqthm/index.html>.
- 45 The PVS Proof Assistant. <https://pvs.csl.sri.com/>.
- 46 Matthew Wilding. A machine-checked proof of the optimality of a real-time scheduling policy. In *International Conference on Computer Aided Verification*, pages 369–378. Springer, 1998.
- 47 Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 61–78. Springer, 2018.
- 48 Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233. IEEE, 2019.
- 49 Maolin Yang, Jian-Jia Chen, and Wen-Hung Huang. A misconception in blocking time analyses under multiprocessor synchronization protocols. *Real-Time Systems*, 53(2):187–195, 2017.
- 50 Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

- 51 Xingyuan Zhang, Christian Urban, and Chunhan Wu. Priority inheritance protocol proved correct. In *International Conference on Interactive Theorem Proving*, pages 217–232. Springer, 2012.
- 52 Quan Zhou, Jihua Huang, Jianjun Li, and Zhi Li. Response time analysis for hybrid task sets under fixed priority scheduling. In *Proceedings of the IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 108–120. IEEE Computer Society, 2022.

Using Markov's Inequality with Power-Of-k Function for Probabilistic WCET Estimation

Sergi Vilardell ✉ 

Polytechnic University of Catalonia, Barcelona, Spain
Barcelona Supercomputing Center (BSC), Spain

Isabel Serra ✉ 

Barcelona Supercomputing Center (BSC), Spain
Centre de Recerca Matemàtica, Barcelona, Spain

Enrico Mezzetti ✉ 

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

Jaume Abella ✉ 

Barcelona Supercomputing Center (BSC), Spain

Francisco J. Cazorla ✉ 

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

Joan del Castillo ✉ 

Autonomous University of Barcelona, Spain

Abstract

Deriving WCET estimates for software programs with probabilistic means (a.k.a. pWCET estimation) has received significant attention during last years as a way to deal with the increased complexity of the processors used in real-time systems. Many works build on Extreme Value Theory (EVT) that is fed with a sample of the collected data (execution times). In its application, EVT carries two sources of uncertainty: the first one that is intrinsic to the EVT model and relates to determining the subset of the sample that belongs to the (upper) tail, and hence, is actually used by EVT for prediction; and the second one that is induced by the sampling process and hence is inherent to all sample-based methods. In this work, we show that Markov's inequality can be used to obtain provable trustworthy probabilistic bounds to the tail of a distribution without incurring any model-intrinsic uncertainty. Yet, it produces pessimistic estimates that we shave substantially by proposing the use of a *power-of-k* function instead of the default identity function used by Markov's inequality. Lastly, we propose a method to deal with sampling uncertainty for Markov's inequality that consistently improves EVT estimates on synthetic and real data obtained from a railway application.

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

Keywords and phrases Markov's inequality, probabilistic time estimates, probabilistic WCET, Extreme Value Theory

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.20

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant PID2019-110854RB-I00 / AEI / 10.13039/501100011033 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773).



© Sergi Vilardell, Isabel Serra, Enrico Mezzetti, Jaume Abella, Francisco J. Cazorla, and Joan del Castillo;

licensed under Creative Commons License CC-BY 4.0

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).

Editor: Martina Maggio; Article No. 20; pp. 20:1–20:24



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

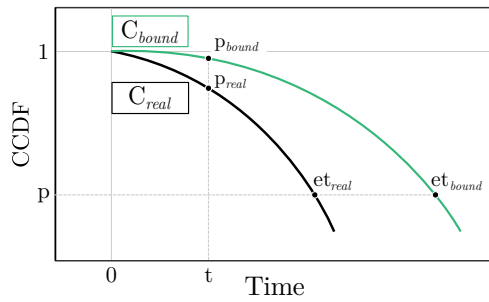
Deriving Worst-Case Execution Time (WCET) estimates of programs is pivotal to show that a real-time system meets its timing requirements [55]. A strand of works tackles this challenge with probabilistic analysis [13] as a way to deal with the increasing complexity of the processors used in real-time systems. These works predict the values of exceedance probabilities of the uppermost tail (a.k.a. high quantiles) of the execution time distributions of programs, which is normally referred to as probabilistic WCET (pWCET) estimate.

Extreme Value Theory (EVT) [16] has been consolidated as a modeling approach for pWCET estimation [1, 7, 27, 46, 52]. Sound applications of EVT can deliver tight and trustworthy pWCET estimates with high confidence. However, the use of EVT for pWCET estimation suffers from several sources of uncertainty that can be categorized into statistical and model-intrinsic (or simply model for short) uncertainty [8]. Statistical uncertainty is, in fact, intrinsic to any sample-based process. It encompasses as the first aspect the testing conditions under which the experiments are performed in reference to those that can arise during system operation. Testing conditions that are representative or worse than operation conditions are the basis to attain representativeness of the sample data (execution time) [2, 4, 25, 38] so that the pWCET estimate holds during system operation. A second aspect of statistical uncertainty relates to the natural uncertainty of a sampling process that, in general, reduces as the sample size increases, and that is handled with confidence intervals. Sampling uncertainty impacts summary statistics (e.g. mean) and tail fitting methods, whose goodness – either of their hypotheses or outcome – is assessed with specific methods [4, 42]. Model uncertainty, instead, relates to uncertainties intrinsic to the mathematical model used for tail prediction. In the case of EVT, model uncertainty relates to determining the threshold from which the upper tail starts. This threshold plays a key role on the trustworthiness (safeness) of EVT results since only samples above it (i.e. the maxima data set) are fed into EVT for pWCET estimation. There is not an exact mathematical method to derive this threshold. Instead, current methods estimate the tail of a distribution [10] based on plot inference [12, 19, 28] and regression analysis [9].

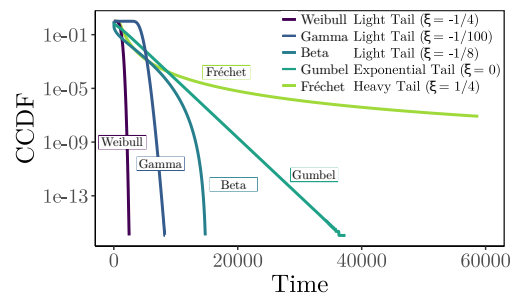
As the first set of contributions, we show that Markov's inequality [36] provides pWCET estimates that are trustworthy by construction at the theoretical (analytical distribution) level and hence, free of any model uncertainty. We, then, illustrate that Markov's inequality is highly pessimistic for decreasing exceedance probabilities, which are specially relevant for pWCET estimation (Section 3). To cope with this limitation, we propose the use of Markov with power-of- k (MIK) functions, $f(X) = X^k$, instead of the identity function, $f(X) = X$, used by the default Markov's formulation. We develop the MEMIK (Minimum Envelope for Markov's Inequality) algorithm that exploits MIK, delivering trustworthy upper-bounds to the underlying distribution, while achieving much tighter estimates than Markov for arbitrarily low exceedance probabilities (Section 4).

As the second set of contributions, at the empirical level, we address the statistical uncertainty of the Markov model by proposing the RESTK (restricted k) method. In order to approximate the expected value of X^k , also defined as the k -th moment of a distribution, RESTK derives the *sample moment* of a distribution from a collected sample (Section 5).

We evaluate RESTK on a set of representative distributions and tail functions and show that it consistently outperforms EVT: EVT produces 21.8% over-estimation on average (up to 37%) and RESTK keeps average over-estimation at 9.4% (up to 20%). We also evaluate RESTK on sampled data from a real railway use case: the central safety processing unit of the European Train Control System (ETCS) reference architecture. On average, RESTK over-estimates less than 9% with respect to real data (distribution) and it outperforms EVT.



■ **Figure 1** Generic representation of the tightness of pWCET estimations. CCDF stands for Complementary Cumulative Distribution Function.



■ **Figure 2** CCDF for GEV distributions with $\xi = -1/4$, $\xi = 0$, and $\xi = 1/4$; $\xi = -1/8$ for Beta; and $\xi = -1/100$ for Gamma.

The rest of this paper is structured as follows. Section 2 presents some relevant background, and introduces and exemplifies EVT model uncertainty stemming from tail selection. Section 3 introduces Markov's inequality and discusses its use for pWCET estimation. Section 4 introduces our proposal of using $f(X) = X^k$ to tighten pWCET estimates. Section 5 introduces the RESTK method as a way to deal with statistical uncertainty for Markov's inequality. Section 6 compares the pWCET projections obtained with EVT and Markov models. Section 7 shows analogous results for the railway use case. Section 8 surveys the main related works, and Section 9 presents the main conclusions of this work.

2 Background and Problem Statement

2.1 Probabilistic WCET estimation

Let us assume a random variable $X > 0$, corresponding to the execution time of a real-time program and whose maximum value is finite. We aim at bounding the probabilities of X in its uppermost tail (i.e. high execution times) by providing safe and tight timing bounds.

An execution time probability distribution C_{bound} , see Figure 1, is said to upper-bound another probability distribution C_{real} – so being a pWCET bound – when for any exceedance probability p the execution time of the former, $et_{bound}(p)$, is higher (or equal) than that of the latter, $et_{real}(p)$. It also holds that, for any given execution time et , the exceedance probability of the former $p_{bound}(et)$ is higher (or equal) than that of the latter $p_{real}(et)$. This can be expressed as $tightness(p) = \frac{et_{bound}(p)}{et_{real}(p)}$.

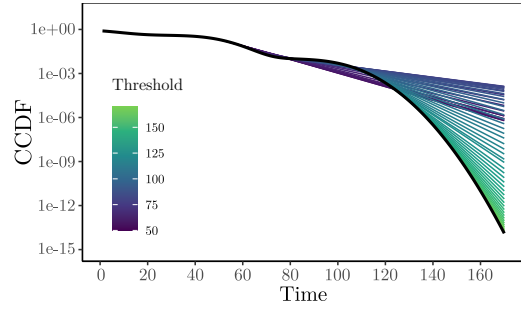
2.2 Representative Distributions

In the scope of timing analysis of real-time programs, it is reasonable to assume that the target program will always terminate. Hence, a WCET value upper-bounding all possible program's executions always exists. It therefore follows that, the tail of its execution time distribution is necessarily a light distribution, which has been shown to be safely and tightly upper-bounded with light and exponential tail distributions [1, 17, 46]. For this reason, we focus on light and exponential tails.

While heavy tails are, therefore, unnecessarily pessimistic and hence, left out of our discussion, an execution time sample could apparently correspond to a heavy tail distribution. This could be, for instance, the case when the sample size is not large enough to provide

■ **Table 1** Reference distributions used to drive discussions. The parameters are μ (mean), σ (standard deviation), α (shape) β (shape), λ (scale), and σ (scale).

Acronym	Parameters
Gaussian	$\mu = 100, \sigma = 10$
Weibull	$\alpha = 4, \lambda = 80$
Beta	$\alpha = 1/4, \beta = 8$
Mixture	$\mu = \{5, 50, 100\},$ $\sigma = 10$ $w = \{0.6, 0.39, 0.1\}$



■ **Figure 3** EVT estimation on the mixture distribution (three Gaussian with parameters $\mu = \{5, 50, 100\}, \sigma = \{10, 10, 10\}$ and weights $w = \{0.6, 0.39, 0.01\}$, respectively).

sufficient representativeness in a mixture distribution. In the general case, this concern relates to the sampling process (sample size in particular), shared across all applications of statistics, and therefore, beyond the scope of our methodology. Nevertheless, nothing precludes the use of our methodology for heavy tails.

The different types of tails are illustrated with the Complementary Cumulative Distribution Function (CCDF) of several example distributions in Figure 2. All three Generalized Extreme Value (GEV) distributions have location $\mu = 0$ and scale $\sigma = 1000$: the Weibull distribution (light tail) with shape $\xi = -1/4$ has a sharp slope and a maximum value (2500 in the example); Gumbel (exponential tail) with $\xi = 0$ has also a relatively sharp slope but it has no maximum; and the exceedance probability for the Fréchet distribution (heavy tail) with $\xi = 1/4$ decreases polynomially. The Beta (light tail) distribution has a similar profile to the Weibull distribution and it is commonly used to model high quantiles of random variables with a finite defined interval [3, 30, 37], which fits WCET modeling. And the Gamma (light tail) distribution is also typically used in EVT and WCET [4].

All former distributions are unimodal, which is a common way to represent execution time profiles. However, it has also been shown that the execution time of many programs presents “clusters”. That is, the program’s execution time varies around two or more central values rather than vary around a single mean. This results in *mixture distributions* that can arise both in sequential applications and parallel applications [1, 56, 57]. As an example of the former, let us assume a program whose execution time profile is influenced by the latency of its load/store operations which can hit or miss the data L1 cache and L2 cache across different runs depending on the program’s inputs. This results in a mixture distribution with 3 clusters (peaks) around the data L1, L2 and memory latencies, respectively. An example of mixture distribution is represented by the black line in Figure 3.

Overall, we use a solid set of reference distributions that are in line with the state of the art [4, 11, 27, 42]. In particular, we use unimodal (Gaussian, Weibull, Beta, and Gamma) and challenging multi-modal distributions (Gaussians and Weibulls) with different tail profiles to increase representativeness. Besides, we use a real railway use case in Section 7. In order to drive the explanations, we use a specific set of reference parameters for several distributions (see Table 1), while in the evaluation section we analyze a wider set of parameters (see Table 3). We have not included the Gamma in the reference distributions as it yields very similar results to the Weibull. However, results are provided for two different Gamma distributions in the (evaluation) Section 6 as part of the complete result set.

2.3 EVT usage for pWCET estimation

The main theorems of EVT can be deemed as two different ways of thinking about the extremes [22]. The first, Block Maxima (BM), splits a sample of a distribution into fixed-size *blocks* and selects the maximum value in each block. The second one, Peaks over Threshold (PoT) defines a *threshold* such that the values above it are deemed as “rare enough”, i.e. they are considered the tail of the distribution. EVT’s main result is characterized on both theorems when the set of maxima and the threshold tend to infinity.

Both, PoT and BM, share the fundamental goal of identifying where the tail of the distribution starts to fit the proper distribution to the tail. However, for PoT such cut point is defined explicitly as a threshold, therefore allowing its use in the domain of distributions from a mathematical perspective. Hence, in this work we focus on PoT for our analysis, although mathematical uncertainties identified are generally shared by PoT and BM.

Peaks Over Threshold (PoT). Given a random variable X with a cumulative distribution function (CDF), F , and a threshold, $u > 0$, such that $y = x - u$, the excess random variable X_u defined as $(X - u | X > u)$ is given by the CDF F_u defined as

$$F_u(y) = P(X - u \leq y | X > u) = \frac{F(u + y) - F(u)}{1 - F(u)}, \quad y \geq 0 \quad (1)$$

A PoT model is a semi-parametric model where the law of X for $x < u$ is described by the empirical distribution, and for $x > u$ is defined by

$$P(X > x) = S(x) = S(u)S_u(x - u) \quad (2)$$

where $S(x) = 1 - F(x)$ and $S_u(x_u) = 1 - F_u(x_u)$ are the CCDFs, with $x_u = x - u$.

► **Theorem 1** (Pickands–Balkema–de Haan theorem [5]). *Let F be a distribution function such that $F(x) < 1$ with F_u being its conditional excess distribution function. Then, F_u converges in probability to the generalized Pareto distribution (GPD) for large u . That is, $F_u \xrightarrow{\mathcal{L}} G(y; \xi, \sigma)$ as $u \rightarrow \infty$, where*

$$G(y; \xi, \sigma) = \begin{cases} 1 - \left(1 + \frac{\xi y}{\sigma}\right)^{-\frac{1}{\xi}} & \text{if } \xi \neq 0 \\ 1 - \exp\left(-\frac{y}{\sigma}\right) & \text{if } \xi = 0 \end{cases} \quad (3)$$

With $\sigma > 0$, and $y \geq 0$ when $\xi \geq 0$ and $0 \leq y \leq -\sigma/\xi$ when $\xi < 0$. This result is crucial for tail estimation. If the conditions for the theorem are met, the GPD family of functions results in accurate estimates for the most extreme values on the tail. However, the conditions are met when the threshold $u \rightarrow \infty$, which brings uncertainty in the implementation of estimates with the GPD since finite values of u need to be used.

Overall, there is an unavoidable model uncertainty when selecting the threshold u by definition due to having to select a finite value. This is beyond specific aspects related to statistical uncertainty from the sampling process or the threshold u estimation process. Tail selection has been addressed by techniques like the Hill Plot based on Hill estimator [28], the Mean Excess Plot [19], or the CV Plot [12]. The selection of the threshold u changes the fit of the GPD model and requires careful analysis and selection to prevent GPD from over-approximating or under-approximating the analyzed distribution.

As an illustrative (visual) example, Figure 3 shows a mixture of three Gaussians with $\mu = \{5, 50, 100\}$, $\sigma = \{10, 10, 10\}$ and weights $w = \{0.6, 0.39, 0.01\}$. Each mode around Time $\{5, 50, 100\}$ represents a Gaussian in the mixture. For each different tail threshold u such that

$u > 60$, we fit an exponential tail, given that Gaussian distributions have exponential tails. In Figure 3 we see three scenarios that we exemplify with approximate ranges of u . For values of u around $[60, 80]$ (purple lines), we see how the exceedance probability of the reference distribution is underestimated in the Time range $[80, 120]$ (probabilities $10^{-3} - 10^{-5}$). For u in the range $[120, 140]$ (blue-green lines), GPD over estimates the reference distribution. For values of u above 140 (green lines), the estimate becomes increasingly tight.

3 Chebyshev and Markov Inequalities for pWCET Estimation

This section analyzes the applicability of Markov's inequality as an alternative model to EVT for the problem of trustworthy pWCET estimation and shows that it is not subject to any model uncertainty, thus resulting in provably safe bounds for the analyzed distribution. Yet, as we also show in this section, Markov's inequality produces very pessimistic estimates. For completeness, we start by introducing Chebyshev's inequality, as it is the generalization of Markov's inequality.

3.1 Chebyshev's Inequality

► **Definition 2.** Let X be a discrete random variable with probability function $f_X(x)$, where x are the particular values that X can take. The expected value of X is:

$$E(X) = \sum_{x \in X} x f_X(x) = \sum_{x \in X} x P(X = x) \quad (4)$$

► **Theorem 3** (Chebyshev's Inequality [49]). Let X be a non-negative random variable, $b > 0$, and f a non-negative and increasing function. Chebyshev's inequality states that:

$$P(X \geq b) \leq \frac{E(f(X))}{f(b)} \quad (5)$$

For WCET estimation, X corresponds to the execution time distribution to be bounded, and b to an execution time for which we want to find its upper-bound probability.

Regarding f , it needs to be defined to realize the general Chebyshev's inequality into a specific upper-bound function. The function f can be any non-negative function so that for a particular domain D , Property 6 holds.

$$\forall x \in D, \quad f(x) > 0 \quad (6)$$

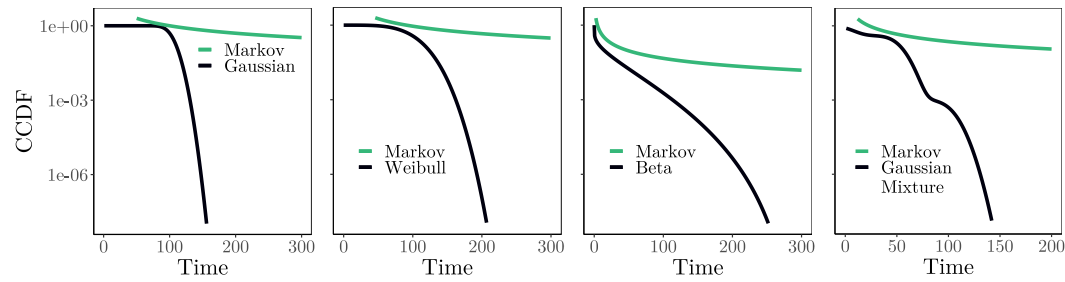
f must also be an increasing function so that for a given interval I , Property 7 holds.

$$a, b \in I \quad | \quad a < b, \quad \Rightarrow \quad f(a) \leq f(b) \quad (7)$$

Interestingly, Chebyshev's inequality does not require determining where the actual tail distribution starts but instead works with the entire distribution, hence removing EVT's model uncertainty for tail selection. In fact, Chebyshev's inequality carries no model uncertainty.

► **Observation 4.** Chebyshev's inequality is a model uncertainty free general model for pWCET estimation.

Chebyshev's inequality also applies to continuous and discrete distributions regardless of their characteristics (e.g. shape, variance, kurtosis, etc.). Also, it is non-parametric, i.e. it makes no assumption on parameters for the studied distribution.



(a) Gaussian distribution. (b) Weibull distribution. (c) Beta distribution. (d) Gaussian Mixture.

■ **Figure 4** Markov's Inequality bound for the reference distributions.

3.2 Markov's Inequality

Markov's inequality is a specific instantiation of Chebyshev's inequality.

► **Corollary 5** (Markov's Inequality [36]). *Let $X > 0$ and let the function f be the identity function $f(X) = X$. Hence, Markov's inequality yields:*

$$P(X \geq b) \leq \frac{E(X)}{b} \quad (8)$$

As for the baseline Chebyshev's inequality, Markov's inequality holds for any real-valued random variable with a finite expected value and positive value b . Also, it (i) is a trustworthy upper-bound, by construction, of the underlying distribution; (ii) has no model uncertainty; (iii) is non-parametric; and (iv) can be applied to discrete and continuous distributions.

3.3 Markov's inequality on low probabilities

Besides trustworthiness, pWCET estimates are also required to be reasonably tight, specially for the range of relevant probabilities usually considered for pWCET estimation, e.g. $[10^{-6}, 10^{-15}]$. In this line, our analysis shows that Markov's inequality tends to be hardly useful for pWCET estimation.

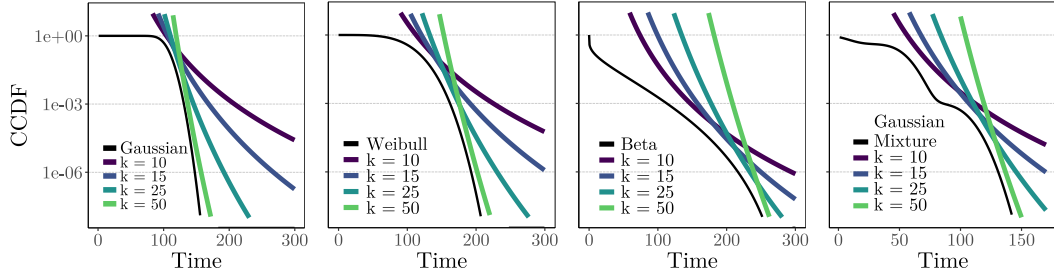
This is better illustrated in Figure 4 which shows for all considered distributions the probability bounds given by Markov's inequality. We can observe that estimates are very pessimistic, orders of magnitude higher than the real probability. This includes the range of probabilities of interest for pWCET estimation. In fact, we see that Markov's inequality never goes below 10^{-2} for all distributions for the execution time value range plotted.

► **Observation 6.** *Markov's inequality in its original form is too pessimistic to be usable in practice for pWCET estimation.*

4 Power-of-K functions for Markov's Inequality

One of the main insights of this work is that the key reason for Markov's inequality resulting in loose pWCET bounds lies on the fact that it builds on the identity function, $f(X) = X$, of a random variable. In this section, we show how a different function can lead to increased tightness on the produced pWCET estimates while preserving trustworthiness.

In particular, we contend that the power function for any $k \in \mathbb{R}_{>0} = \{k \in \mathbb{R} | k > 0\}$, i.e. $f(X) = X^k$, or power-of- k function for short, can be safely used instead of the identity function to obtain tighter and trustworthy pWCET bounds.



(a) Gaussian distribution. (b) Weibull distribution. (c) Beta distribution. (d) Gaussian Mixture.

■ **Figure 5** MIK bounds for the reference distributions.

► **Definition 7.** Let X be a discrete random variable and k a positive real value. The expected value of X^k , also defined as the k -th (theoretical) moment, is:

$$E(X^k) = \sum_x x^k P(X = x) \quad (9)$$

► **Corollary 8** (Markov's Inequality to the power-of- k). Let $X > 0$ and let the function f be the power-of- k function $f(X) = X^k$. Markov's inequality to the power-of- k yields:

$$P(X \geq b) \leq \frac{E(X^k)}{b^k} \quad (10)$$

Hence, the probability that X takes a value greater or equal to b is bounded by $E(X^k)/b^k$. This makes Markov's inequality with $f(x) = x^k$ (MIK for short) a safe pWCET estimate when X represents the execution time of a program.

Proof. Theorem 3 holds true when Property 6 and Property 7 are fulfilled. The power-of- k function *does not fulfill those properties in general*. However, when the x domain is restricted to the positive real numbers $\mathbb{R}_{>0} = \{x \in \mathbb{R} | x > 0\}$, which in fact includes the domain of execution time profiles, the power-of- k function does fulfill Properties 6 and 7 since x is positive, so x^k is also positive and an increasing function. ◀

Overall, for this application scenario ($x \in \mathbb{R}_{>0}$), Equation 10 is an upper-bound when using the power-of- k function onto the reference distribution for any value of $k \in \mathbb{R}_{>0}$. Hence, it can be leveraged for pWCET estimation.

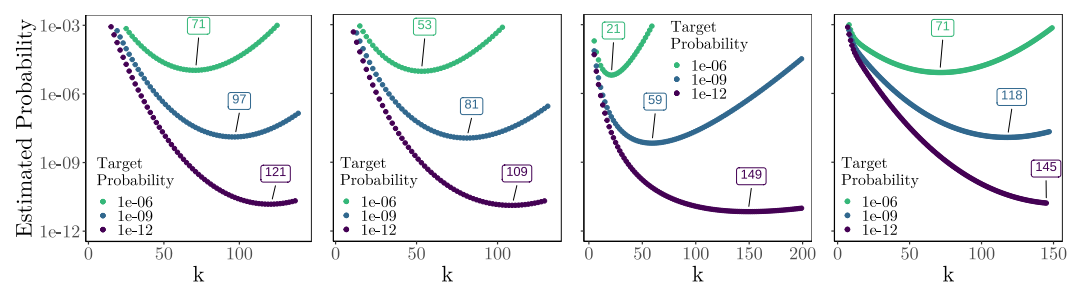
It is worth noting that other functions can exist that fulfill Properties 6 and 7. While exploring them is part of our future work, as shown in Section 4.1, MIK (i.e. $f(X) = X^k$) achieves very tight pWCET estimates which leaves small room for improvement.

► **Observation 9.** For every value of $k \in \mathbb{R}_{>0}$, MIK (i.e. Markov's inequality with $f(X) = X^k$) is a safe pWCET estimate when X represents the execution time of a program.

Note that there is no theoretical constraint on the maximum value of k , which can be any positive real number k .

4.1 Tightness of MIK for increasing values of k

Once we have established the safe use of MIK for pWCET estimation, we illustrate the impact of varying k on tightness. Figure 5 shows for several values of k , $\{10, 15, 25, 50\}$, and the reference distributions presented before, that MIK dramatically increases the tightness



(a) Gaussian distribution. (b) Weibull distribution. (c) Beta distribution. (d) Gaussian Mixture.

■ **Figure 6** Evolution of MIK bounds with the value of k .

provided by Markov's inequality (Figure 4), while remaining a trustworthy upper-bound for every value of k . We also observe that MIK tightness remains for high exceedance probabilities, which hence makes it a promising model to provide pWCET estimates.

► **Observation 10.** *Markov's inequality with $f(X) = X^k$ heavily reduces the pessimism of Markov's inequality.*

Intuitively, from Figure 5, higher values of k result in tighter estimates, i.e. minimizing the distance between the reference distribution and the upper-bound distribution. However, this is not always the case. For instance, if we take a closer look at the Beta distribution (Figure 5(c)), we see that at cut-off probabilities 10^{-3} and 10^{-6} the tightest MIK estimates are not obtained for the highest value of k evaluated (50).

► **Observation 11.** *For a given threshold probability, higher values of k do not necessarily result in a tighter MIK bound.*

This is better illustrated with the examples in Figure 6 that shows quantitatively the evolution of the MIK bound obtained for varying values of k .

In this experiment, for the value of the target distribution at each probability, we evaluate MIK for different values of k . As it can be seen, for every threshold probability and distribution the value of k resulting in the tightest estimation is different. For instance, for the Gaussian distribution (Figure 6(a)) and target probability 10^{-6} , $k = 71$ produces the tightest estimate, while for 10^{-9} and 10^{-12} the best k is 97 and 121, respectively. As a general trend, we see that the lower the target exceedance probability, the higher the value of the best k is. Yet, the highest value of k evaluated for each target probability does not produce the tightest bound. Overall, for each probability there exists a value of k producing the tightest upper-bound, with the optimal value of k depending on the actual reference distribution.

► **Observation 12.** *Increasing the tightness of MIK for each probability is an optimization problem on k which only increases accuracy and does not affect trustworthiness.*

In order to address this optimization problem, we propose **MEMIK** (Minimum Envelope for MIK, i.e. Markov's Inequality to the power-of- K). MEMIK combines the results of MIK bounds obtained for any value of k by keeping, for each point in an interval, the value of k producing the tightest estimate. This set of points form an envelope that is a provable trustworthy and tight tail bound by construction for any exceedance probability. Therefore, MEMIK improves the pessimistic upper-bounds of Markov's inequality (see Figure 4), with a much tighter envelope that is usable for pWCET estimation. Formally, the MEMIK bound is defined as follows:

$$P(X \geq b) \leq \min_k \frac{E(X^k)}{b^k} \quad \text{for } k > 0 \quad (11)$$

■ **Algorithm 1** Compute an envelope using Power-of- k .

```

1: function MEMIK( $t_{range}, t_{step}, p_{all}, max_k(p_{all}), k_{step}$ )
2:   for  $t \in t_{range}, t_{step}$  do
3:     for  $p \in p_{all}$  do
4:        $mik_{best}(p) \leftarrow \infty$ 
5:       for  $k \in ([1, max_k(p)], k_{step})$  do
6:          $vpred \leftarrow EVAL(t, k, p)$ 
7:         if  $vpred < mik_{best}(p)$  then
8:            $mik_{best}(p) = vpred$ 
9:            $k_{best}(p) = k$ 
10:        end if
11:       end for
12:        $envelope(t, p) \leftarrow \langle mik_{best}(p), k_{best}(p) \rangle$ 
13:     end for
14:   end for
15:   return  $envelope$ 
16: end function

```

MEMIK, see Algorithm 1, which uses point-wise power-of- k Markov's inequality values, performs a simple complete exploration of MIK values over a given time range t_{range} and over a configurable range of k , determined by the maximum value max_k to be explored for each probability p in the set of probabilities of interest p_{all} . The granularity of MEMIK exploration over t and k is determined by the t_{step} and k_{step} parameters respectively. For each probability p in the interval of interest p_{all} (line 3) and k in the range determined by $max_k(p)$ (line 5), the algorithm estimates the value of the target distribution, $EVAL(t, k, p)$ (line 6). To that end, we evaluate Equation 10 with $E(X^k)$, which corresponds to the theoretical k -th moment of the target distribution from Equation 9, obtaining $vpred$ that we compare to the best MIK value so far for all considered k (line 7).

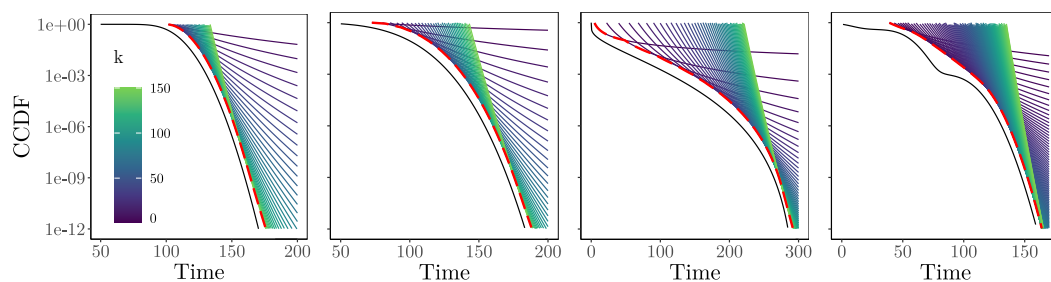
The minimum MIK value produced for a given t and across all k values ($mik_{best}(p)$) is stored, together with the corresponding $k_{best}(p)$, in the data structure $envelope$ (line 12). Eventually, after iterating over the whole time interval, the algorithm returns the $envelope$ data structure (line 15) which holds the point-wise definition of the approximation envelope. Note that, if for any value t , the value of $k_{best}(p)$ matches $max_k(p)$, then $max_k(p)$ can be increased to find tighter bounds.

We applied MEMIK to our reference distributions, for which we can derive the theoretical moments. For this experiment, we varied the value of k up to 150 with $k_{step} = 1$. We obtained the envelopes depicted in Figure 7 which provides evidence that MEMIK produces tight and trustworthy estimates for all distributions, with an observed error of around 5%.

Overall, this section provides the key result that our proposal, Markov's inequality to the power-of- k (MIK), unlike EVT, suffers no model uncertainty at the theoretical level and hence, provides correct-by-construction pWCET estimates that are much tighter than those provided by the default Markov's inequality. This leaves sampling uncertainty as the problem to address.

5 Handling Markov Sampling Uncertainty

So far we have been reasoning on examples for which we could compute the theoretical k -th moments for each distribution. This was possible since the distributions considered were known and, hence, we could compute exactly the value of each moment (i.e. $E(X^k)$ for each



(a) Gaussian distribution. (b) Weibull distribution. (c) Beta distribution. (d) Gaussian Mixture.

■ **Figure 7** MEMIK bound (envelope) on the reference distributions.

value of k) using its analytical closed form. However, we need to consider the scenario in which only samples are available. Hence, as for any other sample-based method, we need to deal with the underlying sampling uncertainty.

A commonality of sample-related methods like EVT [13], and something that we also assume, is that, input samples are independent and identically distributed [16] (i.i.d.) or at least exhibit extremal independence [44]. The i.i.d. property can be pursued with platform randomization [31] or data (time measurements) sample randomization [33].

For the case of the Markov's inequality, this translates into deriving the **sample moments**, referred to as $\hat{E}(X^k)$ (for each value of k). In particular, we need an estimator for high-order moments that can produce good estimates for any distribution.

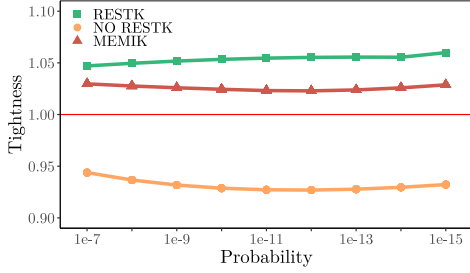
5.1 Sample moment estimation

The k -th moment of a random variable X can be estimated as (N is the sample size [23]):

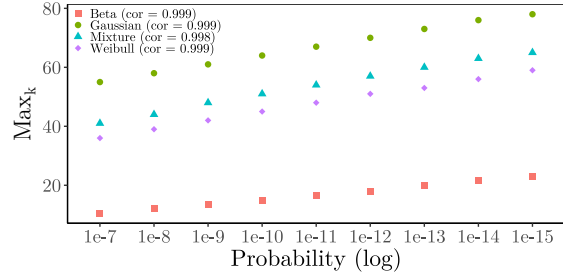
$$\hat{E}(X^k) = \frac{1}{N} \sum_i^N X_i^k \quad (12)$$

In general, this estimator is the best one to deal with high-order sample moments [26], as it is asymptotically unbiased. Given that it asymptotically tends to a Gaussian distribution [6], the properties of the Central Limit Theorem's apply to it. However, the estimator is asymptotically unbiased [23] only when using large amounts of data. For instance, for a sample of a Gaussian distribution with $\mu = 100, \sigma = 10$ and $N = 10^3$, the difference between the 3rd exact moment, and the sample moment using Equation 12 is about 0.02%, it is between 1% and 3% for the 50th moment, and can be up to 160% for the 100th moment.

When the sample moment, i.e. the estimate of the k -th moment, is higher than the theoretical moment, there is a risk of underestimating the upper tail of the distribution by assigning to a certain probability a smaller quantile than it has in reality. The approach we propose to limit that risk consists in setting a maximum value of k allowed for each different probability, which we refer to as max_k . In order to illustrate the effect of not controlling max_k , Figure 8 assesses the tightness of MEMIK over a particular set of nine probabilities (from 10^{-7} to 10^{-15}). The red triangles represent the theoretical bound of MEMIK using Equation 11 with $E(X^k)$ being the theoretical moments, while the orange dots (NO RESTK) represent the application over multiple simulations of Equation 11 using the sample moment from Equation 12. On both applications we set up a high value for max_k , 150. We can see how the loss of consistency of the sample moment estimator on Equation 12 results in bad estimates.



■ **Figure 8** MEMIK with sample moments ($n = 1000$ and $n_{sims} = 100$) on the reference Gaussian distribution with (RESTK) and without (NO RESTK) restricting k . Also MEMIK evaluated with theoretical moments.



■ **Figure 9** Minimum max_k values for the reference distributions used in this work.

An intuitive way to control the gap between the k -th sample and the theoretical moments is to vastly increase the sample size, which in our domain would require an unaffordable number of runs of the task under analysis. Alternatively, one can control the range of values of k explored in Equation 12. By doing so, we trade some tightness for trustworthiness. That is, if we explore values of k until a low max_k limit, we can see in Figure 6 that the theoretical bound is not optimal in terms of accuracy. On the other hand, small values of max_k also limit the inaccuracy of the sample moment estimator. Note that it is not possible to identify a general optimal value of max_k for any kind of data under analysis. The appropriate max_k value changes across distribution types, across the same distribution type with different parameters, and even across probabilities for a given distribution. For this reason, we propose the *restricted k method* (**RESTK**) that builds on the information gathered directly from the samples to derive max_k so as to produce trustworthy and tight results.

5.2 Understanding the behavior of max_k

We gain insight on the behavior of max_k along 3 axes. We analyze i) whether for a given distribution there exists a pattern for max_k that can provide tight and safe results using the sample moment estimator; ii) whether this pattern can be predicted using only the information from the sample; and iii) whether the pattern can be generalized for any distribution.

We focus on the same example distributions used in previous sections. We fix the interval $[1, 150]$ as exploration range for k . In order to account for sample uncertainty, we perform $n_{sims} = 10^3$ Monte-Carlo simulations, each one considering a random sample of size $n = 10^3$.

We first compute Markov to the power-of- k (MIK) using Equation 10 with the sample moment estimator in Equation 12, for all selected k . In each simulation, we increase values of k and find the **first** (smallest) value of k that produces underestimation. This is computed by comparing the estimation with the actual value of the distribution. That is, we take the value of k for which the estimated quantile is smaller than the theoretical quantile. Then, we set $k - 1$ as our max_k . For each Monte-Carlo simulation, we compute the max_k for all target probabilities. When all simulations are performed, we keep the smallest max_k for each probability. As a result, for each experiment of n_{sims} Monte-Carlo simulations, we obtain one value of max_k for each probability under study. We plot those values in Figure 9 from which we derive two main conclusions.

We observe that the values in Figure 9 follow a linear distribution. For each distribution we fit minima max_k values to a linear model and we derive the resulting correlation coefficient. The correlation coefficient quantifies the strength of the linear relation between two variables. It ranges between -1 and 1 , with 1 or -1 indicating perfect correlation (all points would lie along a straight line).

The distributions we use in this work include 4 types of unimodal distributions, 2 multimodal distributions and several parameters thereof (see Table 3 in Section 6). For all those distributions, Table 2 shows that the correlation coefficient is very high and steadily stays above 0.99. Even the empirical distributions derived from the case study analyzed in Section 7 result in a high coefficient of correlation (0.98 on average), despite they tend to produce more variability in the estimations. Hence, empirical evidence in support of linearity for the minimum observed value of max_k is strong for the distribution tails and range of probabilities representative for the WCET. Besides, the application of RESTK includes a method to assess whether the linearity property holds, building on the observed data. It is also noted that, similar empirical reasoning is used to support statistical arguments whether phenomena adheres to specific distributions builds on empirical tests. For instance, in the case of EVT, previous work uses QQ-plots to assess, based on observation, whether some tails can be considered exponential [34].

■ **Table 2** Correlation Coefficient for all the distributions used in this work.

Gaus1	Gaus2	Weib1	Weib2	Beta1	Beta2	Gam1	Gam2	Mix1	Mix2	Mix3	Mix4
.999	.999	.999	.999	.999	.998	.999	.999	.998	.998	.997	.998

Overall, by restricting max_k , one can avoid under-estimating the upper tail of the modeled distribution. This is exemplified in Figure 8, where the green squares (RESTK) represent the estimates obtained for the Gaussian distribution when applying the max_k values in Figure 9. By restricting max_k , we address the lack of trustworthiness in Figure 8 (NO RESTK) and produce tight and trustworthy bounds. Analogous results are obtained for the other distributions.

5.3 Deriving max_k from unknown distributions

When deriving Figure 9, we built on the values of the theoretical quantiles so as to determine the value of k for which the sample moment starts underestimating. Given a sample of size 10^p , we can estimate confidently quantiles from exceedance probabilities bigger than $1/10^{(p-1)}$. In this case, based on the law of large numbers, it is very likely to see around 10 realizations whose probability is of the order of $10^{(p-1)}$ [47]. That is, on a sample of size $N = 1000$ we will see around 10 realizations whose probability is 0.01 (1%). Therefore, for a sample of size 10^4 , quantiles corresponding to exceedance probabilities 10^{-3} and 10^{-2} can be estimated easily with the usual quantile estimation functions from statistical packages [29]. The computation of confidence intervals for quantile estimation can be done using distribution-free methods like Kaigh and Lagenbruch or bootstrap [48]; and in any case the accuracy of the estimation can be increased using a bootstrap technique to correct variability.

RESTK estimates at least three max_k points to construct its model and assess linearity. The latter is assessed by deriving the correlation coefficient for these three points. If such coefficient is above a threshold $th = 0.95$, we deem max_k boundary to be linear and vice-versa (in which case RESTK cannot be applied). For instance, the quantiles corresponding to exceedance probabilities 10^{-3} , 10^{-4} , and 10^{-5} can be estimated very accurately with a sample of size $n = 10^6$. These reference points allow us to assess when RESTK underestimates,

■ **Algorithm 2** Computing the boundary necessary to apply RESTK approach.

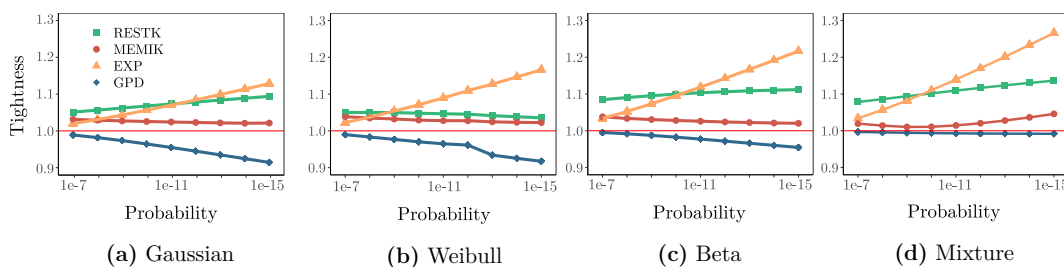
```

1: function RESTKBOUNDARY(sample, krange, kstep, nboot, nsims, ptest, pall, th)
2:   for  $p \in p_{test}$  do
3:      $q_{est} \leftarrow estimateQuantiles(sample, p)$ 
4:     for  $sim \in n_{sim}s$  do
5:        $sample_{boot} \leftarrow bootstrap(sample, n_{boot})$ 
6:        $max_k(p) \leftarrow \infty$ 
7:        $tightness_{best}(p) \leftarrow \infty$ 
8:       for  $k \in k_{range}, k_{step}$  do
9:          $vref \leftarrow q_{est}$ 
10:         $vpred \leftarrow MIK(k, sample_{boot}, p)$ 
11:         $tightness \leftarrow vpred/vref$ 
12:        if  $tightness < 1$  then
13:          break
14:        end if
15:        if  $tightness < tightness_{best}(p)$  then
16:           $current_k(p) = k$ 
17:        end if
18:        end for
19:        if  $current_k(p) < max_k(p)$  then
20:           $max_k(p) = current_k(p)$ 
21:        end if
22:      end for
23:    end for
24:     $max_k(p_{all}) \leftarrow buildLinearModel(max_k(p_{test}), p_{all})$ 
25:    if  $corr(max_k(p_{all})) < th$  then
26:      return no pWCET estimate
27:    end if
28:    return  $max_k(p_{all})$ 
29: end function

```

and hence generate three max_k points, one for each probability. With those points, we can generate the regression line that projects max_k for any probability of interest (e.g., those in Figure 9) and assess that the correlation coefficient is above the desired threshold.

Algorithm 2 generalizes the RESTK process starting from a main sample of the distribution under analysis (size 10^p), selecting the range of k to explore and the number of $n_{sim}s$ to run. First, RESTK estimates the quantiles at given probabilities p_{test} from the main sample (Line 3), e.g. $10^{-(p-1)}$, $10^{-(p-2)}$, and $10^{-(p-3)}$. For each simulation, n_{boot} bootstrap samples of size 10^{p-3} are generated from the main sample (Line 5). For each of these samples, we compute the maximum k and maximum tightness for all the probabilities to test. The predicted value $vpred$ is obtained by computing MIK from Equation 10 with the sample moment estimator, $\hat{E}(X^k)$ in Equation 12, (Line 10). The tightness of the predicted value is computed (Line 11) by using as reference value $vref$ the estimated quantiles obtained before (Line 3). The algorithm finds the values of k in the considered range that produce the tightest estimate (Lines 15-16) and terminates its exploration as soon as a k that underestimates ($tightness < 1$) is found (Line 12). After exploring all selected probabilities for all k and all simulations, the algorithm returns the smallest max_k across all simulations (Lines 20-21).



■ **Figure 10** MEMIK with sample moments ($n = 1000$ and $n_{sims} = 100$) on the reference distributions, hence restricting k (RESTK). Also MEMIK evaluated with theoretical moments (MEMIK), and EVT evaluated with exponential tails (EXP) and with a GPD with light tails (GPD).

Once the max_k (e.g. for $10^{-(p-1)}$, $10^{-(p-2)}$, and $10^{-(p-3)}$) are obtained, RESTK builds a linear model for all possible probabilities (Line 24). The final check (Line 25) will ensure that the correlation of max_k is above $th = 0.95$, otherwise RESTK provides no pWCET estimate. As we show in Table 2, the correlation should always be close to 1 for max_k . The threshold $th = 0.95$ is a standard and stringent threshold for confidence in statistics, and used as a way to discard estimates that do not meet the safety criteria of finding a proper max_k .

RESTK enables the computation of a value for max_k that reduces the risk of underestimation for any probability of interest. This can be directly exploited by running MEMIK (Algorithm 1) on a predetermined range for k and for each probability p under study by using $max_k(p)$ as the upper bound for k , instead of considering an arbitrary range. Also, note that with RESTK, we use Equation 10 with sampled moments ($\hat{E}(X^k)$) as $EVAL(t, k, p)$ function in MEMIK. The rest of the MEMIK algorithm remains unchanged when using the RESTK method.

6 RESTK and EVT PW CET Estimates on Distributions

Our implementation of RESTK and MEMIK is programmed in *R* [40]. We run experiments on an Intel Core i5-7600K CPU clocked at 3.8GHz. The maximum execution time required per experiment was very short, 50 milliseconds or lower. We analyze values of k in the range $k \in [1, 150]$ with $k_{step} = 1$ to estimate max_k for all reference distributions, which, as shown in Figure 9 is a wide enough range to find the best max_k across distributions and probabilities. For all methods compared in this section, we use a sample size of $n = 10^6$. For RESTK, we set the number of bootstrap simulations to $n_{boot} = 2000$.

For EVT, we use the PoT methodology to fit tails and the CV Plot [12] to find the appropriate threshold for the PoT model. We use two EVT models fitted for pWCET estimation, namely exponential and light tails models. For each specific model, a different threshold using the CV plot will be found to ensure the best possible fit.

- *Exponential*: with an exponential model, the shape of the GPD is fixed to $\xi = 0$, which only leaves us to estimate the threshold u and the scale σ . The threshold is estimated with the CV Plot fixing $\xi = 0$, which finds where the exponential tail begins. Once we find the tail, we separate it from the rest of the sample and estimate the scale σ with it.
- *GPD light tails*: for the light tails model, we need the value of ξ , with $\xi < 0$, best fitting the data. Using the CV Plot we find the threshold u where the light tail begins. Then, we separate the tail from the sample and estimate the shape ξ and the scale σ .

■ **Table 3** Distributions used for the analysis and their respective parameters.

Acronym	Type	Parameters
Gaussian1	Gaussian	$\mu = 100, \sigma = 10$
Gaussian2	Gaussian	$\mu = 100, \sigma = 50$
Weibull1	Weibull	$\alpha = 4, \lambda = 80$
Weibull2	Weibull	$\alpha = 8, \lambda = 80$
Beta1	Beta	$\alpha = 8, \beta = 1/4$
Beta2	Beta	$\alpha = 8, \beta = 1/8$
Gamma1	Gamma	$\alpha = 100, \lambda = 1$
Gamma2	Gamma	$\alpha = 150, \lambda = 1$
Mixture1	Mixture of Gaussians	$\mu = \{5, 50, 100\}, \sigma = 10, w = \{0.6, 0.39, 0.1\}$
Mixture2	Mixture of Gaussians	$\mu = \{50, 100, 400\}, \sigma = 50, w = \{0.6, 0.39, 0.1\}$
Mixture3	Mixture of Weibulls	$\lambda = \{5, 50, 100\}, \alpha = 4, w = \{0.6, 0.39, 0.1\}$
Mixture4	Mixture of Weibulls	$\lambda = \{5, 50, 100\}, \alpha = 8, w = \{0.6, 0.39, 0.1\}$

Reference Distributions. We start the comparison with Figure 10 that depicts for the reference distributions the results of PoT with exponential and light tail models (EXP and GPD). It also shows the results obtained with MEMIK, which we can obtain as we have the actual distributions, and RESTK. Note that MEMIK provides the theoretical bound achievable with RESTK – it produces a safe bound and the tightest estimates. RESTK, EXP, and GPD build on a sample (the same one for a fair comparison) of the distribution. Following common practice, we show in Figure 10 the bias of our estimator, which is the expected value (mean) of RESTK output. It is noted that in RESTK application process all the distributions of this section fulfilled the linearity assessment (line 25 in Algorithm 2).

As we can see in this initial set of results, GPD tends to underestimate while EXP increases overestimation for high exceedance probabilities. RESTK produces values that are more consistent across all probabilities, improving EXP specially for higher exceedance probabilities. For 10^{-12} overestimates are 8%, 13%, 24% and 11% for the four reference distributions, respectively. The values increase to 13%, 20%, 37% and 17% for 10^{-15} . It can also be observed that the overestimation introduced by RESTK w.r.t. MEMIK to handle sampling uncertainty is limited: at 10^{-12} the difference is 4.75 percentage points (p.p) on average with a maximum of 8 p.p across all four reference distributions, and at 10^{-15} the overestimation difference is 5.25 p.p on average with a maximum of 10 p.p.

Extended set of Distributions. We consider a wider set of parameters for each distribution as listed in Table 3, resulting in 12 different distributions. The first distribution of each type (but the Gamma) is the reference distribution with the parameters used in previous sections. The rest of the distributions of each type encompass a different set of parameters to increase representativeness. The set of values explored for each parameter aims at showing the capabilities of RESTK under different scenarios. To that end we modify the following parameters.

1. the variance of the distribution for Gaussian1 and Gaussian2; and Mixtures1 and Mixture2
2. the shape of the tail of the distribution for Beta1 and Beta2, Weibull1 and Weibull2, Gamma1 and Gamma2, and Mixture3 and Mixture4.

■ **Table 4** Tightness of the different models (MEK stands for MEMIK and RES for RESTK).

	probability 10^{-12}				probability 10^{-15}			
	GPD	EXP	MEK	RES	GPD	EXP	MEK	RES
Gaussian1	0.93	1.08	1.02	1.06	0.90	1.13	1.02	1.06
Gaussian2	0.90	1.20	1.05	1.14	0.86	1.28	1.03	1.11
Weibull1	0.91	1.13	1.02	1.09	0.87	1.20	1.02	1.09
Weibull2	0.96	1.09	1.03	1.04	0.94	1.14	1.05	1.04
Beta1	0.98	1.24	1.03	1.18	0.98	1.37	1.03	1.20
Beta2	0.98	1.17	1.03	1.11	0.98	1.26	1.02	1.13
Gamma1	0.93	1.09	1.03	1.07	0.89	1.11	1.03	1.07
Gamma2	0.95	1.09	1.02	1.06	0.92	1.13	1.02	1.07
Mixture1	0.92	1.11	1.03	1.03	0.88	1.17	1.02	1.02
Mixture2	0.94	1.16	1.03	1.07	0.90	1.23	1.03	1.05
Mixture3	0.88	1.25	1.03	1.15	0.83	1.37	1.02	1.13
Mixture4	0.94	1.15	1.02	1.15	0.91	1.23	1.03	1.16

This covers **all** possible variability scenarios as the scale and location do not affect the results for Markov's Inequality. As shown in [36], a change of location does not affect the inequality if the shift keeps the random variable positive, $P(X - a \geq b - a) \leq \frac{E(X-a)}{b-a} < \frac{E(X)}{b}$. Also, scaling the random variable X as λX where λ is real-valued, does not affect Markov's Inequality as $P(\lambda X \geq \lambda b) \leq \frac{E(\lambda X)}{\lambda b} = \frac{\lambda E(X)}{\lambda b} = \frac{E(X)}{b}$.

Looking at the results for the broader set of experiments in Table 4, we observe the following:

- GPD is always close to the true quantile, but in all cases it produces optimistic results. Furthermore, the higher exceedance probability, the more optimistic the estimate is. For instance, GPD on the Gaussian1 has a tightness of $\{0.93, 0.90\}$ at $p = \{10^{-12}, 10^{-15}\}$ respectively. This behavior is observed for all reference distributions.
- EXP follows the opposite pattern. In general, we can see in Figure 10 that, for an exceedance probability $p = 10^{-7}$, the estimates across distributions are always safe and quite tight. However, for higher exceedance probabilities, EXP tends to give more pessimistic estimates. For instance, EXP on the Gaussian1 has a tightness of 1.01 at $p = 10^{-7}$ that increases to 1.13 at $p = 10^{-15}$. At this probability and across all distributions EXP overestimation is 21.8% on average, 37% in the worst case.
- The estimates with MEMIK, i.e. with theoretical moments, are very tight, below 6% for all distributions.
- RESTK achieves results similar to MEMIK, preserving tightness and trustworthiness. Even for very high exceedance probabilities, RESTK is able to produce consistent estimates. Building on the Gaussian1 distribution, we see that while EXP can achieve a tighter estimate for low exceedance probabilities (e.g. $p = 10^{-7}$), EXP suffers from increased pessimism for higher exceedance probabilities whereas RESTK stays stable. This behavior is more striking in distributions harder to analyze like mixtures. For instance, for Mixture1 RESTK not only maintains tightness stable across probabilities, $\{1.03, 1.02\}$, but it gets also a tighter bound than EXP for probabilities $p = 10^{-9}$ and beyond as seen in Figure 10. At 10^{-15} RESTK estimates across all distribution overestimate by 9.4%, far below the 21.8% of EXP. In the worst case it is 20% (w.r.t. to 37% of EXP).

Overall, experimental results show the ability of RESTK to produce bounds suitable for pWCET estimation, being those trustworthy, tight and stable across probabilities and distributions, as opposed to existing models, which fail to meet all three goals simultaneously. The benefits of RESTK increase as the cutoff probability decreases to $10^{-12} - 10^{-15}$, which are the main range of interest for pWCET estimation considering maximum failure rates of 10^{-9} per hour and tasks running thousands of times per hour.

7 Railway Use Case

In order to evaluate the effectiveness of RESTK, we use an industrial critical real-time use case. In particular, we focus on the central safety processing unit of the European Train Control System (ETCS) reference architecture. The ETCS is a safety-critical application (SIL 4) responsible of signaling and control in the European Rail Traffic Management System (ERTMS) framework.

ETCS protects the train motion by constantly monitoring traveled distance and speed, and is programmed to activate the emergency break system whenever unauthorized speed values are detected. The ETCS subsystem comprises three main tasks that are executed sequentially to provide the required safety function: the Odometry module, estimating a set of parameters based on the inputs collected from the train environment (e.g., estimated position); the Service module, managing the Service braking system; and the Emergency module, actually controlling the Emergency braking system. While all three tasks do exhibit strict real-time requirements, we focus our evaluation on the Emergency module, the core of the ETCS safety-critical module.

The ETCS validation suite, which is made available with the application, includes 10 different input vectors (TEST0 to TEST9) corresponding to the operating conditions for functional and timing validation regarded as relevant by the application owner. We collected execution times for each input vector, which stimulates a different Emergency module operational scenario with its own timing distribution. Experiments were conducted on a Cobham Gaisler LEON3 platform.

Ground truth. For real programs, for which the actual distribution is not known, common practice consists in using as ‘ground truth’ the observed quantiles for samples as large as reasonably possible (e.g. 10^4 [34] and 10^8 [46]). We follow the same approach and consider the quantile observed for the 10^7 sample as the reference value. More than two weeks of execution were needed to complete the execution of all the 10^7 runs per input vector (TEST).

Setup. The setup and parameters used to run RESTK are the same used for the reference distributions. As the number of runs we have is $n = 10^7$ per input set, we make projections for $p = 10^{-6}$, for which the observed frequencies closely match the actual probability (i.e. 95% confidence intervals are within 0.1% of the mean). In this case, the estimated quantile at probability $p = 10^{-6}$ is our ground truth. Only TEST6 is an exception to this and, due to the variability observed for that quantile, we use a 95% confidence interval, which is 1% off the mean. In this section, we use a sample size of $n = 10^4$ for GPD, EXP and RESTK. Note that MEMIK with theoretical moments cannot be used since the actual distribution is unknown and we only have sampled data.

Results. As part of the RESTK application process, all the 10 distributions fulfilled the linearity assessment (line 25 in Algorithm 2). As shown in Table 5, pWCET estimates are similar to those presented in the previous section. In particular:

■ **Table 5** Tightness of the estimates for the ETCS case study.

	probability 10^{-6}						
	GPD	EXP	RESTK		GPD	EXP	RESTK
TEST0	1.01	1.21	1.10	TEST5	0.98	1.12	1.07
TEST1	0.98	1.12	1.06	TEST6	0.94	1.06	1.01
TEST2	1.00	1.13	1.11	TEST7	1.01	1.16	1.09
TEST3	0.98	1.20	1.10	TEST8	1.01	1.23	1.12
TEST4	0.99	1.17	1.13	TEST9	0.98	1.11	1.10

- GPD achieves extremely tight estimates for 4 tests, with tightness up to 1.01, but on the other 6 tests it produces optimistic results. In general, while potentially very tight, GPD can easily underestimate the bounds.
- On the other hand, EXP never underestimates although it produces pessimistic results, as high as 23% for such a relatively low quantile.
- RESTK consistently produces tighter estimates than EXP in all tests for the use case. On average, EXP exhibits 15.1% pessimism, whereas RESTK reduces it to 8.9%.

Discussion. Overall, with the combined results over a wide set of distributions, shown in the previous section, and the results for the ETCS case study presented in this section, we conclude that RESTK consistently provides tighter estimates than EXP and improves for lower exceedance probabilities.

8 Related Works

Probabilistic and statistical approaches [13, 18] have been increasingly considered as a promising solution to cope with the rise in complexity of hardware and software systems, as determined by unprecedented increasing computing performance requirements [20].

Extreme Value Theory (EVT), a consolidated approach for modeling and predicting the occurrence of rare events, has emerged as the preferred option for modeling the worst-case execution time (WCET) of software programs. EVT has been considered particularly fit for probabilistic modeling of WCET as the latter is normally considered a rare event in the program's timing behavior. EVT is at the foundation of several Measurement-Based Probabilistic Timing Analysis (MBPTA) approaches [1, 7, 13, 27, 46, 52], which have been already positively assessed in some industrial use cases [54]. Probabilistic approaches building on sample and population sizes have also been built for overlapping concerns across task scheduling and timing analysis [39], hence being orthogonal to WCET estimation. Several works assess the necessary conditions for a correct application of EVT to the timing problem since an inattentive application of the EVT statistical tools can severely affect both trustworthiness and quality of the derived probabilistic WCET (pWCET) bounds [24, 34, 38].

Several studies focus on EVT applicability preconditions on the (timing) observations being independent and identically distributed (i.i.d.) random variables [13, 17]. EVT has also been shown to be applicable also to stationary data preserving extremal independence [44]. Different statistical tests have been assessed for that purpose in the real-time literature [1, 13, 17, 42]. Also, platform randomization [31] or data sample randomization [33] have been used to meet the i.i.d. requirement. However, even in case such statistical preconditions are met, the reliability of the obtained pWCET bounds is still affected by the choice in the EVT inputs

(i.e. selection of samples belonging to the tail) and parameters of the fit distribution. Several methods have been proposed for selecting and assessing the quality of EVT parameters and sample selection and, in turn, their impact on the trustworthiness of the computed bounds [1, 4, 43].

Statistical tests have been also proposed, for example, to assess the reliability of pWCET bounds in [4]. Unfortunately, model uncertainty in EVT application cannot be removed or quantified since no existing approach provides an exact, optimal method for tail selection. A different challenge to pWCET reliability relates to *representativeness* of the observations [2, 4, 25, 38] which corresponds, in the timing domain, to guaranteeing the observations are actually representative or upperbound the execution time distribution, which in fact, is an inherent trait for any sampled-based approach. Statistical and model uncertainties in the scope of EVT are discussed in [8] where robustness in tail estimation is achieved by considering, together with GEV, a family of *plausible* probability models (i.e., close to GEV) and selecting the most conservative estimated probability value among all models. Markov's and Chebyshev's inequalities have been historically applied in a wide variety of fields such as Engineering [50], Big Data sampling [45], and radiative transfer [51]. In the context of real-time computing, moment-based bounds on tail probabilities have also been considered in the scope of probabilistic schedulability analysis [14, 15, 21, 53]. Chernoff bounds [14, 15] and generalizations thereof [53] are exploited to compute the cumulative distribution of the interference caused by higher-priority tasks on a task response time, ultimately delivering a probability for deadline misses. While building on application of Markov's inequality to the timing dimension, these approaches do not address the problem of computing probabilistic bounds to tasks' execution time, which are instead assumed to be available, but offer a scalable alternative to the computational complexity of convolutions. Some works [35, 41] consider the use of Chebyshev's inequality for WCET and/or cache hit and miss rates estimation. However, those works consider Chebyshev's inequality (without power-of- k functions) only to estimate the impact of the variance on those metrics, and focus on the analysis of statistical uncertainties. Other works consider using higher moments to improve concentration inequalities similar to Markov [32]. Although their approach is similar, the work focuses purely on the theoretical tightness of the probability bounding. The challenge we overcame in our work was translating this theoretical advantage into a practical reality for unknown distributions, which can easily lead to optimistic bounds without proper care as we shown. That is, neither model uncertainties nor tightness aspects for pWCET estimation for high quantiles are considered. Authors in [58] consider a similar approach to those works for WCET estimation, but discard Chebyshev's inequality altogether given the pessimism expected for high quantiles.

9 Conclusions

In this work we presented for the first time a method based on Markov's Inequality for pWCET estimation that represents a solid alternative approach to EVT. In particular, we showed that MIK (Markov Inequality to the power-of- k) has no model uncertainty and proposed a method to handle sampling uncertainty (RESTK) that consistently provides more trustworthy, tighter and stable results than EVT in different scenarios, including a railway case study. These promising results suggest that RESTK can be effectively used as a standalone method for pWCET estimation, or even as an alternative approach to validate EVT results in those cases where EVT is already consolidated. In this line, the fact that MIK (RESTK) and EVT build on completely different mathematical foundations provides stronger evidence on the trustworthiness of the obtained pWCET estimates.

References

- 1 Jaume Abella, Maria Padilla, Joan Del Castillo, and Francisco J. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4), June 2017. doi:10.1145/3065924.
- 2 Jaume Abella, Eduardo Quiñones, Franck Wartel, Tullio Vardanega, and Francisco J. Cazorla. Heart of gold: Making the improbable happen to increase confidence in MBPTA. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 255–265. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.33.
- 3 Charalampos Antoniadis, Dimitrios Garyfallou, Nestor Evmorfopoulos, and Georgios Stamoulis. Evt-based worst case delay estimation under process variation. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1333–1338, 2018. doi:10.23919/DATE.2018.8342220.
- 4 Luis Fernando Arcaro, Karila Palma Silva, Rômulo Silva de Oliveira, and Luís Almeida. Reliability test based on a binomial experiment for probabilistic worst-case execution times. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 51–62. IEEE, 2020.
- 5 August Aimé Balkema and Laurens de Haan. Residual Life Time at Great Age. *The Annals of Probability*, 2(5):792–804, 1974. doi:10.1214/aop/1176996548.
- 6 Francesco Bartolucci and Luca Scrucca. Point estimation methods with applications to item response theory models. In Penelope Peterson, Eva Baker, and Barry McGaw, editors, *International Encyclopedia of Education (Third Edition)*, pages 366–373. Elsevier, Oxford, third edition, 2010. doi:10.1016/B978-0-08-044894-7.01376-2.
- 7 Kostiantyn Berezovskyi, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. WCET Measurement-Based and Extreme Value Theory Characterisation of CUDA Kernels. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems, RTNS '14*, pages 279–288, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2659787.2659827.
- 8 Jose Blanchet, Fei He, and Karthyek Murthy. On distributionally robust extreme value analysis. *Extremes*, 23:317–347, 2020. doi:10.1007/s10687-019-00371-1.
- 9 Frederico Caeiro and Maria Gomes. Semi-parametric tail inference through probability-weighted moments. *Journal of Statistical Planning and Inference*, 141(2):937–950, 2011. doi:10.1016/j.jspi.2010.08.015.
- 10 Frederico Caeiro and Maria Gomes. Threshold selection in extreme value analysis. *Extremes*, September 2014. doi:10.1007/s10687-021-00405-7.
- 11 Enrique Castillo, Ali Hadi, Narayanaswamy Balakrishnan, and José Sarabia. *Extreme Value and Related Models With Applications in Engineering and Science*. Wiley series in probability and statistics. Wiley, January 2005.
- 12 Joan Del Castillo, Jalila Daoudi, and Richard Lockhart. Methods to distinguish between polynomial and exponential tails. *Scandinavian Journal of Statistics*, 41(2):382–393, 2014. doi:10.1111/sjos.12037.
- 13 Francisco J. Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1), February 2019. doi:10.1145/3301283.
- 14 Kuan-Hsun Chen and Jian-Jia Chen. Probabilistic schedulability tests for uniprocessor fixed-priority scheduling under soft errors. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8, 2017. doi:10.1109/SIES.2017.7993392.
- 15 Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. Efficient computation of deadline-miss probability and potential pitfalls. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 896–901, 2019. doi:10.23919/DATE.2019.8714908.
- 16 Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer, 2001. doi:10.1007/978-1-4471-3675-0.

- 17 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Codé Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, July 2012. doi:10.1109/ECRTS.2012.31.
- 18 Robert Davis and Liliana Cucu-Grosjean. A survey of probabilistic schedulability analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):04–1–04:53, 2019. doi:10.4230/LITES-v006-i001-a004.
- 19 Anthony C. Davison and Richard L. Smith. Models for exceedances over high thresholds. *Journal of the Royal Statistical Society. Series B (Methodological)*, 52(3):393–442, 1990. doi:10.1111/j.2517-6161.1990.tb01796.x.
- 20 Deloitte. *Semiconductors – the Next Wave Opportunities and winning strategies for semiconductor companies*, 2019. URL: <https://www2.deloitte.com/content/dam/Deloitte/cn/Documents/technology-media-telecommunications/deloitte-cn-tmt-semiconductors-the-next-wave-en-190422.pdf>.
- 21 Jose L. Diaz, Daniel F. Garcia, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José M. Lopez, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 289–300, 2002. doi:10.1109/REAL.2002.1181583.
- 22 Paul Embrechts, Thomas Mikosch, and Claudia Klüppelberg. *Modelling Extremal Events: For Insurance and Finance*. Springer-Verlag, Berlin, Heidelberg, 1997.
- 23 Ronald A. Fisher. Moments and product moments of sampling distributions. *Proceedings of the London Mathematical Society*, s2-30(1):199–238, 1930. doi:10.1112/plms/s2-30.1.199.
- 24 Samuel Jimenez Gil, Iain Bate, George Lima, Luca Santinelli, Adriana Gogonel, and Liliana Cucu-Grosjean. Open challenges for probabilistic measurement-based worst-case execution time. *IEEE Embedded Systems Letters*, 2017. doi:10.1109/LES.2017.2712858.
- 25 Fabrice Guet, Luca Santinelli, and Jérôme Morio. On the representativity of execution time measurements: Studying dependence and multi-mode tasks. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, volume 57 of *OASICS*, pages 3:1–3:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASICS.WCET.2017.3.
- 26 Paul R. Halmos. The Theory of Unbiased Estimation. *The Annals of Mathematical Statistics*, 17(1):34–43, 1946. doi:10.1214/aoms/1177731020.
- 27 Jeffery Hansen, Scott Hissam, and Gabriel A. Moreno. Statistical-Based WCET Estimation and Validation. In Niklas Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 of *OpenAccess Series in Informatics (OASICS)*, pages 1–11, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. doi:10.4230/OASICS.WCET.2009.2291.
- 28 Bruce M. Hill. A Simple General Approach to Inference About the Tail of a Distribution. *The Annals of Statistics*, 3(5):1163–1174, 1975. doi:10.1214/aos/1176343247.
- 29 Rob Hyndman and Yanan Fan. Sample quantiles in statistical packages. *The American Statistician*, 50:361–365, November 1996. doi:10.1080/00031305.1996.10473566.
- 30 Norman Lloyd Johnson. *Continuous univariate distributions. Vol. 2*. Wiley and Sons, New York, 2nd ed. / norman l. johnson, samuel kotz, n. balakrishnan. edition, 1994.
- 31 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 513–518. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.116.
- 32 Bar Light. Concentration inequalities using higher moments information. *arXiv*, 2020. doi:10.48550/ARXIV.2006.05130.

- 33 George Lima and Iain Bate. Valid Application of EVT in Timing Analysis by Randomising Execution Time Measurements. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 187–198, 2017. doi:10.1109/RTAS.2017.17.
- 34 George Lima, Dário Dias, and Edna Barros. Extreme Value Theory for Estimating Task Execution Time Bounds: A Careful Look. In *Euromicro Conference on Real-Time Systems, ECRTS*, 2016. doi:10.1109/ECRTS.2016.20.
- 35 Jyh-Charn S. Liu and Sharif M. Shahrier. On predictability of caches for real-time applications. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 52–56, 1994. doi:10.1109/MASCOT.1994.284448.
- 36 Andrey Markov. On certain applications of algebraic continued fractions. *Ph.D. thesis, St. Petersburg*, 1884.
- 37 Thomas Mikosch. Regular variation, subexponentiality and their applications in probability theory. *International Journal of Production Economics - INT J PROD ECON*, January 1999.
- 38 Suzana Milutinovic, Enrico Mezzetti, Jaume Abella, Tullio Vardanega, and Francisco J. Cazorla. On uses of Extreme Value Theory fit for industrial-quality WCET analysis. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, France, June 14-16, 2017*, pages 1–6. IEEE, 2017. doi:10.1109/SIES.2017.7993402.
- 39 Sims Osborne and James H. Anderson. Simultaneous multithreading and hard real time: Can it be safe? In Marcus Völz, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPICs*, pages 14:1–14:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ECRTS.2020.14.
- 40 R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021. URL: <https://www.R-project.org/>.
- 41 Archana Ravindar and Y. N. Srikant. Estimation of probabilistic bounds on phase CPI and relevance in WCET analysis. In *Proceedings of the Tenth ACM International Conference on Embedded Software, EMSOFT '12*, pages 165–174, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2380356.2380388.
- 42 Federico Reghenzani, Giuseppe Massari, and William Fornaciari. Probabilistic-WCET reliability: Statistical testing of EVT hypotheses. *Microprocess. Microsystems*, 77:103–135, 2020. doi:10.1016/j.micpro.2020.103135.
- 43 Federico Reghenzani, Luca Santinelli, and William Fornaciari. Dealing with uncertainty in pWCET estimations. *ACM Trans. Embed. Comput. Syst.*, 19(5):33:1–33:23, 2020. doi:10.1145/3396234.
- 44 Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart. On the Sustainability of the Extreme Value Theory for WCET Estimation. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASICs)*, pages 21–30, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASICs.WCET.2014.21.
- 45 Ashwin Satyanarayana. Intelligent sampling for big data using bootstrap sampling and Chebyshev inequality. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6, 2014. doi:10.1109/CCECE.2014.6901029.
- 46 Karila Palma Silva, Luis Fernando Arcaro, and Romulo Silva De Oliveira. On Using GEV or Gumbel Models When Applying EVT for Probabilistic WCET Estimation. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 220–230, 2017. doi:10.1109/RTSS.2017.00028.
- 47 Didier Sornette. *Critical Phenomena in Natural Sciences: Chaos, Fractals, Selforganization and Disorder: Concepts and Tools*. Springer, January 2006. doi:10.1007/3-540-33182-4.
- 48 Seth M. Steinberg and Clarence E. Davis. Distribution-free confidence intervals for quantiles in small samples. *Communications in Statistics - Theory and Methods*, 14(4):979–990, 1985. doi:10.1080/03610928508805144.
- 49 Pafnuti Tchebichef. Des valeurs moyennes. *Journal de mathématiques pures et appliquées*, 12(2):177–184, 1867.

- 50 Vladimir Utkin. Calculating the reliability of machine parts on the basis of the Chebyshev inequality. *Russian Engineering Research*, 32, January 2012. doi:10.3103/S1068798X11120264.
- 51 Gladimir V.G. Baranoski, Jon G. Rokne, and Guangwu Xu. Applying the exponential Chebyshev inequality to the nondeterministic computation of form factors. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 69(4):447–467, 2001. doi:10.1016/S0022-4073(00)00095-9.
- 52 Sergi Vilardell, Isabel Serra, Jaume Abella, Joan Del Castillo, and Francisco J. Cazorla. Software timing analysis for complex hardware with survivability and risk analysis. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 227–236, 2019. doi:10.1109/ICCD46524.2019.00036.
- 53 Georg von der Brüggen, Nico Piatkowski, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. Efficiently approximating the probability of deadline misses in real-time systems. In *ECRTS*, 2018. doi:10.4230/LIPIcs.ECRTS.2018.6.
- 54 Franck Wartel, Leonidas Kosmidis, Adriana Gogonel, Andrea Baldovin, Zoë R. Stephenson, Benoit Triquet, Eduardo Quiñones, Code Lo, Enrico Mezzetti, Ian Broster, Jaume Abella, Liliana Cucu-Grosjean, Tullio Vardanega, and Francisco J. Cazorla. Timing analysis of an avionics case study on complex hardware/software platforms. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 397–402. ACM, 2015. doi:10.7873/DATE.2015.0189.
- 55 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P.uschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.
- 56 Julien Worms and Sid Touati. Parametric and Non-Parametric Statistics for Program Performance Analysis and Comparison. [Research Report] RR-8875, INRIA Sophia Antipolis - I3S; Université Nice Sophia Antipolis; Université Versailles Saint Quentin en Yvelines; Laboratoire de mathématiques de Versailles, 2017. URL: <https://hal.inria.fr/hal-01286112>.
- 57 Julien Worms and Sid Touati. Modelling program's performance with gaussian mixtures for parametric statistics. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):383–395, 2018. doi:10.1109/TMSCS.2017.2754251.
- 58 Pavel G. Zaykov and Jan Kubalčík. Worst-case measurement-based statistical tool. In *2019 IEEE Aerospace Conference*, pages 1–10, 2019. doi:10.1109/AERO.2019.8741824.