DISS. ETH NO. xxxxx

Implementation of Mixed-Criticality Applications on Multi-Core Architectures

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
GEORGIA GIANNOPOULOU
M.Sc. ETH Zurich

born on 19.12.1985
citizen of Greece

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Rodolfo Pellizzoni, co-examiner

2016

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 167

Georgia Giannopoulou

Implementation of Mixed-Criticality Applications on Multi-Core Architectures

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. xxxxx

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Rodolfo Pellizzoni, co-examiner

Examination date: December 16, 2016

Abstract

Embedded systems are increasingly used in safety-critical domains, such as avionics and automotive. Given the potential impact of failures on human lives and the environment, the correct design of such systems is typically subject to certification. Correctness depends not only on functional specifications, but also on the ability to fulfill stringent timing constraints. For this, system designers need to provide real-time guarantees, usually in the form of analytically derived worst-case execution time bounds.

Nowadays, safety-critical systems are often mixed-critical, in which multiple functionalities with different safety criticality levels are integrated in a common embedded platform for reduced cost, size, weight and power dissipation. The current industrial practice requires that applications with different safety criticality are temporally isolated, such that they cannot delay the activities of each other. Given the ever increasing computational demand, the next envisioned step is the deployment of mixed-criticality applications on multi-core platforms. However, this is challenging because multi-core platforms feature shared resources, such as last-level caches and memory interconnects. Concurrently executed applications (with potentially different safety criticality) can delay each other due to contention on these resources. Eliminating or bounding the temporal effects of such interference is not trivial due to the uncertainty with respect to the occurrence of resource accesses in time and the state of the resources.

In this thesis, we address challenges related to the development of mixed-criticality multi-core systems. The main contributions can be summarized as follows:

- We propose scheduling policies for efficiently exploiting the computing power of multicores, while preserving temporal isolation among applications with different safety criticality levels.
- We combine these policies with design optimization methods for minimizing interference among applications with the same criticality level.
- We proposes analytic and state-based approaches for bounding the delays that concurrently executed tasks experience due to contention on shared resources.
- We demonstrate how the proposed scheduling policies can be deployed on a state-of-the-art many-core platform in a way that enables the provision of real-time guarantees at design time and ensures an efficient resource utilization at runtime.

Zusammenfassung

Eingebettete Systeme finden immer mehr Anwendung in sicherheitskritischen Anwendungsbereichen, wie zum Beispiel in der Luftfahrt oder Automobilindustrie. Aufgrund der potentiellen Auswirkungen eines Fehlers auf Mensch und Umwelt, müssen solche Systeme typischerweise zertifiziert werden. Die Korrektheit hängt nicht nur vom Erfüllen funktionaler Spezifikationen ab, sondern auch davon, dass strikte zeitliche Auflagen eingehalten werden. Dazu müssen bei der Entwicklung Echtzeit-Garantien abgegeben werden können, typischerweise in der Form analytisch hergeleiteter Obergrenzen der schlechtesten Ausführungszeit.

Heutzutage werden bei sicherheitskritischen Systemen oft Funktionalitäten verschiedener Kritikalität in einer gemeinsamen eingebetteten Plattform integriert, um damit Größe, Gewicht, Energieverbrauch und Kosten zu minimieren. Gemäß aktueller industrieller Praxis müssen dabei Anwendungen unterschiedlicher Kritikalität zeitlich isoliert werden, um gegenseitige Verzögerungen auszuschließen. Mit dem immer größer werdenden Bedarf an Rechenleistung ist es voraussehbar, dass in Zukunft für solche Systeme auch Multicore-Plattformen eingesetzt werden. Eine besondere Herausforderungen stellen dabei die auf Multicore-Plattformen vorhandenen geteilten Ressourcen dar, wie zum Beispiel last-level Caches und Speicherbusse. Durch Konkurrenz bei der Nutzung dieser Ressourcen ist es möglich, dass parallel ausgeführte Anwendungen (mit möglicherweise unterschiedlicher Kritikalität) gegenseitig verzögert werden. Die Eliminierung oder Begrenzung solcher zeitlicher Interferenzen ist nicht trivial, wegen der Unsicherheit bezüglich des Zeitpunkts der Zugriffe und des Status der Ressourcen.

Diese Arbeit befasst sich mit den Herausforderungen bei der Entwicklung von Multicore-Systemen mit unterschiedlichen Kritikalitäten. Der Hauptbeitrag kann wie folgt zusammengefasst werden:

- Wir schlagen Ablaufplanungsmethoden für die effiziente Nutzung der Rechenleistung von Multicore-Systemen vor, welche die zeitliche Isolation von Anwendungen unterschiedlicher Kritikalität weiterhin gewährleisten.
- Wir kombinieren diese Strategien mit Methoden zur Entwurfsoptimierung, um die Interferenz zwischen Anwendungen unterschiedlicher Kritikalität zu minimieren.
- Wir schlagen analytische und zustandsübergangsbasierte Vorgehen vor, um die durch Konkurrenz um gemeinsame Ressourcen entstehende Verzögerung parallel ausgeführter Tasks zu beschränken.

- Wir demonstrieren wie die vorgeschlagenen Ablaufplanungsmethoden auf hochmodernen Manycore-Plattformen eingesetzt werden können, um Echtzeit-Eigenschaftem während des Designs zu garantieren und eine effiziente Nutzung der Ressourcen zur Laufzeit sicherzustellen.

Acknowledgments

Learning how to conduct academic research and writing a PhD thesis is a challenging procedure. Here I would like to extend my sincere gratitude to all those who helped me survive professionally and personally throughout this great adventure.

First and foremost, I would like to thank Prof. Lothar Thiele for offering the opportunity to write the thesis in his group and for being extremely supportive throughout the last five years. Our discussions, of which several conclusions are included in this thesis, taught me a lot not only about the design of embedded systems, but also about asking the right questions in research and working effectively towards addressing them. I greatly appreciate the guidance and feedback during all stages of my research, even if this entailed tedious requests from my side, like proof-reading the same part of a paper for the third time or cross-checking an outdated Matlab script with real-time calculus operations.

I sincerely thank Prof. Rodolfo Pellizzoni for accepting to review my thesis and for the interesting and enlightening discussions about shared memory interference in the first year of my graduate studies. His work has been a source of inspiration in the years that followed.

Furthermore, I would like to thank all researchers who directly or indirectly contributed to the technical content of this thesis: Pengcheng Huang, Nikolay Stoimenov, Rehan Ahmed, Davide Bartolini, Benoit Dupont de Dinechin, Kai Lampka, Rofolfo Pellizzoni, Zheng Pei Wu, Andreas Tretter, Lukas Sigrist, Andres Gomez, Lars Schor, Pratyush Kumar. Our discussions and joint work had a significant impact on this dissertation.

I would like to express my gratitude to our partners in the Certainty project for our collaboration towards building a common understanding about the design of certifiable mixed-criticality multi-core systems. Special thanks to Petro Poplavko, Dario Socci, Marius Bozga, Saddek Bensalem for our close collaboration during the development of the DOL-BIP-Critical tool chain and for inviting me to Verimag in April 2015; to Madeleine Faugere and Sylvain Girbal for sharing their experience concerning the avionics certification procedure and for kindly offering the Flight Management System software which is used as case study in this thesis; to Benoit Dupont de Dinechin and Amaury Graillat for providing insights into the architecture and programming of the Kalray MPPA-256 processor; to Pranav Tendulkar, Ioannis Galanommatis and Oded Maler for providing access to their runtime environment for streaming

applications on the MPPA-256 and for interesting discussions on memory benchmarking.

During my graduate studies, I had the opportunity to supervise several semester and Master theses, which helped me gain a practical insight into several aspects of the development of mixed-criticality and more generally, embedded systems, and triggered a lot of interesting new research questions. For this and all important lessons I learnt from our collaboration, I sincerely thank Felix Wermelinger, Stefan Draskovic, Neil Dhruva, Lukas Sigrist, Bartlomiej Grzeskowiak, Sujay Narayana, Fabian Dalbert, Akos Pasztor, Matthias Baer and Michael Walter.

I owe a lot to all my current and former colleagues at the Computer Engineering laboratory for educative and inspiring research-oriented discussions, but also for the great working atmosphere throughout the years. Special thanks go to my office-mate Pengcheng Huang for our successful teamwork, for the long discussions about the interpretation of mixed-criticality systems and for the insights I got into Chinese culture, which helped me change a bit my view on the world and people. Many thanks to Lars Schor and Pratyush Kumar for their friendship and advice whenever I encountered difficulties in the first years of my studies. And of course, many thanks to Felix Sutton, Devendra Rai, Balz Maag, Roman Lim, Lukas Sigrist for the creative cake engineering hours; to Romain Jacob and Matthias Meyer for social/political discussions over lunch; to Philipp Miedl and Federico Ferrari for offering excursion opportunities to Bern; to all members of the MPSoC group for summer BBQs at the lake.

Last but not least, I would like to thank my parents Theodoros and Irene, my brother Nikos and my partner Thomas for being inexplicably patient and coping with my moody behavior and limited free time over the last years. I can confirm with high confidence that this thesis would not exist without their support and encouragement. I hope to be able to do something equally important for you one day.

The work presented in this thesis has been partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 288175 (CERTAINTY project). This support is gratefully acknowledged.

Contents

List of Figures

List of Tables

1

Introduction

Embedded and cyber-physical systems, which facilitate sensing of environmental parameters, real-time processing and control of physical processes through actuating devices, are used nowadays in almost every aspect of personal and industrial life [Lee08, Sta08, RLSS10]. A special class of such systems is employed in safety-critical application domains, such as avionics, automotive, medical devices, defence and factory automation. Since a potential failure in such applications can have catastrophic consequences on human lives, the natural environment and infrastructure, the design of safety-critical embedded systems must be provably correct in terms of functionality and timeliness. Timing guarantees in the form of worst-case execution time and communication time bounds need to be provided at design time and enforced at runtime. As an example, consider a flight management system which is responsible for determining the current location of an aircraft based on sensor inputs and for computing the trajectory that guides the auto-pilot based on a predetermined flight plan and pilot directives. According to safety specifications, a pilot directive must be processed so as to trigger corresponding changes to the online computed trajectory within 200 ms [DFG+14]. Apparently, correctness in such a system depends not only an the correct implementation of the required functionality, but also, with equal importance, on the ability to invariably satisfy stringent timing requirements like the above.

In recent years, industrial research for safety-critical embedded systems explores two main trends. On the one hand, the advances in performance of embedded computing platforms and the wish to reduce size, weight and power of computing elements promotes the practice of integrating multiple applications, with potentially different requirements

in terms of safety, in a common platform. This integration has led to the design of so-called *mixed-criticality systems* [BD16]. On the other hand, the ever-increasing computational requirements of safety-critical systems and the prevalence of multi-core platforms in the electronics market makes the exploitation of *multi-core architectures* an attractive design choice for the next-generation safety-critical embedded systems.

Although the combination of the two trends, namely the deployment of mixed-criticality applications on multi-core platforms, seems a very promising solution in terms of performance, cost, size, weight and power, it is not yet commonly applied. The main challenge lies in the shared utilization of non-computational platform resources such as last-level caches and memory buses in the majority of commercial multi-core platforms. This undermines timing predictability because concurrently executed applications (with potentially different safety criticality levels) can delay each other due to contention on the shared resources. Providing timing guarantees in the presence of such interferences is not trivial. This is why existing government regulations and certification standards for safety-criticality applications classify multi-core platforms as "highly complex" systems and do not yet encourage their exploitation [PDK+15].

This thesis addresses existing challenges that are associated with the implementation of mixed-criticality applications on modern multi-core platforms. It presents scheduling methods for efficiently exploiting the computing power of multicores, while eliminating interference among applications with different safety criticality levels, and design optimization methods for reducing interference among applications with the same criticality. Moreover, it proposes different approaches for bounding the delays that concurrently executed tasks experience due to resource contention. Finally, it demonstrates how the proposed scheduling policies can be deployed on a state-of-the-art commercial platform, in a way that enables the provision of timing guarantees at design time and ensures an efficient resource utilization at runtime.

1.1 Trends in Embedded System Design

In this section, we review two current trends in the design of embedded systems in safety-critical domains, which form the basis for this thesis.

1.1.1 Mixed-Criticality Systems

The design of safety-critical applications, e.g., in the avionics and automotive, is typically subject to certification due to the impact of potential failures on human lives and the environment. Existing certification standards classify applications into different criticality levels which express the required protection against failure. For instance, the

DO-178C standard for avionics defines five Design Assurance Levels (DAL A to DAL E) [RTC12], the ISO 26262 standard for automotive defines four Automotive Safety Integrity Levels (ASIL 1 to 4) [iso11], and a similar classification exists in EN 50129 for railway [CEN03] and in IEC 61508 for industrial control [Com10]. According to DO-178C, erroneous behavior of a DAL-A application might cause an aircraft loss, whereas erroneous behavior of a DAL-D application might cause inconvenient or suboptimal operation in the worst case [Ves07]. Driven by the severity of potential failure consequences, the criticality level of an application influences the design requirements and the rigor and cost of the development, testing, validation and certification procedures. For instance, providing timing guarantees for high-criticality applications relies typically upon very conservative tools and formal approaches, while for low-criticality applications it may suffice to use measurement-based worst-case execution time estimations.

An increasingly popular trend in the design of safety-critical systems concerns the integration of applications with different safety criticality levels in a common hardware platform. For instance, in unmanned aerial vehicles (UAVs) a single platform can be used to host safety-critical functionalities related to the flight of the UAV and mission-critical functionalities related to its designated mission [BEG16]. The development of so-called *mixed-criticality systems* enables a significant reduction in the cost, size and weight of embedded electronics as well as a reduction in power dissipation which is particularly important in mobile systems.

Certification standards require that safety applications of different criticality levels are *separated* or *isolated* so that they do not interfere with each other. Isolation reduces software complexity and certification effort and cost, since several applications are integrated in a platform, yet each of them can be developed and evaluated according to its own criticality level [PDK+15]. System designers typically rely on partitioning mechanisms on hardware and operating system level to achieve isolation. For instance, in avionics, the ARINC 653 standard clearly defines *spatial* and *temporal* partitioning mechanisms that "Integrated Modular Avionics" systems need to implement [ARI03, Rus99]. Spatial partitioning enforces isolation of address spaces, such that one partition cannot corrupt the memory of another partition, and can be achieved e.g., by designated memory management units (MMU) which check if the active partition has permission to access the requested memory address [HRK12, PDK+15]. Temporal partitioning enforces isolation in time, such that different partitions cannot affect the execution of each other, e.g., by using the same resource (processing core, memory, interconnect, cache) simultaneously. Partitioning techniques in hardware and software level consist a well-studied topic and have been applied to industrial mixed-criticality systems on single-core processors over the

last two decades [PD14]. However, as processors become more complex, e.g., by implementing multi-core architectures, ensuring the property of isolation becomes increasingly difficult as will be discussed later in Section 1.2.

1.1.2 Multi-Core Embedded Platforms

Following the breakdown of Dennard scaling and due to the resulting inability to increase clock frequencies significantly, in the last 10 years embedded processor manufacturers have mainly focused on multi-core architectures as a means to continue improving computational performance [EBSA+11]. Multi-core architectures enable workload distribution across several processing cores which do not need to run at maximal frequency, thus reducing the peak heat and power dissipation. To meet the ever-increasing computational demand of embedded systems, the number of processing cores on such architectures has been increasing at a rapid pace. A characteristic example for this demand can be identified in today's smart phone devices which often employ quad-core or octa-core processors, such as Qualcomm Snapdragon 820 [Qua] or Samsung Exynos 8890 [Sam] which are powered by a single battery and offer almost equivalent computational performance as general-purpose computers.

Safety-critical industries also face an increasing pressure for migrating to multi/many-core platforms. To illustrate the benefits, consider for instance modern cars like Lexus LS460 (release series 2006) which feature already more than 100 inter-connected Electronic Control Units (ECUs) [Tak12]. Adding a multitude of advanced applications to facilitate autonomous driving in next-generation cars becomes increasingly difficult due to size, weight and cost limitations. Employing multi-core architectures offers the opportunity for consolidating several applications in a common platform, thus reducing the size requirements, the car weight and production cost through a reduced amount of processing units and wiring.

In this thesis, we focus our interest on homogeneous shared-memory multi/many-core processors. All processing cores are identical with respect to clock frequency and architecture, and share (symmetric) access to a main memory. This abstract model describes, for example, Intel Core i7 [int], Freescale P4080 [p40] and the ARM Cortex-A processor family [ARM]. In shared-memory processors, one way to achieve scalability in terms of core count is by applying a hierarchical cluster-based pattern. According to this, processing cores are grouped into (homogeneous shared-memory) clusters with a private address space, and inter-cluster communication is achieved through a network-on-chip. This paradigm is exploited in several state-of-the-art commercial platforms, such as the Intel Single-chip Cloud Computer (SCC) with

24 dual-core clusters [HDH+10], the STMicroelectronics STHorm/P2012 processor with four 16-core clusters [MBF+12] and the Kalray MPPA-256 processor with 16 16-core clusters [dDAB+13, dDvAPL14] and the aspiration to scale to 1,024 cores (64 16-core clusters) within the next years. Being able to exploit the computational performance and the relatively low power dissipation of such architectures in safety-critical embedded systems would indeed mark the start of a new era, with unprecedented possibilities.

1.2 Challenges in Mixed-Criticality Multi-Core System Design

Combining the previously discussed trends seems an attractive solution for the next-generation safety-critical applications. Indeed, it could facilitate the integration of multiple functionalities in a small number of multi-core chips, thus increasing computational performance while size, weight and power dissipation of embedded electronics get overall reduced. However, the deployment of mixed-criticality multi-core systems remains largely a research topic, without industrial applicability yet. Below we review some of the challenges that hinder the realization of mixed-criticality multi-core systems.

1.2.1 Isolation among Criticality Levels

As discussed in Section 1.1.1, the integration of mixed-criticality applications on single-core processors has become possible based on the property of separation or isolation. Industrial guidelines specify hardware and software mechanisms that can be applied to achieve temporal and spatial partitioning, such that any two partitions cannot interfere by delaying the activities or by corrupting the address space of each other [RTC12, ARI03, Rus99]. However, as the processors become more complex, e.g., by integrating multiple cores on a system-on-chip, ensuring the property of isolation is no longer trivial. The main difficulty lies in the common fact that the cores of commercial multi-core platforms share access to on-chip resources, such as last-level caches, interconnects (e.g., NoC) and memories. If two partitions with applications of different criticality level are executed concurrently on a multi-core platform, temporal isolation is jeopardized by the potential contention on shared resources. For instance, it is possible that a task from the lower-criticality partition can delay a task from the higher-criticality partition by blocking the access of the latter to the main memory or by evicting its cache lines from the shared last-level cache.

To better illustrate the difficulty in ensuring isolation among criticality

levels in the presence of shared resources, we list below potential sources of inter-core interference which exist in most of the commercial-off-the-shelf multicores.

1) Shared Last-Level Cache. We distinguish two types of inter-core interference on a shared cache: (i) storage interference due to contention for allocation of memory blocks and (ii) temporal interference due to contention for access to the shared cache.

Storage Interference. Mancuso et al. [MDB+13] list three types of cache interference that fall into this category: *intra-task interference*, occurring when a task evicts its own cached blocks, *cache pollution* caused by asynchronous Kernel Mode activities, such as interrupt service routines and deferrable functions, and *inter-core interference*, occurring when two tasks running on different cores evict each other on a shared cache. The first two types are common in single-core systems too, whereas the last type is relevant only for multicores.

The contents of a shared cache at any point depend, therefore, on the relative time order of cache accesses and the requested data from the cores as well as the cache replacement policy and the implemented coherence protocol (if any). A special case of inter-core interference due the cache coherence protocol is known as *false sharing* [TS12]. This happens when tasks on different cores write to a shared cache line, but at different locations, in a set-associative cache. Whenever a core modifies its location, the cache coherence protocol marks the cache line dirty, so when the other location(s) get(s) a read/write request, a reload of the cache line from memory is forced. Frequent (unnecessary) reloads of the shared cache line can result in severe time penalties for the tasks, whose data remain actually coherent, similarly to frequent reloads caused by mutual data evictions among cores. Kim et al. [KKR13] have shown empirically that storage interference can increase a task's response time by up to 40% on a quad-core processor with a shared L3 cache. Note that the term response time differs from execution time, as the former accounts for the delays caused by interference on shared resources, while the latter does not (it is derived for task execution in isolation).

Temporal Interference. When a shared cache does not provide enough ports to support parallel access for all cores, blocking effects appear. Namely, execution on a core can stall until another core releases a cache port. The delay in task execution depends on the access patterns of all cores to the cache, the arbitration policy on the cache ports and the state (contents) of the cache.

2) Shared Bus/Memory. We consider synchronous (blocking) memory accesses, such as the ones caused by cache misses, which cause execution on a core to stall until the access is completed. In architectures where the

cores (caches) are connected to the global memory through a shared data bus (e.g., ARM Cortex A17 [ARM]), tasks from different cores interfere with each other while trying to access the bus simultaneously. Depending on the interfering access patterns and the bandwidth allocation policy of the bus, the data fetch time varies since the requesting task cannot exploit the full memory bandwidth, like in a single-core system. Similarly, in architectures where the cores have private paths to the memory and arbitration among access requests from different cores is handled at the memory controller (e.g., Kalray MPPA-256 [dDvAPL14]), tasks can block each other upon accessing the memory. Namely, every time a task issues an access request, a delay must be accounted until pending accesses from other cores are completed and the memory controller arbiter grants the access. This delay depends on the access request patterns from other cores and the exact arbitration policy of the memory. Kim et al. [KdNA+14] empirically observed a 12x increase in the response time of a PARSEC benchmark when the benchmark was executed in parallel to memory-intensive tasks on a quad-core Intel Core i7 with a shared DRAM.

3) Shared I/O Peripherals. Similar to the memory, synchronous accesses to mutually-exclusive shared platform resources, such as DMA controllers for access to peripheral devices or external memories, introduce delays to the task execution. Bounding these delays is non-trivial as they depend on the access request patterns and data demand of all other cores and the arbitration policy of the shared resource. Note that arbitration policies for such resources are usually not documented for commercial multicores. Pellizzoni et al. [PBB+11] have shown empirically that I/O traffic can increase a task's response time by more than 60% even on a single-core processor.

A more detailed presentation of sources of inter-core interference that jeopardize the property of temporal isolation on commercial multicores can be found in [KNP+14]. Eliminating (or safely bounding) the temporal effects of such interferences, while still exploiting hardware parallelism, is not trivial. This is the reason why the current industrial practice in avionics is to disable all but one core on multi-core processors [KNP+14]. Authorities such as the European Aviation Safety Agency (EASA) and the U.S. Federal Aviation Administration (FAA) only recently started considering the utilization of multi-core (mainly dual-core) processors for mixed-criticality applications [EAS11, Cer14b]. The main concern for certification is exactly the enforcement of temporal isolation in the presence of shared platform resources, which leads to a classification of multi-core processors as highly complex systems for which additional means for isolation are required compared to single-core processors [PDK+15].

Based on the above discussion, the first question that is considered in this thesis is the following:

How can resource-sharing multicores be utilized for mixed-criticality applications in a way that ensures both, temporal isolation among different criticality levels and efficient resource utilization (hardware parallelism)?

We address this question in Chapter 2 by proposing a scheduling model which enforces temporal isolation by allowing only applications of the same criticality level to be executed on the multi-core platform at any time. In this way, the computing power of the multicore can be fully exploited and interference on shared platform resources is constrained to tasks with the same criticality level. The notion of temporal partitions on single cores [ARI03] is now lifted to global partitions on multicores, with exclusive access to all platform resources. Chapters 2 and 3 present scheduling policies that comply with the introduced model and propose design optimization approaches for efficient resource utilization.

1.2.2 WCRT Estimation under Resource Contention

Even if interference among applications with different criticality levels is eliminated, thus fulfilling the requirement for isolation, the problem of bounding the temporal effects of intra-criticality interference remains open if one wishes to maximally utilize the computational resources of a multi-core platform. Bounding worst-case execution times on single-core processors is a well-studied problem with mature solutions and available tools, e.g., based on formal static analysis and abstract interpretation [Inf]. On the contrary, bounding worst-case response times (WCRT) of concurrently executed tasks on multicores while accounting for resource contention is a relatively new and ongoing research topic, which arose with the advent of multi-core processors [PSC+10, SNE10, DAN+11].

Based on the discussion of Section 1.2.1, two (but not the only) factors that can significantly affect the response times of concurrently executed tasks on a shared-memory multi-core architecture are the mutual data evictions on the shared caches and the arbitration policy of mutually-exclusive shared resources, such as the memory. In this thesis, we focus on the second factor since we restrict our interest to multicores with private caches. The challenge in bounding the temporal effects of the memory interference lies mainly in the high uncertainty with respect to (i) the degree of time overlap for the executions of any two tasks (which depends on synchronization among cores and the employed scheduling policies on the cores), (ii) the access patterns of the tasks to the shared resources (which depends on the program paths and micro-architectural states) and (iii) the accessed memory locations (which depends on the program state). For upper-bounding the delays on the shared memory, one needs, additionally, accurate knowledge of all components of the memory path (buses, controllers) and the related arbitration policies. This information is often not disclosed by the vendors. All above factors make

the derivation of safe and tight WCRT bounds for concurrently executed tasks highly challenging [DAN+13].

Even if one possesses accurate models of all tasks' execution and access patterns as well as the platform architecture, enumerating and analyzing all possible interference scenarios is a time-consuming approach, which becomes intractable as the number of cores and executed tasks increases [LYGY10]. An alternative approach is to rely on abstract models to specify the task's execution and access patterns, such as event arrival curves [LBT01], and on pessimistic assumptions concerning the state of the resource arbiter(s). Such approaches to WCRT derivation are known to scale better with an increasing number of cores, however they often lead to very pessimistic results. The analysis pessimism, which typically increases with the system size, can lead to resource over-provisioning and hence, inefficient utilization of the platform.

Based on the above discussion, the second question that is considered in this thesis is the following:

How can we estimate safe and tight worst-case response time bounds of concurrently executed tasks under resource contention scenarios using methods that scale efficiently with the number of cores?

We address this challenge by proposing methods for bounding and reducing the temporal effects of inter-core interference upon accessing the shared memory. Chapter 3 presents an analytic approach to WCRT estimation for cluster-based architectures such as the Kalray MPPA-256 [dDvAPL14], in which memory accesses can be delayed either by accesses of concurrently executed tasks on different cores or by incoming traffic from a network-on-chip. We additionally propose optimization methods for partitioning task to processing cores and data to memory banks, such that interference on the memory path is minimized. Chapter 4 explores alternative approaches to WCRT estimation which are based on state-based system models with timed automata [AD90] and exhaustive model checking. The computational complexity of model checking is alleviated by the adoption of a predictable execution model with dedicated computation and memory access phases [PSC+10, PBB+11] and by abstractly representing memory access patterns from several cores with arrival curves from real-time calculus [TCN00]. Our methods for bounding WCRT are applicable to multi-core architectures which fulfill the property of full timing compositionality as defined in [WGR+09], i.e., in which execution and memory accessing times can be safely decoupled. The Kalray MPPA-256 cores, for example, fulfill this property.

1.2.3 Efficient and Timing-Predictable Implementation of Mixed-Criticality Scheduler

In safety-critical domains, the software employed for scheduling and resource management, including for example the operating system, software-based partitioning mechanisms, hypervisors, is itself subject to verification. Thus, it needs to be provably correct in terms of functionality and timeliness in the same sense as real-time applications [PBA+14]. In the railway domain, PikeOS [Fis13] is the first operating system for multicores which has been successfully certified. According to the authors of [PDK+15], a similar achievement in other safety-critical domains, such as avionics, may be much harder to realize due to the requirement to guarantee availability at very high assurance levels and to prove software determinism as defined in [Cer14b].

From the above requirements it becomes obvious that any proposed scheduling policy for mixed-criticality multi-core systems should be also evaluated on the basis of whether it can be implemented on existing platforms in a timing-predictable way, i.e., with bounded runtime overheads. For an efficient resource utilization, it is also crucial that the overhead bounds are low. However, most mixed-criticality scheduling policies in research literature (for a survey, see [BD16]) rely on computationally expensive runtime mechanisms, such as inter-core synchronization, parallel execution time monitoring, dynamic schedule adaptations, execution throttling or termination. Sigrist et al. [SGH+15] have shown that the runtime overheads of a mixed-criticality multi-core scheduler, implemented in the user space of the Linux operating system, can have a detrimental effect on real-time schedulability. In their experiments on Intel Xeon-Phi (8 cores), 50% of theoretically schedulable task sets experienced missed deadlines at runtime due to unaccounted scheduling overheads.

Based on the above discussion, the third question that is posed in this thesis is the following:

> *How can we implement mixed-criticality scheduling primitives with bounded and low overhead on existing multicores?*

We consider implementation aspects of mixed-criticality scheduling with focus on the Kalray MPPA-256 Andey processor [dDvAPL14] in Chapter 5. By applying global temporal partitioning, as proposed in Chapter 3, we evaluate alternative implementations of the required scheduling primitives for temporal isolation and show how to empirically bound their runtime overheads on the MPPA-256. We also show how the measured overheads can be integrated into the timing analysis of the system at design time in order to provide real-time guarantees for the concurrently executed applications.

1.3 Aim of this Thesis

With this work, we aim to defend the following thesis:

> *By introducing novel scheduling policies, design optimization approaches, and methods for analyzing the worst-case response time of concurrently executed tasks under resource contention, it is possible to deploy mixed-criticality applications on timing-compositional multi-core processors in an efficient and timing-predictable manner.*

1.4 Thesis Outline and Contributions

This thesis proposes novel approaches for scheduling, design optimization, worst-case response time analysis and deployment of mixed-criticality applications on multi-core architectures. The developed approaches aim (i) to ensure temporal isolation among applications with different criticality level in the presence of shared resources, (ii) to reduce and bound the effects of resource interference on the response time on concurrently executed tasks, (iii) to demonstrate that a timing-predictable and efficient implementation of mixed-criticality scheduling on existing many-core platforms is feasible. The integration of the proposed approaches yields a design flow for the development of mixed-criticality multi-core systems, as illustrated in Figure 1.1. In the following, we present the main contributions of the thesis.

Chapter 2: Eliminating Inter-Criticality Interference
In order to address the challenge of preserving temporal isolation among applications with different safety criticality levels on a resource-sharing multicore, we introduce a novel scheduling model called Isolation Scheduling. Isolation Scheduling (IS) partitions temporally a platform, such that only tasks with the same criticality level can be executed and utilize the on-chip resources at any time. In this way, inter-criticality temporal interference is avoided by construction.

For a better understanding and evaluation of the limitations of the IS model, we propose a partitioned, hierarchical scheduling policy for sporadic task sets, IS-Server. IS-Server partitions the platform according to a time-triggered schedule and applies earliest-deadline-first (EDF) with a fixed task to core mapping within each partition. For the mapping of tasks to processing cores, we present and compare optimization formulations and heuristic approaches with the goal of maximizing schedulability. We evaluate the schedulability loss due to the IS constraint (exclusive platform utilization by tasks of equal criticality) theoretically and empirically, by introducing a speedup bound and by performing extensive simulations with synthetic task sets which enable a comparison between IS-Server and state-of-the-art scheduling policies that do not

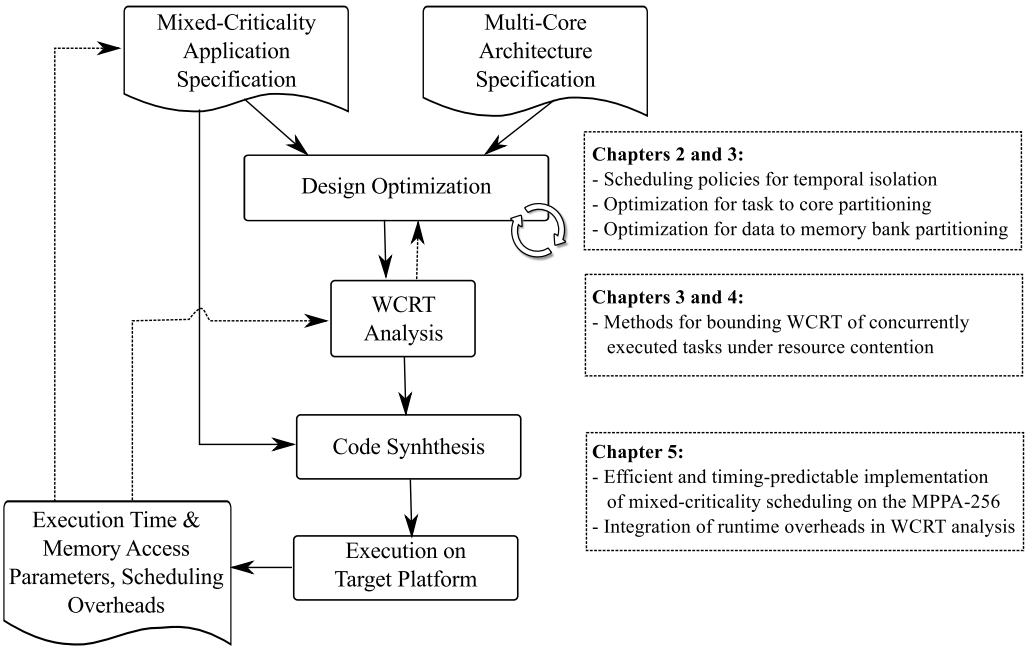


Figure 1.1: Design flow for the efficient and timing-predictable deployment of mixed-criticality applications on shared-memory multi-core architectures. The main contributions of this thesis are listed on the right side of the figure.

enforce the IS constraint. Finally, we conduct a comparative study among existing scheduling policies that comply with the IS model, showing that IS-Server achieves comparable or better schedulability, while being applicable under more general task assumptions, e.g., to task sets with random (not equal or harmonic) deadlines.

Chapter 3: Bounding Intra-Criticality Interference

In order to bound the effects of intra-criticality interference under the Isolation Scheduling model, we introduce a combined analysis of computing, memory and communication scheduling. The analysis is applicable to cluster-based many-core architectures, such as the Kalray MPPA-256, in which accesses to shared cluster memory can be delayed either by simultaneous accesses from other tasks or by incoming traffic from a network-on-chip (NoC) which is used for inter-cluster communication and access to external memories. For such architectures, we propose an Isolation Scheduling policy which partitions the platform according to a flexible time-triggered and criticality-monotonic scheme (FTTS). Unlike IS-Server, FTTS does not allow dynamic task preemptions in order to reduce the complexity of interference analysis. We present a worst-case response time analysis under FTTS, which accounts for the delays due to contention on the shared memory path and due to NoC data transfers. For the memory interference analysis, we explicitly consider the internal multi-bank structure of the cluster memory and assume a given data to memory bank mapping, such that interference among tasks which

access data in disjoint banks can be neglected. For the NoC analysis, we assume that NoC flows are statically routed and regulated at the source node. The worst-case data transfer times can be analyzed, then, using the real-time calculus [TCN00]. Additionally, we propose optimization approaches for the mapping of tasks to processing cores and of task data and instructions to memory banks, such that memory interference is minimized and the workload is distributed to the cores in a balanced way, leaving maximal idle time for incremental design (adding tasks later into the system). To demonstrate the applicability of the scheduling policy and the impact of the design optimization approaches, we use a case study based on a real-world avionics application and detailed architecture models based on the Kalray MPPA-256. Finally, we evaluate the efficiency of FTTS in finding schedulable solutions against state-of-the-art approaches that do not comply with the Isolation Scheduling model, showing that FTTS can outperform them in the special case of harmonic workloads.

Chapter 4: A Dedicated Execution Model for Tighter Interference Analysis

The previously introduced WCRT analysis is customized to the architecture model of Kalray MPPA-256 and similar platforms, in which the shared resources are arbitrated in-order, in a round-robin fashion. Extending the analysis to more complex arbitration policies that are common on commercial multicores, e.g., for DDR SDRAM controllers [KdNA+14], is not trivial. An extension or adaptation of our analytic method would need to rely on over-approximations due to the inherent difficulty in modelling the dynamic state of the arbiters. However, such over-approximations typically lead to very pessimistic WCRT estimates, and in turn to resource over-provisioning and platform under-utilization. To avoid this situation, we propose a novel state-based analysis method for tightly bounding intra-criticality interference under the Isolation Scheduling model. For this, we model all possible execution and memory access patterns of the processing cores (given a fixed task to core mapping and a fixed sequential schedule on each core) along with the respective arbitration policy using timed automata [AD90]. For the arbitration, we consider dynamic (first-come-first-serve or round-robin), static (time multiplexing) or hybrid (FlexRay [fle]) policies. We, then, employ a timed model checker [BDL04, BY04] which exhaustively explores all feasible memory interference scenarios and allows to compute a safe and tight WCRT estimation for every task.

This method apparently can become very time-consuming for a large number of cores and concurrently executed tasks. To reduce the complexity of model checking, we propose (i) a dedicated task execution model and (ii) an abstract representation of the execution and memory access patterns of some cores based on arrival curves

from the real-time calculus [TCN00]. On the one hand, the dedicated execution model restricts accesses to particular, well-defined phases of a task [PSC+10, PBB+11], thus reducing the non-determinism with respect to their occurrence in time. On the other hand, the abstract representation of the execution and access behavior of several cores and the incorporation of the arrival curve into the timed automata system model improves significantly the analysis scalability, by making the number of timed automata in the model independent of the number of cores or concurrently executed tasks in the system. We present a novel approach for computing such an arrival curve and show how this can be integrated into the timed automata model. Finally, we evaluate the accuracy (tightness) and scalability of our state-based WCRT analysis approach against architectural simulation and state-of-the-art analytic approaches, using realistic automotive benchmarks. The experiments confirm that our analysis yields safe WCRT estimates and can scale efficiently to a large number of cores, without compromising the accuracy of the WCRT bounds.

Chapter 5: Deployment of Mixed-Criticality Scheduling on a Multi-Core Architecture
In the last chapter, we address the third challenge of Section 1.2, namely the efficient and timing-predictable implementation of a mixed-criticality scheduler on commercial multicores. For this, we develop one of the first runtime environments for mixed-criticality multi-core scheduling. The runtime environment implements the FTTS scheduling policy of Chapter 3 and is deployed on the Kalray MPPA-256 (one cluster). We propose different mechanisms to implement the necessary scheduling primitives and show how to experimentally bound their runtime overhead. We then propose a WCRT analysis method which extends the respective method of Chapter 3 by modelling an additional, previously unknown source of interference on the memory path and by accounting for the runtime overheads of the scheduler. An empirical evaluation with industrial-representative benchmarks from avionics and signal processing enables us to validate the adherence to the analytically derived WCRT bounds during runtime. Additionally, we demonstrate a maximum achievable utilization of 73.6% (analytically guaranteed) or 78.9% (no deadline misses in practice) on the 16 cores of an MPPA-256 cluster. This effective utilization, which is significantly higher than previously reported results [SGH+15, PMN+16, BDN+16] suggests that FTTS and similar policies based on Isolation Scheduling can offer a viable solution for the efficient and predictable deployment of mixed-criticality applications on timing compositional multicores.

2

Eliminating Inter-Criticality Interference

Deploying safety-critical applications on multicores is challenging because tasks that are executed concurrently on different cores can interfere on shared resources according to unpredictable patterns. This complicates worst-case response time analysis and hinders the provision of real-time guarantees. To address this challenge, we propose in this chapter a scheduling model called Isolation Scheduling (IS). IS provides a framework to exploit multicores for safety-critical applications in which tasks are grouped into classes. IS enforces mutually exclusive execution among different task classes, thus avoiding inter-class interference by construction. In the context of mixed-criticality systems, IS enables elimination of inter-criticality interference, which is important for certification. Subsequent chapters of the thesis will present methodologies for bounding intra-criticality interference.

In this chapter, we propose and analyze a novel approach for Isolation Scheduling, IS-Server. This is a partitioned approach based on hierarchical server scheduling, which can be applied to sporadic, constrained-deadline task sets that are divided into task classes, and (in an extended form, MC-IS-Server) to mixed-criticality task sets. Through extensive simulations, we evaluate the proposed approach in terms of schedulability and runtime overhead and quantify the schedulability loss due to the isolation constraint. Moreover, we conduct a comparative study among state-of-the-art approaches that comply with the IS model, showing that (MC-)IS-Server can outperform existing approaches in terms of schedulability.

2.1 Introduction

Nowadays there is a constantly increasing gap between the requirements of safety-critical real-time applications and the guarantees that architectures of embedded processors can provide. On one hand, real-time applications need predictability in order to enable safe operation based on worst-case response time analysis. On the other hand, embedded processors increasingly feature a multi-core architecture with shared resources, such as last-level cache, memory bus and memory controller, in order to improve performance and computational efficiency. To exploit multi-core architectures, applications need to run jobs concurrently on different cores. However, shared resources undermine predictability, since jobs that run concurrently pay unpredictable performance penalties due to contention on shared resources.

Deploying safety-critical applications on multicores in an efficient, yet predictable manner is a challenging problem. Coarse-grained static partitioning in time and space, e.g., based on the DO-178C standard [RTC12] for avionics and the ISO 26262 standard [iso11] for automotive systems, is an established technique for single-core safety-critical systems, but it cannot be trivially extended to multi-core architectures. If strictly applied, it would allow only one job to be executed at any point in time. More fine-grained partitioning requires individual access control to each shared resource [SCM+14], which relies on special hardware and operating system support. Finally, the approach of finding a global schedule and bounding the contention on shared resources at any time is only feasible with knowledge of the detailed resource sharing behavior of all tasks, and it quickly becomes computationally intractable with an increasing number of cores and tasks [DAN+13].

In this chapter, we propose a scheduling model that we call Isolation Scheduling (IS). IS enables efficient scheduling of safety-critical applications on multi-core processors by exploiting hardware parallelism and shared resources. To make the problem more tractable, IS is based on the assumption that tasks are partitioned into *task classes* that have exclusive access to the processor and the platform resources. This way, interference on shared resources is greatly reduced, since inter-class interference is eliminated by construction and only intra-class interference needs to be considered. Well-established methods [NSE09, PSC+10, LYGY10, WKP13] can then be applied to bound the interference within each class. IS is particularly relevant in the context of mixed-criticality systems [Ves07, BD16], in which tasks are naturally grouped into classes of different safety criticality levels. Industrial standards [RTC12, iso11] require isolation of these classes in order to allow *independent certification* of criticality levels. IS guarantees that tasks of different criticality levels do not interfere on shared platform resources, and therefore, it allows independent certification as well as a much simplified intra-criticality

interference analysis.

Theoretical aspects of the IS model have been already studied by Huang et al. [HGA+15]. A global, preemptive scheduling policy based on fluid scheduling was proposed in order to derive speedup bounds for the IS model and explore its limitations. Although this policy provides a rigorous framework for analysis, it cannot be easily deployed on existing multicores, since it relies on very frequent task class switches, task migrations and preemptions. Such mechanisms make the implementation of scheduling prohibitively expensive in terms of runtime overhead. To address this shortcoming, we propose and analyze a novel scheduling approach for Isolation Scheduling. IS-Server and its extension to mixed-criticality systems, MC-IS-Server, are partitioned approaches based on hierarchical server scheduling, which can be applied to sporadic, constrained-deadline real-time task sets with distinct task classes. In IS-Server, virtual servers are responsible for scheduling the tasks of the respective classes (one server per class). The servers follow a time-triggered schedule, but within each server, tasks are scheduled in a preemptive earliest-deadline-first (EDF) fashion. All cores perform the same server schedule and only one server can be active at a time, thus guaranteeing mutual exclusion among task classes. We propose and compare approaches for partitioning the tasks to cores and dimensioning the time-triggered schedule of the virtual servers. Additionally, through extensive simulations we evaluate the new scheduling approach with respect to schedulability and runtime overhead. The results deliver a deep understanding of the IS model and corresponding scheduling techniques and suggest that the IS model is a useful and flexible abstraction for designing systems that require isolation among task classes.

Contributions. The main contributions of this chapter can be summarized as follows:

- We formalize the Isolation Scheduling model for eliminating inter-class (inter-criticality) interference of safety-critical applications on resource-sharing multicores.
- We present a novel hierarchical scheduling policy, IS-Server, which complies with the IS model. We propose and compare several approaches for partitioning tasks to processing cores aiming at maximizing schedulability under IS-Server, and provide an algorithm for constructing IS-Server schedules. We prove a speedup bound for enforcing inter-class isolation.
- We extend IS-Server and the respective analysis to the particular case of mixed-criticality scheduling, by introducing MC-IS-Server. We are among the first to analyze systems with more than two criticality levels, a case which is often neglected in literature due to its intrinsic complexity.

- We perform extensive simulations with synthetic task sets to evaluate the proposed approaches with respect to schedulability and runtime overhead (in terms of task class switches).
- We conduct a comparative study among five IS-compliant dual-criticality scheduling approaches. We show that MC-IS-Server performs at least equally well as existing approaches in terms of schedulability, while being applicable under more general task assumptions.

Outline. In the remainder of the chapter, Section 2.2 presents existing methods for achieving isolation in safety-critical multi-core systems. Section 2.3 presents our scheduling and task set model. Section 2.4 and 2.5 present the IS-Server scheduling policy and its extension to mixed-criticality systems. Section 2.6 presents the empirical evaluation of the scheduling policies in terms of schedulability and runtime overheads and the comparison to state-of-the-art IS-compliant policies. Finally, Section 2.7 summarizes the results of this chapter.

2.2 Related Work

In this section, we focus on task class isolation as commonly required for the certification of mixed-criticality systems. Initial research on mixed-criticality multi-core scheduling [LB12, KAZ11, Pat12] did not explicitly address interference among task classes when tasks contend for access to shared resources other than processing cores, thus implying that it can be bounded. For instance, Anderson et al. proposed different strategies (partitioned EDF, global EDF, cyclic executive) for scheduling different criticality levels and used a bandwidth reservation server for temporal isolation among criticality levels [ABB09, MEA+10]. In their scheduling framework, tasks with different criticality could run in parallel and, like in the previous works, the interference on shared platform was not explicitly bounded. However, interference analysis for multiple shared resources is a very challenging task. In fact, estimating response time bounds under contention may be even impossible for mixed-criticality systems because a certification authority for higher criticality tasks does not necessarily possess information on the behavior of lower criticality tasks that are co-hosted on the same platform. To address this challenge, Isolation Scheduling minimizes inter-criticality interference by construction.

More recently, researchers acknowledged the problem of inter-criticality interference and proposed mechanisms for criticality-aware arbitration of shared resources, with the objective of statically bounding interference from lower to higher criticality tasks. Yun et al. [YYP+12] and Flodin et al. [FLY14] proposed a software-based memory throttling

mechanism (with predefined [YYP+12] or dynamically allocated [FLY14] per-core budgets) to explicitly control interference on a shared memory controller. Yun et al. [YYP+13] introduced a memory bandwidth reservation scheme with online reclamation support, for isolation of applications on different cores. Paolieri et al. [PQnC+09] and Goossens et al. [GAG13] proposed hardware modifications to a shared memory controller for mixed hard and soft real-time systems. Hassan and Patel [HP16] introduced a dynamically reconfigurable requirement- and criticality-aware arbiter to shared memory buses and Cilku et al. [CFP14] combined a dual-layer bus arbiter with a hypervisor for bounded interference on the memory path. More criticality-aware memory controllers were proposed in [JQA+14, GAGC16]. Several works [RLP+11, WKP13, YMWP14, ETSE14, KBL+15] proposed partitioning data to disjoint DRAM banks in order to minimize inter-core interference on bank arbiters. Furthermore, Kim et al. [KWC+16] combined bank partitioning with shared last-level cache partitioning. Sha et al. [SCM+14] proposed a combination of aforementioned approaches envisioning the design of single-core equivalent systems, for which timing analysis is compositional, namely it can be performed locally on each core with static, safe bounds on the interference from all other cores. With regards to interference on networks-on-chip, Tobuschat et al. [TAED13] implemented virtualization and monitoring mechanisms for independence among mixed-criticality flows, while Tamas-Selicean et al. [TSPS12] exploited the TTEthernet protocol to achieve spatial and temporal isolation for mixed-criticality messages. Such mechanisms ensure bounded interference among criticality levels. However, they can suffer from poor flexibility, e.g., when the resource budgets cannot adapt dynamically to varying resource demand, and they often require hardware support which is not available in commercial-off-the-shelf platforms. With the IS model, we sidestep the need for fine-grained shared resource arbitration. The key idea is to only permit tasks of the same criticality level (i.e., from the same class) to execute concurrently. Based on this insight, the IS policies avoid resource interference among task classes, exploit multiple cores, and only suffer a limited schedulability loss to enforce temporal partitioning among task classes (we quantify this loss analytically and experimentally later in this chapter).

It is worth mentioning that even though Isolation Scheduling does not allow parallel execution of more than one task class, it cannot completely eliminate inter-class interferences that exist due to the state of the shared resources. For instance, pending accesses to a shared memory that were requested from a task class but not served within the task class execution interval can affect/delay future memory accesses of another task class. Similarly, tasks of one task class can "pollute" a shared cache (evict cache lines required by another class) and cause a burst of cache misses at the beginning of the execution

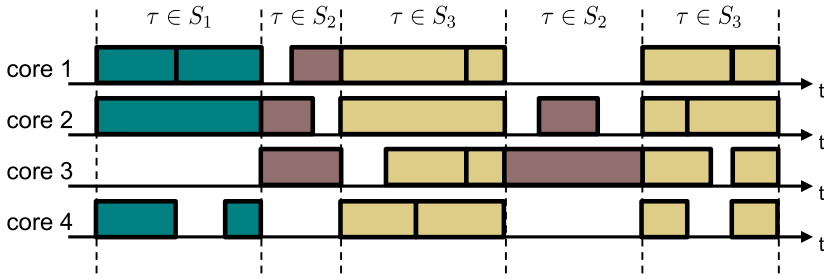


Figure 2.1: Example IS schedule with three task classes S_1, S_2 and S_3. Vertical lines mark the synchronous switching between task classes on all cores.

interval of the next task class. These implicit interferences need to be accounted for in the worst-case response time analysis of each individual task class. To minimize them, Isolation Scheduling can be combined with spatial partitioning approaches, such as memory bank privatization [RLP+11, WKP13, YMWP14] (here, for partitioning memory banks to task classes), shared cache partitioning [Kir89, CJDR00, HBHR11, KKR13, WHKA13, ADLD14, KWC+16] and locking [SM08, MDB+13].

2.3 System Model

This section[1] formalizes the Isolation Scheduling model (Section 2.3.1) and the considered task set models in this chapter (Section 2.3.2). Additionally, it provides an overview of a baseline mixed-criticality scheduling policy, upon which MC-IS-Server is built (Section 2.3.3).

2.3.1 The Isolation Scheduling (IS) Model

The Isolation Scheduling (IS) model partitions a hardware platform temporally among different task classes so that, at any time, only jobs of the same task class can utilize the platform resources. The model targets homogeneous multi-core processors with m identical cores that share on-chip resources.

For any task set, we assume that tasks are partitioned into K task classes $S = \{S_k \mid 1 \leq k \leq K\}$. If tasks need to satisfy timing constraints, we define the set of task classes S to be *IS-schedulable* if all timing requirements are met, while respecting the IS constraint of mutual exclusion between task classes. Formally, the IS constraint is defined as follows.

Definition 2.1. *A scheduling policy enforces the IS constraint if at any time t the executed tasks across all cores belong to the same task class. Namely, for any pair of tasks $\tau_i \in S_a$ and $\tau_j \in S_b$ that are concurrently executed at time t, it must hold that $S_a = S_b$.* □

[1] Parts of this section have appeared in publication [HGA+15], which consists joint work with Pengcheng Huang.

Note that the above definition can be extended also to inter-processor communication if this happens in an asynchronous manner, e.g., over a network-on-chip. In this case, any two active communication flows at time t must be initiated by tasks that belong to the same task class. In the following, we call a scheduling policy which enforces the IS constraint compliant with the IS model or simply IS-compliant.

For an illustration of how the model works, Figure 2.1 shows an example of an IS schedule. To enforce the IS constraint, all processing cores switch *synchronously* between task classes. The synchronous switch between two task classes can be triggered at runtime either dynamically or based on a static time-triggered pattern.

2.3.2 Task Set Model

The proposed scheduling policies in this chapter target sporadic real-time task sets. A special case of such task sets are sporadic mixed-criticality task sets. We present the considered task set models in the following.

Sporadic Real-Time Task Sets. The IS model supports real-time task sets, where each task sporadically instantiates single jobs. Given that the tasks are partitioned into K task classes $S = \{S_k \mid 1 \leq k \leq K\}$, each task τ_i in class S_k is characterized by a tuple (T_i, D_i, C_i), which defines the period of the jobs (minimal inter-arrival time), their relative deadline and worst-case execution time. We consider constrained-deadline tasks, where $D_i \leq T_i$. Eq. (2.1) defines the density δ_i and utilization u_i of a task τ_i:

$$\delta_i := C_i/D_i, \quad u_i := C_i/T_i \; . \tag{2.1}$$

A sporadic task set is *schedulable* under a given scheduling policy if all jobs of tasks τ_i receive enough execution time according to their execution bound C_i to complete by their deadlines.

Mixed-Criticality Task Sets. When we discuss the IS model in the context of mixed-criticality systems, we focus on systems with two to five task classes (i.e., safety criticality levels), as specified for instance by the DO-178C standard [RTC12] with the design assurance levels DAL A to DAL E. We summarize here the established mixed-criticality task set model [BD16], which originated from the work of Vestal [Ves07].

Each task τ_i is characterized by a tuple $(T_i, D_i, \chi_i, \mathbf{C}_i)$, where the period T_i and relative deadline D_i are defined as before. $\chi_i \in \{1, \ldots, K\}$ denotes the task's criticality level, where 1 specifies the lowest and K the highest criticality level in the system. $\mathbf{C}_i$ is now a vector consisting of χ_i execution time bounds, which are monotonically non-decreasing. For instance, for a task τ_i with $\chi_i = 3$: $C_i(1) \leq C_i(2) \leq C_i(3)$, while $C_i(\chi)$ for $\chi > 3$ is not defined. The existence of multiple execution time bounds is based upon Vestal's assumption that "the more confidence one needs in a task execution time bound (the less tolerant one is of missed

deadlines), the larger and more conservative that bound tends to become in practice. [Ves07]". Therefore, one can "assume a task may have a set of alternative worst-case execution times, each assured to a different level of confidence [Ves07]". Eq. (2.2) defines the density $\delta_i(\chi)$ and utilization $u_i(\chi)$ of task τ_i as a function of its execution time bound on level $\chi \in \{1, \ldots, K\}$:

$$\delta_i(\chi) := C_i(\chi)/D_i, \quad u_i(\chi) := C_i(\chi)/T_i \; . \tag{2.2}$$

In the special case of dual-criticality systems, we use S_{LO} to denote the class of tasks with low (LO) criticality and S_{HI} the class of tasks with high (HI) criticality, respectively.

At runtime, a mixed-criticality system has K execution modes $\{M_k | 1 \leq k \leq K\}$ [BD16]. The system execution starts with mode M_1 and remains in this mode as long as all task jobs adhere to their level-1 execution time bounds $C_i(1)$. Whenever at least one task job overruns its level-k execution time bound $C_i(k)$, the system switches dynamically to mode M_{k+1}. After a switch from M_k to M_{k+1} mode, the system can switch back to M_k mode under certain circumstances, e.g., when there are no more ready jobs with criticality level $\chi_i = k + 1$ (awaiting to be executed) in the system [SGTG12, BCLS14]. In the special case of dual-criticality systems, we use LO and HI to denote execution modes M_1 and M_2, respectively.

A mixed-criticality task set is *schedulable* under a scheduling policy if the policy satisfies the following condition $\forall k \in \{1, \ldots, K\}$: In mode M_k, all jobs of tasks τ_i with $\chi_i \geq k$ receive enough execution time according to their execution bound $C_i(k)$ to complete by their deadlines.

2.3.3 Baseline Mixed-Criticality Scheduling Policy

Here, we provide a short overview of the dual-criticality scheduling policy *partitioned EDF-VD* [BCLS14], which is used later in Section 2.5. For a more extensive overview of existing mixed-criticality scheduling policies, we refer to the survey by Burns and Davis [BD16].

Partitioned EDF-VD. Baruah et al. [BBD+12] introduced EDF with virtual deadlines (EDF-VD) for uniprocessor implicit-deadline task sets, later extended to multicores under static task partitioning [BCLS14]. EDF-VD adapts classic preemptive EDF scheduling to ensure the schedulability of HI criticality tasks when the system switches to HI mode. To achieve this goal, EDF-VD assigns *virtual deadlines* to HI criticality jobs, i.e., to jobs of tasks in the S_{HI} class. The virtual deadline is computed by multiplying the original deadline by a fixed factor $x \in (0, 1]$. In LO mode, jobs are scheduled according to EDF, using the original deadlines for LO criticality jobs and the virtual deadlines of HI criticality jobs. Since the virtual deadlines are down-scaled, HI criticality jobs will have some slack to "catch up" upon switching to HI mode. In case the system switches to HI mode, all LO criticality jobs are dropped and HI criticality jobs are scheduled with EDF using their original deadlines.

Baruah et al. [BBD+12] proved that, in LO mode, all deadlines (for HI criticality tasks, virtual deadlines) will be met if:

$$x \geq \frac{U_{\text{HI}}^{\text{LO}}}{1 - U_{\text{LO}}^{\text{LO}}} \,. \tag{2.3}$$

Moreover, Huang et al. [HGST14] proved that there will be no deadline violations for HI criticality tasks during the switch to HI mode or during HI mode operation if all HI criticality tasks finish by their virtual deadlines in LO mode *and*

$$\sum_{\tau_i \in S_{\text{HI}}} \frac{u_i(\text{HI})}{u_i(\text{LO}) + (1 - x)} \leq 1 \,. \tag{2.4}$$

We will use conditions (2.3) and (2.4) in our MC-IS-Server scheduling policy (Section 2.5).

2.4 The IS-Server Policy

As mentioned earlier, theoretical aspects of the IS model have been investigated in [HGA+15]. The authors proposed a fluid scheduling policy, IS-DP-Fair, inspired by DP-Fair [LFS+10], which enforces proportional progress of all tasks across the cores within dedicated system slices (one task class per slice). IS-DP-Fair is an optimal, in terms of schedulability, multi-core scheduling policy for periodic implicit-deadline task sets under the IS constraint. Although fluid scheduling policies [LFS+10, LPG+14, HGA+15] provide a rigorous tool for schedulability analysis, their implementation on actual systems is challenging. This is mainly because of the large number of required task preemptions, migrations, and in our case, synchronous task class switches, which can be costly. To address this concern, we explore a hierarchical, server-based scheduling strategy for *sporadic constrained-deadline* task sets under the IS model; we refer to this strategy as IS-Server. Here, we present IS-Server for multiple (non mixed-criticality) task classes and in Section 2.5 we extend the strategy to mixed-criticality systems.

IS-Server is a partitioned scheduling algorithm, i.e., tasks are not allowed to migrate across cores. K virtual servers are responsible for scheduling the tasks of the respective classes (one server per class). The servers follow a time-triggered schedule, but within each server, tasks are scheduled in a preemptive EDF-based fashion. All cores perform the same server schedule and only one server can be active at a time, thus guaranteeing mutual exclusion among task classes. The IS-Server algorithm has two phases:

1. In the first phase, we partition tasks to cores[2], using either a Mixed

[2]The development of partitioning methods in Section 2.4.1 and Section 2.5.1 consists joint work with Rehan Ahmed.

Variables:

$$\alpha_{i,j} = \begin{cases} 1, & \text{if task } \tau_i \text{ is assigned to core } j \\ 0, & \text{otherwise} \end{cases}$$

Objective:

$$\min \sum_{1 \le k \le K} \left\{ \max_{1 \le j \le m} \left\{ \sum_{\substack{\tau_i \in S_k \\ \alpha_{i,j}=1}} \delta_i \right\} \right\}$$

Constraints:

$$\sum_{1 \le j \le m} \alpha_{i,j} = 1, \ \forall \tau_i \in \{S_k | 1 \le k \le K\},$$

$$\sum_{\alpha_{i,j}=1} \delta_i \le 1, \ \forall 1 \le j \le m$$

Figure 2.2: MILP formulation for the partitioning phase of IS-Server.

Integer Linear Programming formulation (Section 2.4.1.1) or one of the heuristic approaches Worst-Fit Decreasing and Worst-Fit Decreasing-Max (Section 2.4.1.2).

2. After partitioning, we employ a search strategy to find a periodic server schedule that satisfies the timing requirements of all tasks (Section 2.4.2).

Note that existing server methods for multicore platforms [SEL08, BBB09] subdivide the tasks into multiple virtual platforms (servers) that can run in parallel. Such methods violate the IS constraint. Instead, we propose global servers which have exclusive access to the underlying platform. Switching the resource allocation between the global servers happens synchronously on all cores.

2.4.1 Task Partitioning to Cores

In the partitioning phase, we look for task to core assignments. We denote the assignment of task τ_i to core j, $1 \le j \le m$, as $\alpha_{i,j}$. Variable $\alpha_{i,j}$ equals to 1 if τ_i is assigned to core j, or 0 otherwise.

2.4.1.1 Mixed Integer Linear Programming Formulation

Figure 2.2 formulates the partitioning phase of IS-Server as a Mixed Integer Linear Programming (MILP) problem. The objective function that we aim to minimize is the sum of maximum per-class densities across all cores. Having this sum not greater than 1 is a sufficient condition for scheduling constrained-deadline sporadic tasks under the IS constraint

Algorithm 1: Worst-Fit Decreasing (WF)

Input: S: Task set
Output: M_j: m sets of tasks assigned to individual cores.

1: $M_j \leftarrow \emptyset$, $\forall\ 1 \leq j \leq m$: Sets of tasks assigned to individual cores
2: $\text{density}_j \leftarrow 0$, $\forall\ 1 \leq j \leq m$: Densities of cores
3: **for each** $S_k \in S$ **do**
4: Sort tasks $\tau_i \in S_k$ in descending order of density δ_i
5: **while** not all tasks in S_k are assigned **do**
6: $\tau_i \leftarrow$ First unassigned task in class S_k
7: $c \leftarrow \underset{1 \leq j \leq m}{\text{argmin}}(\text{density}_j)$
8: $M_c \leftarrow M_c \cup \{\tau_i\}$
9: $\text{density}_c \leftarrow \text{density}_c + \delta_i$
10: **end while**
11: **end for**

(see Theorem 2.3, Section 2.4.2.3). Note that the objective function is not linear due to the max operator. It can be linearized by adding a variable z_k for each $1 \leq k \leq K$ and m linear constraints for each z_k in the form:

$$z_k \geq \sum_{\substack{\tau_i \in S_k \\ \alpha_{i,j}=1}} \delta_i, \qquad \forall 1 \leq j \leq m\ .$$

The objective then becomes $\sum_{1 \leq k \leq K} z_k$. Since this objective is being *minimized*, each individual z_k will be set to its lowest possible value and so will $\max_{1 \leq j \leq m} \Big\{ \sum_{\substack{\tau_i \in S_k \\ \alpha_{i,j}=1}} \delta_i \Big\}$.

The constraints of the MILP formulation guarantee that each task is assigned to exactly one core and the total density on each core is bounded by 1.

2.4.1.2 Worst-Fit Decreasing and Worst-Fit Decreasing-Max

Due to the complexity of solving the MILP problem, we also evaluate two different heuristic approaches for assigning tasks to cores. The proposed methods are adaptations of the Worst-Fit Decreasing (WF) bin packing heuristic. The rationale behind choosing WF is that this heuristic generally assigns items to individual bins with the objective of keeping the level of all bins equal/balanced. In the context of task to core assignments, this would mean that the densities of all cores for a given class are balanced, which in turn can lead to lower values for the objective function of our partitioning problem (see optimal formulation above). Algorithm 1

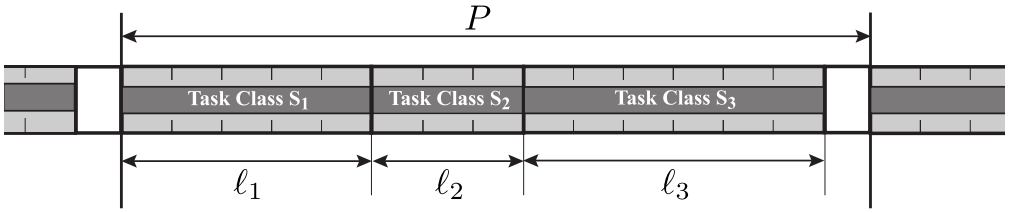


Figure 2.3: TDMA server schedule for three task classes S_1, S_2, S_3. The server schedule is repeated with a period P. Within each server slot (with respective length ℓ_1, ℓ_2, ℓ_3), tasks are scheduled based on partitioned EDF.

outlines the WF algorithm. Essentially, we sort tasks in each task class in descending order w.r.t. their density. Then, we assign tasks from each task class iteratively to the core which has the lowest density.

WF-Max is an adaptation of Algorithm 1. In WF-Max, once all tasks of a given task class are assigned, the densities of all cores are set to the maximum density across all cores. Namely, the following line is added after Line 10 in Algorithm 1:

$$\text{density}_j = \max_{1 \le k \le m} \left\{ \text{density}_k \right\}, \quad \forall\, 1 \le j \le m \,.$$

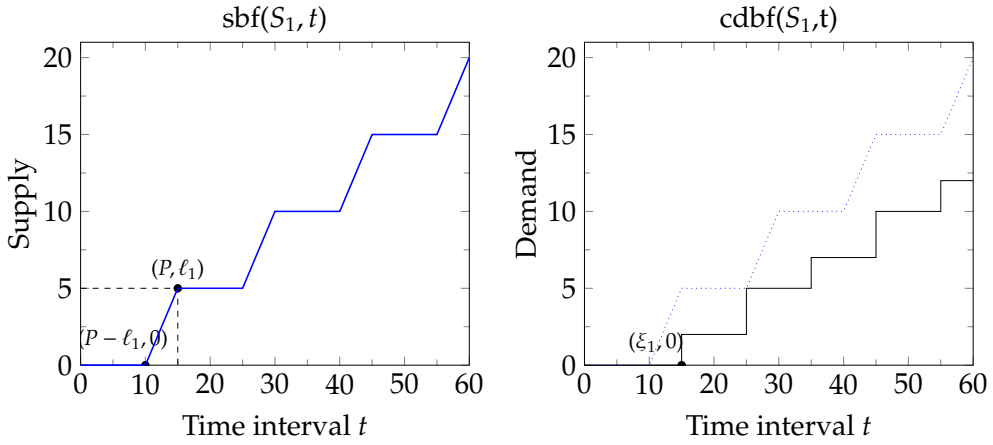
This adaptation models the schedulability loss due to the IS constraint. Therefore, WF-Max is expected to provide better schedulability compared to WF.

2.4.2 The IS-Server Algorithm

Once the partitioning phase is done, we apply hierarchical scheduling using K global virtual servers, one for each task class S_k ($1 \le k \le K$). To enforce the IS constraint, only one global server can be active at a time. To achieve this, we consider the whole multicore as a time-division-multiple-access (TDMA) resource [HHK01, WT06a] and assign disjoint time slots to the servers. During each time slot, tasks of the respective class are scheduled across the cores based on partitioned EDF.

A TDMA-based server schedule is illustrated in Figure 2.3 for three task classes. IS-Server periodically assigns the TDMA resource, i.e., the multicore platform, to the task class servers according to a predefined pattern. This pattern recurs with a period P, to which we refer as the TDMA cycle length. In every TDMA cycle, one single slot is assigned to the server of task class S_k, with length ℓ_k which depends on its execution demand. We call a TDMA server schedule *feasible* if the sum of the slot lengths for all servers is less than or equal to the TDMA cycle length:

$$P \ge \sum_{k=1}^{K} \ell_k \,. \tag{2.5}$$



(a) Supply bound function for class S_1.

(b) Class demand bound function for class S_1. The dotted curve represents sbf(S_1, t).

Figure 2.4: Supply and demand bound functions for task class S_1 in TDMA schedule.

2.4.2.1 Schedulability Analysis of IS-Server

To analyze the schedulability of a task set S under IS-Server, we use the well-known concept of demand and supply bound functions [BCGM99, MFC01, WT05, SL03, SL04]. We now show how to compute these functions for IS-Server and formulate the schedulability conditions.

Execution Supply of TDMA Resource. A supply bound function sbf(t) lower-bounds the amount of supplied execution time by a platform in any time interval of length t [MFC01, WT05]. For instance, a unit-speed single-core processor has sbf(t) = t. To determine the lower supply bounds for each task class under IS-Server, we must consider that in the TDMA server schedule: 1. The server for task class S_k may not be able to execute for a time interval that is upper-bounded by $P - \ell_k$; 2. After this interval, the server is granted exclusive access to the platform for a time interval of length ℓ_k. Therefore, there is no guarantee for supplied execution time to the k-class server during any time interval $0 \leq t < P - \ell_k$, but there is a guarantee for a supplied execution time of $(t - (P - \ell_k))$ in any time interval $P - \ell_k \leq t < P$. This leads to the definition of the supply bound function of the k-class server, sbf(S_k, t), as follows [WT06a]:

$$\text{sbf}(S_k, t) = \max\left\{\left\lfloor \frac{t}{P} \right\rfloor \ell_k, t - \left\lceil \frac{t}{P} \right\rceil (P - \ell_k)\right\} . \tag{2.6}$$

Figure 2.4(a) illustrates the supply bound function for server S_1 in the TDMA server schedule of Figure 2.3.

Execution Demand of Task Classes. We estimate the execution demand of each task class S_k using the concept of demand bound functions, which was described by Baruah et al. [BCGM99] for single tasks. First, we

introduce some definitions.

Task demand bound function (dbf)[BCGM99]: Function $\text{dbf}(\tau_i, t)$ denotes the maximum execution demand of task τ_i in any time interval of length t. Execution demand is calculated as the total execution time of jobs of τ_i that have arrival times and deadlines within the time interval. For sporadic tasks, it is given by:

$$\text{dbf}(\tau_i, t) = \max\left\{\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i \ .$$

Per-Core Class Demand Bound Function (pc_cdbf): Function $\text{pc\_cdbf}(S_k, t, j)$ denotes the maximum aggregate execution demand of all tasks τ_i in task class S_k which run on core j, in any time interval of length t:

$$\text{pc\_cdbf}(S_k, t, j) = \sum_{\substack{\tau_i \in S_k \\ \alpha_{i,j}=1}} \text{dbf}(\tau_i, t) \ .$$

Class Demand Bound Function (cdbf): Function $\text{cdbf}(S_k, t)$ denotes the maximum aggregate execution demand of task class S_k across all cores, in any time interval of length t:

$$\text{cdbf}(S_k, t) = \max_{1 \le j \le m} \{\text{pc\_cdbf}(S_k, t, j)\} \ .$$

A sample class demand bound function is illustrated in Figure 2.4(b).

Schedulability Conditions. It is known [SL03, SL04] that to ensure schedulability of a given workload, the supply bound function must be at least equal to the demand bound function for any time interval. This leads to the following *necessary* and *sufficient* schedulability condition.

Theorem 2.1. *A task set S is schedulable under IS-Server iff*

$$\forall t \ge 0, \forall k \in \{1, \ldots, K\}: \quad \text{cdbf}(S_k, t) \le \text{sbf}(S_k, t) \ . \tag{2.7}$$

Proof. The theorem follows directly from the schedulability test (1) of [SL04]. □

Additionally, we use a *necessary* schedulability condition for quick identification of unschedulable task sets. This condition is based on the principle that the supply bound function of any resource cannot surpass $\text{sbf}_{\max}(t) = t$.

Theorem 2.2. *For any task set S schedulable under IS-Server, the following condition holds:*

$$\forall t \ge 0: \quad \sum_{k=1}^{K} \text{cdbf}(S_k, t) \le t \ . \tag{2.8}$$

Proof. The theorem follows directly from Theorem 1 of [BCGM99]. □

2.4.2.2 TDMA Parameter Selection

In this section, we show how to select the TDMA server schedule parameters, i.e., the cycle length P and the slot length ℓ_k for each server S_k, such that (i) the resulting supply bound function for each task class satisfies the schedulability condition (2.7), and (ii) the frequency of synchronous task class switches is minimized. For the derivation of the TDMA server schedule, we adapt the approach described by Wandeler et al. [WT06a], which applies a modular performance analysis of TDMA resources based on real-time calculus [TCN00]. This approach was originally developed for distributed systems with nodes communicating via message passing over a shared TDMA-arbitrated bus. Here, we adapt it to be applicable to real-time server scheduling.

First, we derive the minimum feasible slot length for each class S_k for a given cycle length P, and then provide an upper bound on the cycle length P itself, such that schedulability conditions can be satisfied. Next, we present a search method for the selection of parameters P, S_k, $\forall k$, with the objective of minimizing the frequency of task class switches.

Minimum Slot Length for Given P. Suppose that the TDMA cycle length P is fixed. By applying the results (Eq. 12) of [WT06a], we get the minimum slot length ℓ_k that can be assigned to the server for task class S_k, such that schedulability condition (2.7) is satisfied:

$$\ell_k = \sup_{t \geq 0} \left\{ \min \left\{ \frac{\text{cdbf}(S_k, t)}{\left\lfloor \frac{t}{P} \right\rfloor}, \frac{\text{cdbf}(S_k, t) - t + \left\lceil \frac{t}{P} \right\rceil P}{\left\lceil \frac{t}{P} \right\rceil} \right\} \right\} . \tag{2.9}$$

Based on the analysis of [WT06a], ℓ_k is the smallest time slot allocation that guarantees $\forall t \geq 0 : \text{cdbf}(S_k, t) \leq \text{sbf}(S_k, t)$ on a TDMA resource with cycle length P. The computation of ℓ_k can be performed using the operations of the real-time calculus toolbox [WT06b], once $\text{cdbf}(S_k, t)$ has been specified in the form of an arrival curve [TCN00]. Note that ℓ_k is monotonically non-decreasing with P.

Upper Bound on Cycle Length. Since the motivation for adopting IS-Server is to reduce the frequency of synchronous task class switches compared to previous IS fluid policies [HGA+15], we aim at maximizing the TDMA cycle length P and consequently, the slot lengths ℓ_k which increase monotonically with P. To provide an upper bound on P, we first introduce parameter ξ_k, which is defined as the x-coordinate of the first non-zero point of $\text{cdbf}(S_k, t)$ (for an example, see Figure 2.4(b)):

$$\xi_k = \inf_{t \geq 0} \{\text{cdbf}(S_k, t) > 0\} . \tag{2.10}$$

ξ_k determines the maximum time interval that the k-class server can "wait" until it receives execution. In other words, ξ_k determines the maximum

sum of slot lengths that can be allocated to the remaining servers (for classes S_j, $j \neq k$) during a TDMA cycle:

$$\sum_{\substack{j=1 \\ j \neq k}}^{K} \ell_j \leq \xi_k \; . \tag{2.11}$$

By summing up Eq. (2.11) $\forall k \in \{1, \ldots, K\}$, we have:

$$(K-1) \cdot \sum_{k=1}^{K} \ell_k \leq \sum_{k=1}^{K} \xi_k \; . \tag{2.12}$$

Hence, for $K > 1$, the sum of slot lengths for all servers is upper bounded by:

$$\max \left\{ \sum_{k=1}^{K} \ell_k \right\} = \frac{1}{K-1} \sum_{k=1}^{K} \xi_k \; . \tag{2.13}$$

In principle, the maximum cycle length P_{max} can be greater than the maximum sum of slot lengths for all servers S_k because slack time can be reserved in every TDMA cycle, e.g., for future allocation of time slots or for accounting of runtime overheads (see TDMA schedule specification in Figure 2.3). Since in this work slack time reservation is not of interest, we take the right-hand side of Eq. (2.13) as the upper limit P_{max} of our search for the TDMA cycle length.

Search Method. We now employ a search method to determine the cycle length P and the server slot lengths $\ell_k, \forall k$ of the TDMA server schedule. The objective is finding the maximum cycle length $P \in (0, P_{max}]$ and the corresponding minimum server slot assignments ℓ_k which lead to a *feasible* IS-Server schedule according to condition (2.5). Note that for any cycle length P in the search space, the computation of the minimum server slots according to Eq. (2.9) ensures schedulability of the task set (condition (2.7)), but not necessarily feasibility of the IS-Server schedule (condition (2.5)).

Algorithm 2 implements the search method for the TDMA server schedule parameters. Initially (lines 1–4), it checks whether the task set S features more than one task class. In the special case of a single task class, the respective server receives full service, hence the cycle length and the slot length ℓ_1 are both set to infinity (no class switch). Subsequently (lines 5–7), it checks the necessary schedulability condition (2.8) for early detection of unschedulable task sets. Because this test can be computationally expensive, in practice we perform it by computing the long-term rate of functions $\text{cdbf}(S_k, t)$, $\forall k$, and comparing their sum against value 1 (the rate of function $\text{sbf}_{max}(t) = t$).

If this test is successful, the algorithm proceeds (lines 8–20) with linear search for P in $(0, P_{max}]$ in reverse order, i.e., from P_{max} to 0. To reduce

Algorithm 2: Search method for parameters of TDMA server schedule

Input: cdbf(S_k, t) for all classes, upper bound P_{max}, search quantum q
Output: Cycle length P, slot lengths ℓ_k for all classes, or INFEASIBLE

1: **if** $K = 1$ **then**
2: $P \leftarrow$ Inf; $\ell_1 \leftarrow$ Inf ▷ A single task class receives full service
3: **return** P, ℓ_1
4: **end if**
5: **if** $\exists t : \sum_{k=1}^{K} \text{cdbf}(S_k, t) > t$ **then**
6: **return** INFEASIBLE ▷ Violation of condition (2.8)
7: **end if**
8: **for** $p \leftarrow q \cdot \left\lceil \frac{P_{max}}{q} \right\rceil$ to q **do** ▷ Search function
9: **for each** $k \in \{1, \ldots, K\}$ **do**
10: $slot\_len(p, k) \leftarrow$ minimum slot length computed by Eq. (2.9)
11: **end for**
12: $slack(p) \leftarrow p - \sum_{k=1}^{K} slot\_len(p, k)$
13: **if** $slack(p) \geq 0$ **then**
14: $P \leftarrow p$ ▷ Max. cycle length leading to feasible solution
15: **for each** $k \in \{1, \ldots, K\}$ **do**
16: $\ell_k \leftarrow slot\_len(p, k)$
17: **end for**
18: **return** $P, \ell_k, \forall k \in \{1, \ldots, K\}$
19: **end if**
20: **end for**
21: **return** INFEASIBLE ▷ No feasible solution found

computational complexity, we quantize the search space with granularity q. Namely, the algorithm searches in the period set $\left\{q, 2q, \ldots, q \cdot \left\lceil \frac{P_{max}}{q} \right\rceil\right\}$. For each considered period, it computes: (i) the minimum slot lengths for all servers such that schedulability condition (2.7) holds, and (ii) the difference between the sum of the slot lengths and the period. If this difference is non-negative, the current TDMA parameters define a feasible server schedule, which satisfies the real-time requirements of all tasks. Note that function $slot\_len(p, k)$ computes the minimum slot length for the k-class server and for given period p, while $slack(p)$ computes the difference between the sum of the slot lengths and the current period p.

The search stops when a period p is found for which $slack(p) \geq 0$ or when the whole search space has been explored. In the first case, p represents the maximum cycle length in the search space that leads to a feasible TDMA server schedule. If such a value is found, the algorithm returns the cycle length and the respective minimum slot lengths for all task class servers (lines 13–19). If no feasible solution has been found after an exhaustive search of the period set, the algorithm returns `INFEASIBLE` (line 21). Note that due to the quantization of the search space, it is

possible that existing feasible solutions with $P \in (zq, (z+1)q), z \in \mathbb{Z}^*$ are not found by Algorithm 2. This can be solved by decreasing the search quantum q at the expense of increased computational complexity.

2.4.2.3 Schedulability Loss of IS-Server

Here, we quantify the loss of schedulability due to enforcing the IS constraint under partitioned scheduling. Note that we are only interested in the schedulability loss due to Isolation Scheduling and *not* due to a specific task partitioning method. Theorem 2.3 provides a bound on the required speedup to enforce isolation in this case[3].

Theorem 2.3. *Any sporadic, constrained-deadline task set S schedulable by a partitioned scheduling policy without the IS constraint on a m-core unit-speed processor is schedulable under the IS constraint on a m-core processor that is $\min\{K, m\}$ times faster. This speedup bound is tight in the special case of tasks with implicit deadlines.*

Proof. Assume that, after partitioning of a task set to m cores, the total density of class k on core j is δ_k^j. Since the task set is schedulable under partitioned scheduling and EDF is an optimal uniprocessor scheduling technique, we have $\forall j : \sum_{k=1}^{K} \delta_k^j \leq 1$. Given such a partitioning, a sufficient condition so that the system is schedulable under the IS constraint is that:

$$\sum_{k=1}^{K} \max_{1 \leq j \leq m} \delta_k^j \leq 1 \ . \tag{2.14}$$

To see why this condition is sufficient[4], assume an IS-compliant policy that is similar to the DP-Fair scheduling policy [LFS+10]. Namely, we have arbitrary small quantum σ in the system (for IS-Server, equal to the TDMA cycle length), and within σ, a slot size of $\sigma \cdot \max_{j=1}^{m} \delta_k^j$ is allocated to task class k. As long as any task class k receives its "fair" portion within the quantum, i.e., $\sum_{k=1}^{K} \sigma \cdot \max_{j=1}^{m} \delta_k^j \leq \sigma$, we know from fluid scheduling [LFS+10, HGA+15] that all tasks in any task class will meet their deadlines.

Therefore, the problem of finding a speedup bound for Isolation Scheduling can be formulated as:

$$\max \left\{ \sum_{k=1}^{K} \max_{1 \leq j \leq m} \delta_k^j \right\} \quad \text{s.t. } \forall 1 \leq j \leq m : \quad \sum_{k=1}^{K} \delta_k^j \leq 1 \ . \tag{2.15}$$

[3]The proof of Theorem 2.3 consists joint work with Pengcheng Huang.

[4]Note that condition (2.14) is also necessary if tasks have implicit deadlines.

As a result, we have:

$$\sum_{k=1}^{K} \max_{1 \le j \le m} \delta_k^j \le \sum_{k=1}^{K} 1 \le K \,. \tag{2.16}$$

Furthermore, we have:

$$\sum_{k=1}^{K} \max_{1 \le j \le m} \delta_k^j \le \sum_{k=1}^{K} \sum_{j=1}^{m} \delta_k^j \le \sum_{j=1}^{m} \sum_{k=1}^{K} \delta_k^j \le \sum_{j=1}^{m} 1 \le m \,. \tag{2.17}$$

Thus, the upper bound of $\max \sum_{k=1}^{K} \max_{1 \le j \le m} \delta_k^j$ is $\min\{K, m\}$. The theorem follows directly.

The above speedup bound is tight w.r.t. Eq. (2.14), since it can be indeed achieved with certain choices of task class densities (see the following two cases). In addition, if tasks in S have implicit deadlines, our derived speedup bound is also exact since condition (2.14) is an exact schedulablity test in that case.

Case 1: $K \ge m$**.** We let $\delta_k^j = 1, \forall k = j \wedge j \le m-1$, $\delta_k^m = \frac{1}{k-m+1}, \forall k \ge m$, and $\delta_k^j = 0$ for all other cases. Then, $\sum_{k=1}^{K} \max_{1 \le j \le m} \delta_k^j = m$.

Case 2: $K < m$**.** We let $\delta_k^j = 1, \forall k = j \wedge k \le K$, and $\delta_k^j = 0$ for all other cases. Then, $\sum_{k=1}^{K} \max_{1 \le j \le m} \delta_k^j = K$.

□

2.5 The MC-IS-Server Policy

In this section, we extend the IS-Server strategy to *mixed-criticality* systems; we refer to the extended scheduling policy as MC-IS-Server. The extension builds upon the theory of the EDF-VD scheduling policy [BCLS14] which was presented in Section 2.3.3. We first focus on dual-criticality task sets. We show how to adapt the task partitioning algorithms of the previous section in order to capture the special schedulability requirements of mixed-criticality systems and propose a new heuristic approach, MC-EY-WF (Section 2.5.1). We then show how MC-IS-Server algorithm is applied to dual-criticality systems (Section 2.5.2) and how task partitioning and scheduling can be extended to task sets with $K > 2$ criticality levels (Section 2.5.3).

2.5.1 Task Partitioning to Cores (Dual-Criticality)

The partitioning phase of MC-IS-Server assigns tasks to cores in order to guarantee schedulability in both LO and HI execution mode (see Section 2.3.2). Recall that in LO (or M_1) execution mode, all tasks τ_i are executed according to their LO-level execution time bound $C_i(\text{LO})$ and need to complete by their deadlines. In HI (or M_2) execution mode, only high-criticality tasks $\tau_i \in S_{\text{HI}}$ need to complete by their deadlines, when executed according to their HI-level execution time bound $C_i(\text{HI})$. To guarantee schedulability, we need to ensure that schedulability conditions (2.3) and (2.4) of Section 2.3.3 hold for *each* core. In the following, we first provide an overview about existing methods for partitioning dual-criticality task sets to cores, and then we propose partitioning methods that specifically aim at minimizing the schedulability loss due to the IS constraint on dual-criticality multicores.

2.5.1.1 Existing approaches

We review existing approaches for partitioned mixed-criticality scheduling on multicore platforms and reason why they may (not) be applicable to Isolation Scheduling. Kelly et al. [KAZ11] proposed and compared several bin packing heuristics for assigning mixed-criticality tasks to multicores. However, their analysis is restricted application and comparison of fixed-priority scheduling algorithms, making the approaches of [KAZ11] not applicable to our work. Burns et al. [BFB15] proposed partitioned scheduling of mixed-criticality tasks using cyclic executives. In cyclic-executive scheduling, the processor executes a set of frames. Each frame contains one or more tasks for which a static schedule has been predetermined. This execution model is restrictive in the sense that task periods/deadlines have to be multiples of frame duration. Therefore, mostly equal-period or harmonic-period tasks are supported. Nonetheless, the approaches presented in [BFB15] are relevant to our work because the cyclic-executive strategy also enforces the IS constraint. The authors evaluated three different partitioning schemes: First-Fit (FF), Worst-Fit (WF), First-Fit with Branch and Bound (FFBB). The results in [BFB15] show that FF is not a good heuristic for enforcing Isolation Scheduling. This is because FF fills up one core before moving on to the next core. This leads to utilization imbalance among the cores, thus resulting in a high schedulability loss due to the IS constraint. The performance of WF and FFBB is comparable, with FFBB being marginally better for high utilizations. We compare our results with the FFBB scheme in Section 2.6. Gu et al. [GGDY14] proposed a mixed-criticality partitioning approach called MPVD. In this approach, first all HI criticality tasks are assigned to cores using WF. Afterwards, virtual deadlines are assigned to HI criticality tasks based on a per-task deadline shortening approach, originally proposed by Ekberg and Yi [EY13]. After this phase,

Variables: x_j is the deadline scaling factor for HI tasks on core j

$$\alpha_{i,j} = \begin{cases} 1, & \text{if task } \tau_i \text{ is assigned to core } j \\ 0, & \text{otherwise} \end{cases}$$

Objective:

$$\min \left\{ \max_{1 \le j \le m} \left\{ \sum_{\substack{\tau_i \in S_{\text{HI}} \\ \alpha_{i,j}=1}} \delta_i(\text{LO})/x_j \right\} + \max_{1 \le j \le m} \left\{ \sum_{\substack{\tau_i \in S_{\text{LO}} \\ \alpha_{i,j}=1}} \delta_i(\text{LO}) \right\} \right\}$$

Constraints:

$$\sum_{1 \le j \le m} \alpha_{i,j} = 1, \quad \forall \tau_i \in \{S_{\text{HI}}, S_{\text{LO}}\}$$

$$\sum_{\substack{\tau_i \in S_{\text{HI}} \\ \alpha_{i,j}=1}} \left(\frac{\delta_i(\text{HI})}{\delta_i(\text{LO}) + 1 - x_j} \right) \le 1, \quad \forall 1 \le j \le m$$

Figure 2.5: MIQCP formulation for the partitioning phase of MC-IS-Server.

LO criticality tasks are assigned using FF. Baruah et al. [BCLS14] proposed partitioned EDF-VD. In partitioned EDF-VD, first HI criticality tasks are assigned to cores using FF. Any given core is allowed to have a maximum HI mode utilization of 3/4. After assigning the HI criticality tasks, all LO criticality tasks are assigned using FF. For both works [GGDY14] and [BCLS14], the selection of FF makes the partitioning approaches ill-suited for Isolation Scheduling. In this work, we focus on task partitioning approaches based on Worst-Fit, since this heuristic provides a good utilization balance and therefore, it is expected to incur low schedulability loss due to the IS constraint.

2.5.1.2 Mixed Integer Quadratically Constrained Programming Formulation

Figure 2.5 formulates the partitioning phase of MC-IS-Server as a Mixed Integer Quadratically Constrained Programming (MIQCP) problem. Similarly to other mixed-criticality scheduling policies (see Section 2.3.3), in LO execution mode we scale the deadlines of HI criticality tasks assigned to core j by a factor x_j. In our formulation, x_j is a variable, which makes the optimization problem quadratically constrained. The objective function that we want to minimize is the sum of maximum densities across all cores, for LO and HI criticality tasks, in LO execution mode. The selected objective function leads to balanced loads for both task classes in LO mode. Balancing load is crucial because it minimizes the schedulability loss due to the IS constraint. Note that in HI mode there

is no such schedulability loss, since all LO criticality tasks are suspended. The constraints of the formulation guarantee that each task is assigned to exactly one core and that HI mode schedulability, as defined by Eq. (2.4)[5], is satisfied.

While solving the MIQCP problem is computationally expensive, this phase only needs to run offline. In practice, the time the optimizer takes to run is reasonable on modern hardware. For instance, 99% of the evaluated task sets in Section 2.6 were partitioned in less than 100 seconds on a quad-core Core-i7 Haswell platform. For cases with a very high number of cores, where the complexity of solving the MIQCP problem becomes prohibitive, heuristic approaches are proposed in the following subsections.

2.5.1.3 MC Worst-Fit Decreasing and Worst-Fit Decreasing-Max

We evaluate two heuristic algorithms for partitioning dual-criticality tasksets to cores, which build upon the algorithms of Section 2.4.1.2. Details of the Mixed-Critical Worst-Fit Decreasing (MC-WF) are given in Algorithm 3. We first sort the tasks of each criticality level in descending order w.r.t. their LO mode density. Following this, the HI criticality tasks are assigned to cores (lines 7–17). Specifically, we try assigning a given HI criticality task to each of the m cores. For a given core j, we compute the maximum value of the deadline shortening factor x_j^* (resulting in minimum increase in LO mode density), such that transition from LO mode to HI mode is feasible for HI criticality tasks (line 10). We then compute the LO mode density of all cores (vector l_j, line 11), while considering the newly computed value of x_j^*. After trying all cores, we choose the assignment which yields the lowest mean-squared error (lines 13–14). The reason for choosing the assignment with the minimum mean-squared error is that it results in minimal deviation of the LO mode densities across all cores. In turn, this leads to the best load balancing across the cores, hence minimizing the schedulability loss due to the IS constraint. After all HI criticality tasks have been assigned, each LO criticality task is assigned to the core which has the lowest LO mode density.

Like in the non mixed-critical case, we also evaluate a *Max* version of MC-WF algorithm (MC-WF-Max). In MC-WF-Max, after all HI criticality tasks are assigned, LO mode densities of all cores are set to the maximum LO mode density across the multi-core platform.

[5]Note that we use an adapted version of Eq. (2.4) where task utilizations are replaced by task densities. This adaptation is done to accommodate mixed-criticality task sets with *constrained* deadlines.

Algorithm 3: Mixed-Critical Worst-Fit Decreasing (MC-WF)

Input: S_{HI}: HI criticality task set, S_{LO}: LO criticality task set

Output: M_j: m sets of tasks assigned to individual cores, x_j: m deadline shortening factors for cores

1: **for each** $S \in \{S_{\mathrm{HI}}, S_{\mathrm{LO}}\}$ **do**
2: Sort tasks in descending order w.r.t. LO mode density $\delta_i(\mathrm{LO})$
3: **end for**
4: $M_j \leftarrow \emptyset,\ \forall\ 1 \leq j \leq m$: Set of tasks assigned to core j.
5: $\mathrm{density}_j \leftarrow 0,\ \forall 1 \leq j \leq m$: LO mode density of core j.
6: $x_j \leftarrow 1,\ \forall 1 \leq j \leq m$: Common deadline shortening factor for core j.
7: **while** not all tasks in S_{HI} are assigned **do**
8: $\tau_z \leftarrow$ First unassigned task in class S_{HI}
9: **for** $j \leftarrow 1$ to m **do**
10: $$x_j^* \leftarrow \max\left\{x \middle| \sum_{\tau_i \in (M_j \cup \{\tau_z\})} \frac{\delta_i(\mathrm{HI})}{\delta_i(\mathrm{LO}) + (1-x)} \leq 1 \wedge 0 < x \leq 1\right\}$$
11: $$l_{j,k} \leftarrow \begin{cases} \left(\sum_{\tau_i \in (M_j \cup \{\tau_z\})} \delta_i(\mathrm{LO})\right) \Big/ x_j^*, & \text{if k=j} \\ \mathrm{density}_k, & \text{otherwise} \end{cases}$$
12: **end for**
13: $$c \leftarrow \underset{1 \leq j \leq m}{\mathrm{argmin}} \left\{\frac{1}{m} \sum_{k=1}^{m} (\overline{l_j} - l_{j,k})^2\right\}, \text{ where } \overline{l_j} = \sum_{k=1}^{m} l_{j,k}/m$$
14: $M_c \leftarrow M_c \cup \{\tau_z\}$
15: $x_c \leftarrow x_c^*$
16: $\mathrm{density}_c \leftarrow l_{c,c}$
17: **end while**
18: **while** not all tasks in S_{LO} are assigned **do**
19: $\tau_z \leftarrow$ First unassigned task in class S_{LO}
20: $$c \leftarrow \underset{1 \leq j \leq m}{\mathrm{argmin}} \{\mathrm{density}_j\}$$
21: $M_c \leftarrow M_c \cup \{\tau_z\}$
22: $\mathrm{density}_c \leftarrow \mathrm{density}_c + \delta_z(\mathrm{LO})$
23: **end while**

2.5.1.4 Per-Task Deadline Shortening Factor

In the partitioning approaches of the previous section, we considered a common deadline shortening for all HI criticality tasks executing on a given core. This is a restrictive assumption and schedulability may be improved by considering a separate deadline shortening factor for each *task*, instead of each core. In this section, we first present a scheme for assigning per-task deadline shortening factors in dual-criticality systems, proposed by Ekberg and Yi [EY13]. We refer to this deadline assignment scheme as EY. We then propose the MC-EY-WF algorithm for partitioning dual-criticality task sets for the MC-IS-Server policy.

EY Heuristic. EY [EY13] is a heuristic approach for assigning per-task virtual deadlines to HI criticality tasks in sporadic, constrained-deadline dual-criticality systems. EY is designed to improve dual-criticality schedulability on single cores. Its basic principle lies in the construction of separate task demand bound functions for tasks in LO mode and HI mode. We use $D_i(\text{LO})$ and $D_i(\text{HI})$ to denote the LO and HI mode relative deadlines of task τ_i. For τ_i, we use $\text{dbf}_{\text{LO}}(\tau_i, t)$ and $\text{dbf}_{\text{HI}}(\tau_i, t)$ to denote the LO and HI mode demand bound functions, respectively. These functions are defined as follows:

$$\text{dbf}_{\text{LO}}(\tau_i, t) = \max\left\{\left\lfloor \frac{t - D_i(\text{LO})}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i(\text{LO}), \tag{2.18}$$

where $D_i(\text{LO}) = D_i$ if $\tau_i \in S_{\text{LO}}$ and $D_i(\text{LO}) \leq D_i(\text{HI}) = D_i$ if $\tau_i \in S_{\text{HI}}$.
For $\tau_i \in S_{\text{HI}}$:

$$\text{dbf}_{\text{HI}}(\tau_i, t) = \text{full}(\tau_i, t) - \text{done}(\tau_i, t), \tag{2.19}$$

where

$$\text{full}(\tau_i, t) = \max\left\{\left\lfloor \frac{t - (D_i(\text{HI}) - D_i(\text{LO}))}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i(\text{HI}),$$

$$\text{done}(\tau_i, t) = \begin{cases} \max\{C_i(\text{LO}) - n & \text{if } D_i(\text{HI}) > n \geq D_i(\text{HI}) - D_i(\text{LO}) \\ \quad + D_i(\text{HI}) - D_i(\text{LO}), 0\}, & \\ 0, & \text{otherwise} \end{cases}$$

and $n = t \mod T_i$. For $\tau_i \in S_{\text{LO}}$: $\text{dbf}_{\text{HI}}(\tau_i, t) = 0, \forall t$. For all tasks executing on a unit-speed core, we have the following schedulability conditions:

$$\forall t \geq 0 : \quad \sum_{\tau_i \in S} \text{dbf}_{\text{LO}}(\tau_i, t) \leq t \qquad \text{and} \qquad \sum_{\tau_i \in S} \text{dbf}_{\text{HI}}(\tau_i, t) \leq t \,. \tag{2.20}$$

For a given HI critical task $\tau_i \in S_{\text{HI}}$, we can *tune* $D_i(\text{LO})$ to improve schedulability. Shortening $D_i(\text{LO})$ shifts dbf_{HI} to the right. Therefore, schedulability in HI mode improves. However, shortening $D_i(\text{LO})$ simultaneously shifts dbf_{LO} to the left, making the LO mode difficult

Algorithm 4: Mixed Critical EY Worst-Fit Decreasing (MC-EY-WF)

Input: S_{HI}: HI criticality taskset. S_{LO}: LO criticality taskset

Output: M_j: m sets of tasks assigned to individual cores or FAILURE. Deadlines of HI criticality tasks are shortened

1: S_{HI}, S_{LO}, M_j, density$_j$ as defined in Algorithm 3
2: **while** not all tasks in S_{HI} are assigned **do**
3: $\tau_z \leftarrow$ First unassigned task in class S_{HI}
4: candidates $\leftarrow$ cores sorted in descending order w.r.t LO density
5: **for each** k in candidates **do**
6: Apply EY algorithm [EY13] to $M_k \cup \{\tau_z\}$
7: **if** M_k is schedulable **then**
8: $M_k \leftarrow M_k \cup \{\tau_z\}$ ▷ M_k has tasks with shortened deadlines
9: update density$_k$ based on new deadlines
10: **break**
11: **end if**
12: **end for**
13: **if** τ_z is not assigned to any core **then**
14: **return** FAILURE
15: **end if**
16: **end while**
17: density$_j \leftarrow \max_{1 \le k \le m}\{\text{density}_k\}, \ \forall\ 1 \le j \le m$
18: Assign tasks in S_{LO} according to line 18-23 of Algorithm 3

to schedule. The EY algorithm iteratively reduces by 1 the deadline of the HI criticality task τ_i, for which $\text{dbf}_{\text{HI}}(\tau_i, l)$ - $\text{dbf}_{\text{HI}}(\tau_i, l-1)$ is the highest, where $l = \min_t\{t | \text{dbf}_{\text{HI}}(\tau_i, t) > t\}$. This process of shortening deadlines is repeated until the HI mode becomes feasible or the LO mode becomes infeasible. We encourage interested readers to review [EY13] for details on the EY algorithm.

Mixed-Critical EY Worst Fit (MC-EY-WF). Here, we propose a joint task partitioning and deadline shortening heuristic. Task partitioning is based on the Worst-Fit Decreasing heuristic. Deadline shortening for HI criticality tasks is performed by the EY algorithm. The details of this joint approach are given in Algorithm 4. Similar to the case for MC-WF and MC-WF-Max, first tasks are sorted based on their LO mode density. HI criticality tasks are assigned first. We always try assigning a HI criticality task to a core with the lowest LO mode density. For each assignment attempt, we evaluate the shortened LO mode deadlines of all HI criticality tasks, such that both LO mode and HI mode are schedulable. To check schedulability on a given core, conditions (2.20) need to be satisfied for all tasks assigned to that core. If EY cannot find a feasible deadline assignment, we try assigning the task to the core with the next

lowest LO mode density. This process is repeated until we have either found a feasible assignment or tried all cores without finding a feasible assignment. In the later case, the task set is deemed unschedulable. After all HI criticality tasks are assigned, we set the LO mode densities of all cores to the maximum LO mode density (similar to MC-WF-Max). Finally, all LO criticality tasks are assigned in a WF fashion.

2.5.2 The MC-IS-Server Algorithm (Dual-Criticality)

After the partitioning of the dual-criticality task set is done, the hierarchical scheduling of the task classes follows directly the IS-Server approach of Section 2.4.2. Namely, we consider two global servers: one for HI criticality tasks (class S_{HI}) and one for LO criticality tasks (class S_{LO}). When the system is in LO mode, MC-IS-Server schedules tasks within each global server according to partitioned EDF, using the shortened deadlines for HI criticality tasks. For the partitioning methods MIQCP, MC-WF, and MC-WF-Max, the deadlines are down-scaled by x_j on each core j, while for MC-EY-WF, the deadlines are down-scaled by the respective per-task shortening factors. Upon switch to HI mode, the MC-IS-Server disables the LO server and it schedules the remaining HI criticality tasks according to partitioned EDF, using their original deadlines.

For the selection of the TDMA server schedule parameters, the task class demand and supply bound functions are computed as in Section 2.4.2.1, with the difference that for HI criticality tasks, we consider the shortened deadlines and their LO-level execution times. Derivation of the maximum TDMA cycle length P and the respective minimum server slots ℓ_k for a feasible MC-IS-Server schedule is performed by applying Algorithm 2 (see Section 2.4.2.2).

2.5.3 Extension to $K > 2$ Criticality Levels

The partitioning methods and the selection of the TDMA server schedule parameters of MC-IS-Server are also extensible to $K > 2$ criticality levels. The principal difference to the dual-criticality case lies in the deadline shortening phase, where multiple deadline shortening factors have to be assigned to each task which does not belong to the lowest criticality. The MC-EY-WF heuristic is applicable to $K > 2$ criticality levels and the details of this extension are given in [EY13]. The extended algorithm assigns multiple shortened deadlines to each task, depending on the number of execution modes in the system. The construction of the MC-IS-Server schedule follows then trivially from Algorithm 2 (see Section 2.4.2.2).

2.6 Evaluation

In the following, we evaluate the performance of IS-Server (Section 2.4) and MC-IS-Server (Section 2.5) based on extensive simulations with synthetic task sets. The task sets are generated based on the randomized procedure of Section 2.6.1. The proposed policies are compared against the optimal IS policy IS-DP-Fair and MC-IS-Fluid [HGA+15] in terms of schedulability in Section 2.6.2 and synchronous switches between task classes in Section 2.6.3. The evaluation allows a quantification of the schedulability loss due to (i) the IS constraint, (ii) increasing number of processing cores, (iii) increasing number of task classes or criticalities, and (iv) the static task partitioning to cores. Finally, Section 2.6.4 presents an extensive qualitative and quantitative comparison among (to the best of our knowledge) all scheduling policies that have been proposed in mixed-criticality literature and fit in the IS model. MC-IS-Server is shown to outperform or follow closely state-of-the-art policies in terms of schedulability.

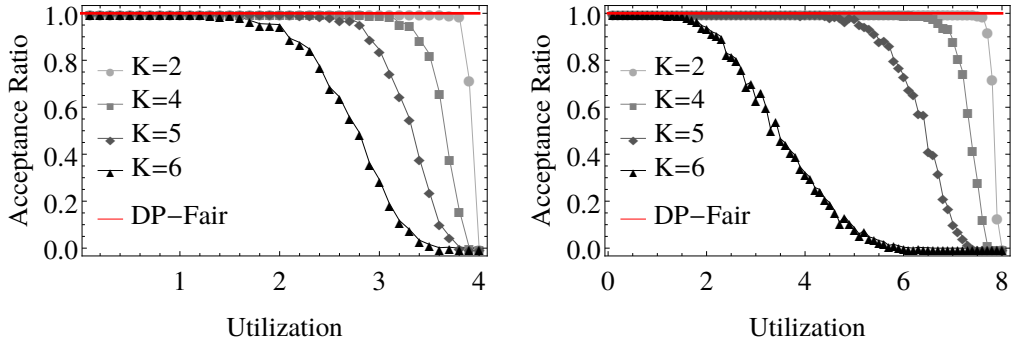
2.6.1 Random Task Set Generation

For the experiments, we synthetically generate periodic implicit-deadline task sets at different system utilization points. For *basic* (non mixed-criticality) IS task classes, we generate tasks in the following manner:

- Periods are randomly chosen from $\{x \in \mathbb{Z} \mid 2 \leq x \leq 2000\}$, with the exception of task sets with harmonic periods (Section 2.6.4), where periods are randomly chosen from $\{100, 200, 400, 800\}$, and task sets with equal periods, where periods are always equal to 1000.
- Task utilizations (densities) are uniformly chosen from $[0.01, 0.2]$.
- Tasks are equally likely to belong to task class $\{S_1, \cdots, S_K\}$.
- The number of task classes K varies in $\{2, 4, 5, 6\}$.

For *dual-criticality* task sets, we implement a widely used task set generator [BBD+12, GSHT13, LPG+14] with the following parameters:

- Probability of any task being HI criticality is 0.4 unless otherwise stated.
- The ratio $r = C_i(\text{HI})/C_i(\text{LO})$ is chosen uniformly from $[1, 5]$ for each HI criticality task.
- Periods and LO level utilizations for dual-criticality task sets are generated in the same way as for non mixed-critical task sets.



(a) IS-Server (WF-Max), 4 cores. (b) IS-Server (WF-Max), 8 cores.

Figure 2.6: Isolation Scheduling: Impact of number of classes on schedulability.

- The utilization (density) of any dual-criticality task set is defined as:

$$U := \max\left\{U_{\text{HI}}^{\text{LO}} + U_{\text{LO}}^{\text{LO}}, U_{\text{HI}}^{\text{HI}}\right\},$$

  where $U_{\chi_1}^{\chi_2}$ denotes the utilization of task class S_{χ_1} with low ($\chi_2 = \text{LO}$) or high ($\chi_2 = \text{HI}$) execution time bounds.

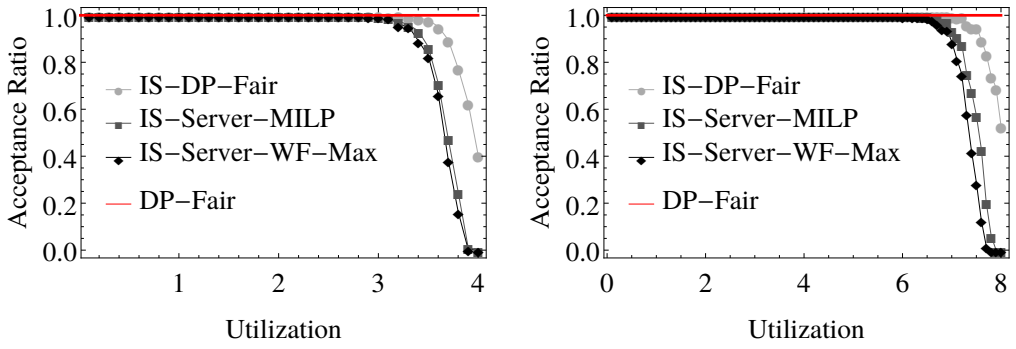
For all task sets, we perform experiments with system utilizations varying in the interval $[0.1, 4]$ (4 cores) or $[0.1, 8]$ (8 cores). Utilization is incremented in steps of 0.1, and for each utilization point we generate 500 task sets.

2.6.2 Schedulability

We intend to evaluate the ability to find schedulable solutions with the IS-Server and MC-IS-Server approaches for multi-class and mixed-criticality task sets. As reference, we consider the state-of-the-art fluid-based policies IS-DP-Fair (optimal in terms of schedulability for periodic implicit-deadline task sets under the IS constraint) and MC-IS-Fluid [HGA+15] as well as DP-Fair [LFS+10] and partitioned EDF-VD [BCLS14], which are *not* designed for Isolation Scheduling. We show that the schedulability loss due to the IS constraint can be negligible, but increases with increasing number of classes and cores. Additionally, we show that IS-Server can perform very closely to the optimal IS-DP-Fair depending on the task partitioning method.

2.6.2.1 Isolation Scheduling

Impact of Number of Classes. First, we evaluate the schedulability loss caused by enforcing the IS constraint. For this purpose, we generate 500 task sets for each system utilization and compute the fraction of task sets that are schedulable under the IS-Server approach (WF-Max



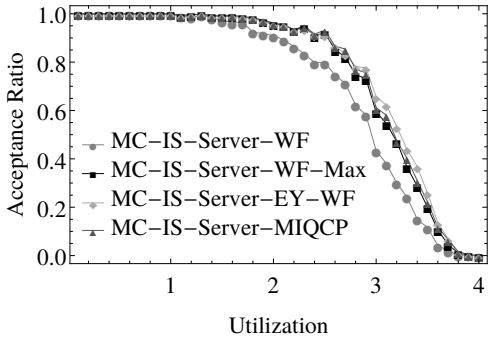
(a) $K = 4$ classes, 4 cores. (b) $K = 4$ classes, 8 cores.

Figure 2.7: Impact of IS constraint: Comparison of DP-Fair, IS-DP-Fair, IS-Server (with MILP or WF-Max task partitioning).

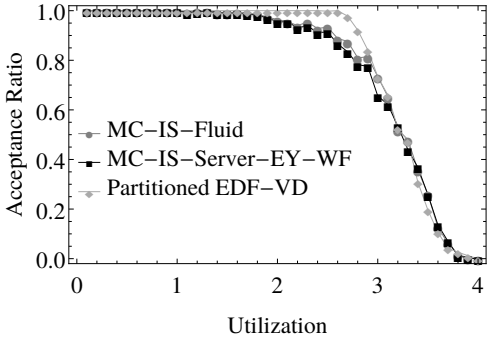
partitioning[6]) on $m = 4$ and $m = 8$ cores. The results are depicted in Figure 2.6(a) (4 cores) and Figure 2.6(b) (8 cores). For both configurations, the acceptance ratio by IS-Server decreases as the number of classes increases (for $K > 2$). This result matches with the theoretical analysis. According to Theorem 2.3, with increasing number of classes or cores, the speedup bound of IS-Server to preserve schedulability under the IS constraint increases, resulting in decreased schedulability.

Comparison of DP-Fair, IS-DP-Fair, IS-Server. Second, we compare schedulability under four different approaches: DP-Fair (optimal multi-core scheduling), IS-DP-Fair (optimal multi-core IS policy) and IS-Server, with the task partitioning in the last case being based on either the MILP formulation or the WF-Max heuristic. For brevity, we consider the case with $K = 4$ task classes, for which the results are presented in Figure 2.7(a) (4 cores) and Figure 2.7(b) (8 cores). For IS-Server, the WF-Max heuristic for task partitioning seems to work very efficiently, given the almost identical effect on schedulability compared to the MILP optimization formulation. The difference in schedulability between IS-DP-Fair and IS-Server (MILP) reaches up to 61.6% for 4 cores (at $U = 3.9$) and 69.2% for 8 cores (at $U = 7.9$) when the system is almost fully utilized. We attribute this difference mainly to the static task partitioning and the static time-triggered server scheduling of IS-Server. In practice, this cost in schedulability for IS-Server can be compensated by significantly less synchronous switches between task classes and zero task migrations, as opposed to IS-DP-Fair. A comparison based on this criterion is presented later. Additionally, we observe that the maximal deviation between IS-DP-Fair and IS-Server is reduced as the number of task classes, resp. cores, increases, dropping to 21.8% (4 cores) and only 3.6% (8 cores), for 6 classes.

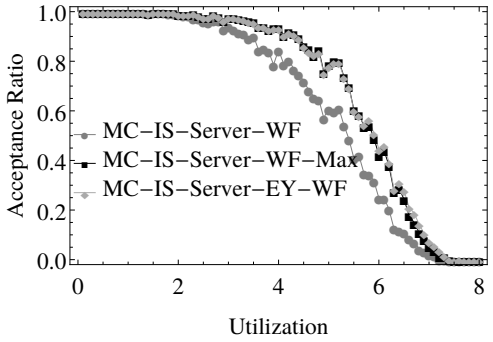
[6]We consider WF-Max partitioning because here, it provides the highest schedulability among the proposed heuristic approaches in Section 2.4.1.



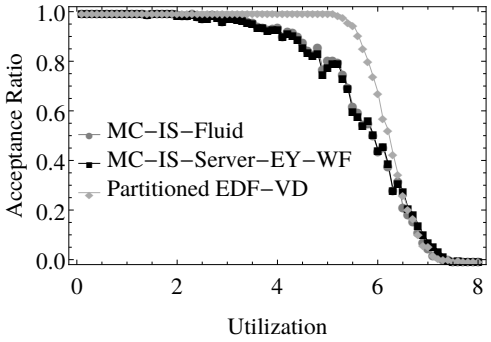
(a) MC-IS-Server with WF, WF-Max, EY-WF, MIQCP task partitioning, 4 cores.



(b) MC-IS-Fluid, MC-IS-Server (EY-WF) and partitioned EDF-VD, 4 cores.



(c) MC-IS-Server with WF, WF-Max, EY-WF task partitioning, 8 cores.



(d) MC-IS-Fluid, MC-IS-Server (EY-WF) and partitioned EDF-VD, 8 cores.

Figure 2.8: Impact of IS constraint in mixed-criticality systems: Comparison of MC-IS-Server (different partitioning approaches), MC-IS-Fluid, partitioned EDF-VD.

This implies a weaker impact of the number of classes on schedulability for IS-Server as compared to IS-DP-Fair.

2.6.2.2 Mixed-Criticality Isolation Scheduling

Comparison of partitioning policies for MC-IS-Server. MC-IS-Server has great practical potential due to its predictable and low-overhead time-triggered implementation. On the other hand, offline task partitioning can play a significant role in schedulability. Although several task partitioning approaches have been proposed in mixed-criticality literature, so far a quantitative comparison among them has not been available in the context of Isolation Scheduling. For this purpose, we compare the MIQCP formulation (Section 2.5.1.2), the heuristics MC-WF, MC-WF-Max (Section 2.5.1.3) and MC-EY-WF (Section 2.5.1.4) w.r.t. their effect on dual-criticality schedulability.

The results are presented in Figure 2.8(a) (4 cores) and Figure 2.8(c) (8 cores), and show MC-EY-WF outperforming the other partitioning

	MC-WF	MC-WF-Max	MC-EY-WF
4 cores	2969	4741	2387
8 cores	5928	11389	4980

Table 2.1: Number of task sets (out of 20,000 in total) that are deemed unschedulable during the partitioning phase of MC-IS-Server.

methods, closely followed by MIQCP and MC-WF-Max. MC-EY-WF enables maximal schedulability due to its freedom to select individual deadline shortening factors per task, which allows it to perform even better than the MIQCP approach (which still adopts a deadline shortening factor per core). The MC-WF-Max heuristic proves to be a good candidate for partitioning, mainly due to two key characteristics: (i) MC-WF-Max is the most effective method in identifying unschedulable task sets already from the partitioning phase. For the quad-core system (see Table 2.1), it deems 1.6x more task sets unschedulable compared to MC-WF and approximately 2x more compared to MC-EY-WF. For the octa-core system, it deems 1.9x more task sets unschedulable compared to MC-WF and 2.3x more compared to MC-EY-WF. This is attributed to its greedy behavior w.r.t. load balancing among cores and criticality levels, and it enables an early identification of unschedulable task sets; (ii) MC-WF-Max leads to server schedules with greater cycle lengths (on average, greater by 0.85% compared to MC-WF and MC-EY-WF, for 4 cores and 1% for 8 cores). This results in less frequent class switches and lower runtime overhead.

In summary, empirical evaluation shows consistently that the best partitioning methods are MILP and MC-WF-Max for IS-Server and MC-EY-WF for dual-criticality scheduling (MC-IS-Server). We consider this outcome important, since these methods can be applied offline to *any* partitioned IS policy. Given their tendency to distribute fairly the workload among task classes and cores, they are expected to perform well also for other hierarchical server scheduling policies that are not based on TDMA/EDF.

Comparison of MC-IS-Fluid, MC-IS-Server, Partitioned EDF-VD. To evaluate further the effect of the IS constraint on dual-criticality systems, we compare the MC-IS-Server approach (MC-EY-WF partitioning) to two state-of-the-art scheduling techniques: MC-IS-Fluid [HGA+15] and partitioned EDF-VD [BCLS14] (with the MC-PARTITION-UT-1 partitioning algorithm). The schedulability results are depicted in Figure 2.8(b) (4 cores) and Figure 2.8(d) (8 cores), respectively. Note that the schedulability of MC-IS-Fluid and MC-IS-Server is very close to partitioned EDF-VD for all utilizations in the case of 4 cores. For 8 cores, partitioned EDF-VD performs better for utilizations between 4 (50%) and 6 (75%) and similar to MC-IS-Server otherwise. We conclude that for dual-criticality systems the cost of enforcing the IS constraint is relatively low (though expected to rise for increasing number of

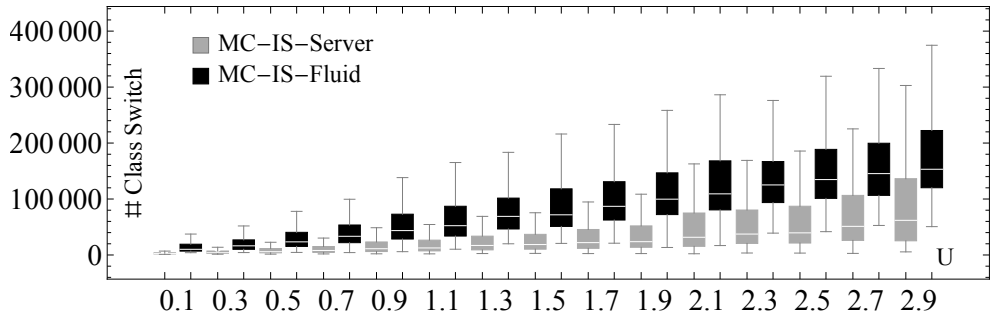


Figure 2.9: Distribution of task class switches for MC-IS-Server and MC-IS-Fluid for increasing system utilization (box-whisker-plot).

criticality levels). Additionally, MC-IS-Server incurs an almost negligible loss in schedulability compared to MC-IS-Fluid. We attribute the good performance of MC-IS-Server to the MC-EY-WF partitioning method, which allows individual deadline shortening factors per task, as opposed to MC-IS-Fluid, where a common factor for all tasks is applied. This result is encouraging given the applicability of MC-IS-Server to real-world systems, which does not need to come at the cost of schedulability.

2.6.3 Runtime Overhead

Conceptually, the advantage of adopting MC-IS-Server over a fluid approach for Isolation Scheduling, such as MC-IS-Fluid, lies in the reduced number of synchronous task class switches, which can cause a significant runtime overhead. Additionally, MC-IS-Server requires no task migrations, since tasks are statically partitioned to cores. We quantify the advantage of reduced task class switches in a dual-criticality quad-core setting in the following.

Specifically, we compare the required number of synchronous switches between two criticality levels for MC-IS-Server and MC-IS-Fluid. To this end, we consider a fixed, sufficiently large time interval $\Pi = 2 \times 10^6$ time units. The number of class switches within Π for the various task sets is presented for MC-IS-Server (MIQCP partitioning) and MC-IS-Fluid in Figure 2.9. Data in these plots are represented in the form of a box-whisker-plot to reveal the distribution of class switches for the 500 considered task sets at each utilization point (median, minimum, maximum). To enhance readability, only the points that lie within the inner fence of the distribution are shown. Also, we consider utilizations up to $U = 3$, since beyond it a significant amount of task sets are not schedulable. The results confirm the significant reduction of class switches by the MC-IS-Server approach. Given the median of class switches across all utilizations, MC-IS-Server achieves a 2.4 to 4.4-fold reduction in synchronous class switches compared to MC-IS-Fluid.

2.6.4 Comparison to State-of-the-art Mixed-criticality Isolation Scheduling

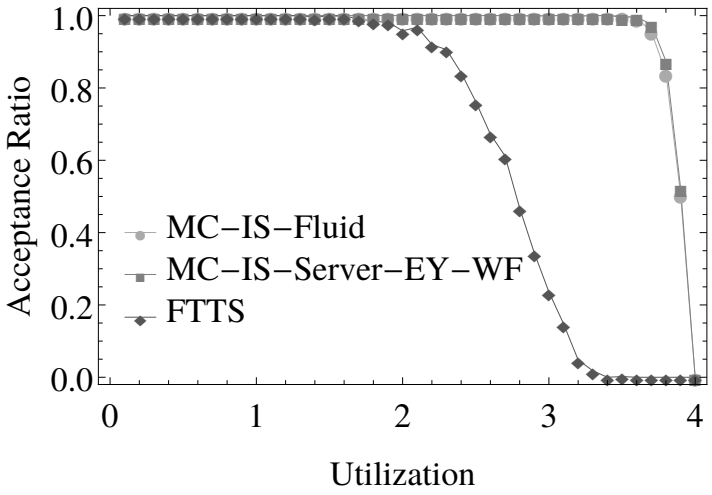
While we are the first to formalize the Isolation Scheduling model, existing works have also advocated the need for separation in multi-core environments and have proposed scheduling policies that comply with the IS model. In the following, we conduct a thorough comparison among those policies and the MC-IS-Server presented in this chapter. To the best of our knowledge, the currently existing Isolation Scheduling-compliant policies (besides the already mentioned MC-IS-Fluid) are the following:

- Flexible Time-Triggered scheduling with Synchronization points (FTTS, presented in Chapter 3), a partitioned time-triggered scheduling policy with globally synchronized time frames across all cores. Within each frame, tasks are executed in different, dynamically dimensioned sub-frames, such that only tasks of the same criticality run in parallel. Task partitioning and scheduling is based on a simulated-annealing heuristic approach. FTTS can be applied to task sets with arbitrary periods, however it performs best for *equal* or *harmonic* periods (see Section 3.8).

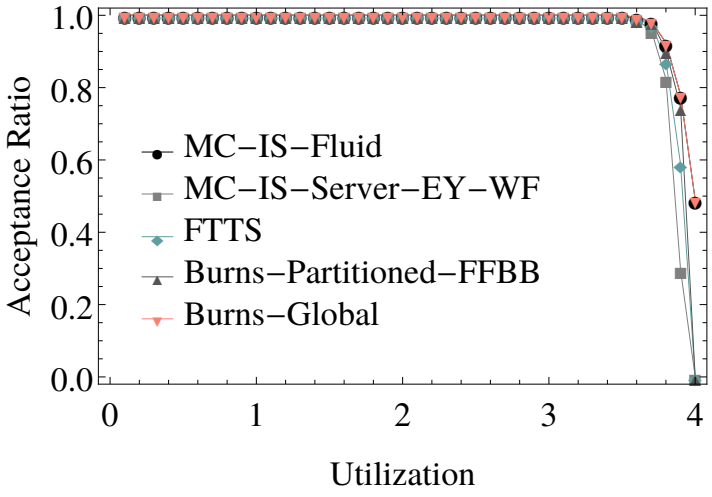
- Partitioned Cyclic Executive scheduling (CE-Partitioned) [BFB15], a cyclic-executive policy, where in each (major) time frame, tasks are distributed to (minor) frames depending on their criticality. The dimensions of the major and minor frames are fixed, as well as the partitioning of tasks to cores. A comparative study in [BFB15] shows that the best heuristic approach for task partitioning is first-fit with a branch-and-bound implementation (FFBB). CE-Partitioned can be applied to task sets with equal or harmonic periods. The methods of [BFB15] are directly applicable only to *equal-period* task sets.

- Global Cyclic Executive scheduling (CE-Global) [BB14], a global cyclic executive policy, similar to CE-Partitioned, yet without the limitation of fixed partitioning of tasks to cores. LO-criticality tasks are allowed to start executing once all HI-criticality tasks have finished execution. The schedulability test is based on a network flow criterion (Theorem 1, [BB14]), which can be validated in polynomial time by applying the Ford-Fulkerson algorithm [FF56]. This policy is applicable only to *equal-period* task sets.

In the following, we consider dual-criticality systems with harmonic or equal periods to enable a comparison between MC-IS-Server and the state-of-the-art IS approaches. To evaluate schedulability under FTTS, the FTTS framework (see Chapter 3) with the integrated partitioning and scheduling optimizer was used. For CE-Partitioned, we implemented FFBB for task partitioning and tested schedulability based on condition



(a) MC-IS-Fluid, MC-IS-Server, FTTS, harmonic periods.



(b) MC-IS-Fluid, MC-IS-Server, FTTS, CE-Partitioned, CE-Global, equal periods.

Figure 2.10: Comparison of MC-IS-Server to state-of-the-art IS policies (4 cores).

(6) of [BFB15]. For CE-Global, we constructed digraphs for each task set, as presented in [BB14], and used a Java implementation [SW07] of the Ford-Fulkerson algorithm to test the condition of Theorem 1 in [BB14].

Task Sets with Harmonic Periods. Here, the generated task sets have harmonic periods, randomly selected from set $\{100, 200, 400, 800\}$. The schedulability results for policies MC-IS-Server (MC-EY-WF), MC-IS-Fluid and FTTS are depicted in Figure 2.10(a). FTTS follows similar schedulability trends as in the experiments of Section 3.8 and is clearly dominated by MC-IS-Server and MC-IS-Fluid, which perform almost identically in this case. The difference in acceptance ratio between FTTS and the two IS policies reaches up to 100% for utilizations above

3.2. This significant schedulability loss can be explained as follows: (i) FTTS is a static, non-preemptive policy with fixed-size time frames. This is a limitation compared to the other two policies, which support preemptive scheduling. (ii) FTTS does not permit task migrations, which limits its schedulability compared to global policies such as MC-IS-Fluid. This experiment indicates a very good performance of MC-IS-Server with MC-EY-WF partitioning in the case of harmonic periods, since the schedulability gap to MC-IS-Fluid is indeed negligible.

Task Sets with Equal Periods. In the last experiment, the generated task sets have equal periods. The schedulability results for MC-IS-Server (MC-EY-WF), MC-IS-Fluid, FTTS, as well as the partitioned and global cyclic-executive policies are shown in Figure 2.10(b). As expected, schedulability under all policies is at its highest, with the first unschedulable task sets appearing only for system utilization above 3.5. That is because several factors that limit schedulability, e.g., no support for preemptions or migrations (FTTS, CE-Partitioned), fixed-length time frames (CE-Partitioned, MC-IS-Server), do not have an impact on equal-period task sets[7]. It is interesting to note that the two global IS policies, i.e., MC-IS-Fluid and CE-Global, perform identically for the special case of equal periods. Moreover, FTTS and CE-Partitioned perform very closely. Their schedulability conditions are indeed equivalent, with their only difference lying in the task partitioning method (simulated annealing versus FFBB heuristic). Based on the results, FFBB outperforms the simulated annealing approach at utilization points 3.8 and 3.9. Finally, MC-IS-Server follows closely the above policies. The schedulability loss compared to FTTS and CE-Partitioned could be possibly attributed to the pessimism of the schedulability analysis (Section 2.4.2.1).

Summarizing the results for harmonic and equal-period task sets, MC-IS-Fluid and CE-Global are the dominant policies in terms of schedulability among the state-of-the-art IS policies. As opposed to CE-Global, MC-IS-Fluid enforces no limitations to the task periods. MC-IS-Server performs very closely to MC-IS-Fluid, and better or close to other IS policies. This is a promising result for its future applicability in mixed-criticality industrial settings, where the case of task sets with harmonic periods is common.

2.7 Summary

In this chapter, we formalized the Isolation Scheduling (IS) model, a flexible abstraction for safety-critical real-time scheduling on resource-sharing multicores. The model is based on the common requirement for separation and enables interference-free scheduling of different *task*

[7]Notice e.g., the improvement in FTTS schedulability compared to the harmonic case.

classes, by enforcing the IS constraint: at any time, only one task class can run exclusively on the platform. We presented two IS-compliant scheduling policies, which support arbitrary number of task classes: IS-Server is based on hierarchical server scheduling, provides a practical approach towards implementing IS systems with low runtime overhead, yet satisfying schedulability; MC-IS-Server is an extension of IS-Server applicable to mixed-criticality systems.

Extensive simulations indicated that enforcing the IS constraint incurs low schedulability loss, which however tends to increase with increasing number of task classes, resp. criticality levels, and number of cores. Exploration of several task partitioning methods for the IS-Server and MC-IS-Server approaches revealed effective techniques, which can lead to comparable schedulability between server-based scheduling and optimal IS fluid-based scheduling approaches. Moreover, a comparative study among IS-compliant mixed-criticality scheduling policies validated the good performance of MC-IS-Server in the special cases of harmonic or equal-period task sets, for which it can outperform state-of-the-art partitioned approaches. Additionally, compared to existing approaches, MC-IS-Server enjoys wider applicability to task sets with multiple criticality levels, without posing limitations on the task periods. Based on the results, (MC-)IS-Server provides a viable solution to implementing IS-compliant systems with satisfying schedulability and reduced runtime overheads for synchronous inter-class switches and task migrations.

3

Bounding Intra-Criticality Interference

The designers of safety-critical systems are facing an increasing pressure for migrating from single-core to multi-core platforms for size, performance and cost purposes. However, scheduling mixed-criticality applications on existing multicores and providing safe worst-case response time bounds for the real-time applications is challenging given their timing interference on shared platform resources. The Isolation Scheduling model that was introduced in the previous chapter addresses this challenge by eliminating inter-criticality interference by construction. In this chapter, we focus on methods for bounding the intra-criticality interference.

For this, we introduce a combined analysis of computing, memory and communication scheduling. Our analysis targets cluster-based many-core systems with two shared resource classes, i.e., a shared memory within each cluster and a network-on-chip for inter-cluster communication and access to external memories. For such architectures, we propose: (1) An IS-compliant mixed-criticality scheduling policy based on a flexible time-triggered criticality-monotonic scheme. Tasks of the same criticality level are scheduled in a partitioned, non-preemptive fashion to reduce the complexity of interference analysis; (2) A response time analysis for the proposed scheduling policy, which takes into account the interferences from the two classes of shared resources; and (3) A design exploration framework and algorithms for optimizing the resource utilizations under mixed-criticality timing constraints. The applicability of the approach is demonstrated with a real-world avionics application. Its efficiency in finding schedulable solutions is evaluated through simulations with

synthetic task sets and compared against state-of-the-art mixed-criticality scheduling policies.

3.1 Introduction

Following the prevalence of multi-core systems in the electronics market, the field of embedded systems has experienced an unprecedented trend towards integrating multiple applications in a single platform. Migration to multicores is envisioned even in safety-critical domains, such as avionics and automotive. Applications in these domains are usually characterized by criticality levels, known as Safety Integrity Levels [iso11] or Design Assurance Levels [RTC12], which express the required protection against failure. For the integration of *mixed-criticality* applications in a common platform, many scheduling approaches have been proposed. However, most of them do not explicitly address the timing effects of resource sharing. Moreover, existing industrial certification standards require temporal isolation among applications of different criticality levels. For single-core systems, designers rely mainly on operating system and hardware-level partitioning mechanisms [ARI03]. For multi-core systems, there is currently no widely accepted solution on how isolation can be achieved in the presence of shared platform resources.

The Isolation Scheduling model that was introduced in the previous chapter allows to exploit task parallelism and resource sharing in a way that eliminates inter-criticality interference by construction. However, it does not solve the problem of bounding the timing effects of intra-criticality interference. This problem can become intractable even for scheduling policies like IS-Server, since micro-architectural features, such as the resource arbitration policy, the access overheads, the memory sub-system organisation, need to be known and accounted for under all interference scenarios. The fact that IS-Server supports dynamic task preemptions makes the problem of bounding delays due to resource contention even more challenging, since all preemption points need to be considered as well as the timing effects of preemptions, e.g., overhead for context switching and mutual eviction of cache lines from tasks running on the same core that leads to increased cache misses. To reduce the complexity of intra-criticality interference analysis, we introduce in this chapter a new IS-compliant policy with restricted support for task preemptions. Timing isolation on core level is achieved through time-triggered scheduling and on global level (shared resources) through dynamic inter-core synchronization with a barrier mechanism. The points of inter-core synchronization are defined by the scheduling strategy and vary in runtime to reflect the dynamic behavior of the applications and enable efficient resource utilization.

The scheduling policy targets cluster-based many-core platforms, such as the Kalray MPPA-256 [dDAB+13] and the STHorm/P2012 [MBF+12]. In such architectures, a cluster consists of several cores with a local shared memory and a private address space, while clusters are connected by specialized networks-on-chip (NoC). Tasks can be delayed when accessing local cluster shared resources not only because of concurrently executing tasks in the same cluster, but also because of data being received/sent from/to other clusters. Typically, the data arriving from other clusters are written to the local cluster memory with the highest priority, thus introducing timing delays to all tasks that try to access the local memory at the same time. Currently, no mixed-criticality scheduling and analysis methods exist that address such interference effects that are present in modern cluster-based architectures. This chapter extends the state-of-the-art by proposing a combined computation, memory and communication analysis and optimization framework for mixed-criticality systems deployed on cluster-based platforms.

Contributions. The main contributions of the chapter can be summarized as follows:

- We introduce an architecture abstraction for cluster-based many-cores with shared computing (processing cores), storage (local cluster memory, external DDR memory) and communication (NoC) resources.

- We propose an IS-compliant mixed-criticality multi-core scheduling policy. This follows a flexible time-triggered and criticality-monotonic scheme, which allows interference on the shared communication and storage infrastructure only among applications with the same criticality level within each cluster.

- We present a worst-case response time (WCRT) analysis for the scheduling policy which accounts for the blocking delays due to contention on the shared memory of a cluster and due to NoC data transfers. We assume that NoC flows are statically routed and regulated at the source node [LMJ+09]. The NoC is modelled and analyzed using network and real-time calculus [LBT01, TCN00].

- We propose a heuristic approach for finding an optimized mapping of mixed-criticality task sets to the cores of a cluster. It accounts for the interferences on the shared cluster memory due to concurrently executed tasks and inter-cluster NoC communication.

- We propose a heuristic approach for finding an optimized partitioning of task data to the memory banks of a cluster. It also accounts for the interferences on the shared memory banks.

- We combine the two inter-dependent heuristic approaches to find optimized mappings of tasks to cores and data to memory banks such that the workload distribution is balanced among cores and interference effects are minimized within a cluster.
- We demonstrate the applicability and efficiency of the design optimization approaches as well as the effect of memory sharing and inter-cluster communication on mixed-criticality schedulability using a real-world avionics application.
- We, also, compare the efficiency of the proposed scheduling policy against state-of-the-art policies based on simulations with synthetic task sets. The proposed policy is shown to outperform existing policies for harmonic workloads.

Outline. The chapter is organised as follows. Section 3.2 provides an overview of recent publications concerning mixed-criticality scheduling and resource interference analysis. Section 3.3 introduces the considered mixed-criticality task model and the many-core architecture abstraction. Section 3.4 describes the flexible time-triggered scheduling policy (FTTS) for mixed-criticality systems. Section 3.5 presents a method for worst-case response time analysis under FTTS, which considers explicitly the delays that each task suffers due to contention on the shared memory path by concurrently executed tasks within a cluster and by incoming NoC traffic. Section 3.6 suggests heuristic optimization approaches for (i) mapping tasks to the cores of a cluster, (ii) mapping data to memory banks, and (iii) an integrated approach for solving these two inter-dependent problems. In Section 3.7 we apply the developed optimization methods to a real-world case study and in Section 3.8 we compare the FTTS with other state-of-the-art mixed-criticality scheduling policies. Section 3.9 summarizes the main results of the chapter.

Note that to facilitate reading, a summary of the most important notations is presented in Table 3.1, while Figure 3.11 presents an overview of the design optimization flow and how the results of the individual sections are integrated into it.

3.2 Related Work

Mixed-Criticality Scheduling. Scheduling of mixed-criticality applications is a research field that has been attracting increasing attention in recent years. After the original work of Vestal [Ves07], which introduced the currently dominating mixed-criticality task model, several scheduling policies were proposed for both single-core and multi-core systems, e.g., [BLS10, BBD+12, BF11, BCLS14, Pat12]. For an up-to-date

compilation and review of these policies, we refer the interested reader to the study of Burns and Davis [BD16].

Among the policies for multicores, we highlight the ones that were designed for temporal isolation among criticality levels, which is a common requirement of certification authorities. Anderson et al. proposed scheduling mixed-criticality tasks on multicores, by adopting different strategies for different criticality levels and utilizing a bandwidth reservation server for temporal isolation [ABB09, MEA+10]. Tamas-Selicean and Pop presented an optimization method for task partitioning and time-triggered scheduling on multicores [TSP11], complying with the ARINC-653 standard [ARI03], the objective being the minimization of the certification cost. These works along with most existing multi-core scheduling policies, however, did not address explicitly the interference when tasks access synchronously shared platform resources and its effect on schedulability. We claim that this can be dangerous since Pellizzoni et al. have shown empirically that traffic (from peripheral devices) on the memory bus in commercial-off-the-shelf systems can increase the response time of a real-time task up to 44% [PBCS08], while Kim et al. empirically observed a 12x increase in the response time of a PARSEC benchmark when the benchmark was executed in parallel to memory-intensive tasks on a quad-core Intel Core i7 with a shared DRAM [KdNA+14]. Also, in Section 3.7, we show that platform parameters, such as the memory or NoC latency or the internal memory organisation, have a significant effect on the schedulability of mixed-criticality task sets. Note that [TSP11] considers inter-task communication via message passing, but the message transmission occurs asynchronously over a broadcast time-triggered bus such that no task's execution is blocked. This requires that no shared memory exists and that the bus schedule can be manually configured, assumptions which do not necessarily hold on commercial platforms.

The proposed scheduling policy in this chapter considers explicitly the timing effects of resource interference. Isolation among tasks with different criticality is achieved despite resource sharing and without need for hardware support, by enforcing the IS constraint. We present response time analysis and design optimization methods for cluster-based architectures where a task can experience interference from tasks executing concurrently in the same cluster, but also from inter-cluster NoC flows of data read/written from/to the cluster. For the analysis and design optimization, we use realistic models for resource (memory, NoC) arbitration based on the Kalray MPPA-256 architecture and compute real-time properties of the NoC flows such as delay and burst characteristics using network calculus.

Mixed-Criticality Resource Sharing. Recent works target at bounding the delay that high-criticality tasks suffer due to contention on shared

resources, while assuming partitioned task scheduling under traditional (single-criticality) policies, e.g., fixed priority. These methods differ from our policy in nature, since they accept interference among tasks with different criticality levels as long as it is bounded. For our IS-compliant scheduling policy, the delay imposed to high-criticality tasks by lower-criticality ones is invariably zero.

Existing methods for bounded interference on the shared memory have been already documented in Section 2.2. With regards to the NoC infrastructure, Tobuschat et al. [TAED13] implemented virtualization and monitoring mechanisms to provide independence among flows of different criticality. Particularly, using back suction [DE10], they targeted at maximizing the allocated bandwidth for lower criticality flows, while providing guaranteed service to the higher criticality flows. In our work, we assume real-time guarantees for all NoC flows. However, our work can be also combined with mixed-criticality NoC guarantees, as in the work of Tobuschat et al. [TAED13].

Data Partitioning. In this chapter, we also address the problem of mapping task data to the banks of a shared memory in order to optimize the memory utilization and minimize the interference among tasks of the same criticality. In the same line, Kim et al. [KLSP10] and Mi et al. [MFXJ10] proposed heuristics for mapping data of different application threads to DRAM banks to reduce the average thread execution times. Contrary to our work, these methods do not provide any real-time guarantees. Liu et al. implemented in [LCX+12] a bank partitioning mechanism by modifying the memory management of the operating system to adopt a custom page-coloring algorithm for the data allocation to banks, the objective being throughput maximization. Closer to our objective lies the work of Yun et al. [YMWP14], where the authors implemented a DRAM bank-aware memory allocator, using the page-based virtual memory system of the operating system to allocate memory pages of different applications/cores to private banks. The target is performance isolation in real-time systems, since partitioning DRAM banks among cores eliminates bank conflicts. In our work, we assume that bank sharing is inevitable, since tasks share access to buffers for communication purposes. We try to minimize the bank conflicts, though, through a combination of task scheduling and data mapping optimization. Note, that mechanisms like in [LCX+12, YMWP14] can be used to implement the memory mapping decisions of our design optimization method. Finally, the works of Reineke et al. [RLP+11] and Wu et al. [WKP13] rely on DRAM controllers to implement bank privatization schemes, where each core accesses its own banks. Similar to the work of Yun et al. [YMWP14], such controllers can ensure performance isolation, however they do not consider data sharing among tasks running on different cores. Additionally, they are hardware solutions, not applicable to commercial off-the-shelf platforms.

3.3 System Model

This section defines the task and platform model as well as a set of mixed-criticality scheduling requirements that are important for certification. The task model (Section 3.3.1) and scheduling requirements (Section 3.3.3) are based on the established mixed-criticality assumptions in literature [BD16], but also on an avionics case study which we addressed in the context of an industrial collaboration. The avionics application is described later in Section 3.7. The platform model is inspired mainly by the Kalray MPPA-256 architecture [dDvAPL14]. An overview of this architecture is provided in Section 3.3.2. For ease of presentation, all introduced notations in this and the following sections are summarized in Table 3.1.

3.3.1 Mixed-Criticality Task Model

We consider periodic mixed-criticality task sets $\tau = \{\tau_1, \ldots, \tau_n\}$ with criticality levels between 1 (lowest) and K (highest). A task is characterized by a 5-tuple $\tau_i = (T_i, D_i, \chi_i, \mathbf{C}_i, C_{i,deg})$, where:

- $T_i, D_i \in \mathbb{N}^+$ denote the task period and relative deadline.
- $\chi_i \in \{1, \ldots, K\}$ denotes the criticality level.
- $\mathbf{C}_i$ is a size-K vector of execution profiles, where $C_i(\ell) = (e_i^{min}(\ell), e_i^{max}(\ell), \mu_i^{min}(\ell), \mu_i^{max}(\ell))$ represents a lower and an upper bound on the execution time (e_i) and number of memory accesses (μ_i) of τ_i at level of assurance $\ell \leq \chi_i$. Note that execution time e_i denotes the computation or CPU time of τ_i *without* considering the time spent on fetching data from the memory. Such decoupling of the execution (computation) time and the memory accessing time is feasible on fully timing compositional platforms [WGR+09].
- $C_{i,deg}$ is a special execution profile for the cases when τ_i ($\chi_i < K$) runs in *degraded mode*. This profile corresponds to the minimum required functionality of τ_i so that no catastrophic effect occurs in the system. If execution of τ_i can be aborted without catastrophic effects, $C_{i,deg} = (0, 0, 0, 0)$.

For simplicity, we assume that the first job of all tasks is released at time 0 and that the relative deadline D_i of τ_i is equal to its period, i.e., $D_i = T_i$. Furthermore, the worst-case parameters of $C_i(\ell)$ are monotonically increasing for increasing ℓ and the best-case parameters are monotonically decreasing, respectively. Namely, the minimum/maximum interval of execution times and memory accesses in $C_i(\ell)$ is included in the corresponding interval of $C_i(\ell + 1)$. Note that the best-case parameters are only needed (i) to ensure that the minimum distance constraint of

dependent tasks is not violated and (ii) to obtain more accurate results from the response time analysis, as discussed in Section 3.5.

The bounds for the execution times and access numbers can be obtained by different tools. For instance, at the lowest level of assurance ($\ell = 1$), the system designer may extract them by profiling and measurement, as in [PBCS08]. At higher levels, certification authorities may use static analysis tools with more and more conservative assumptions as the required confidence increases. Note that the execution profile $C_i(\ell)$ for each task τ_i is derived only for $\ell \leq \chi_i$. For all $\ell > \chi_i$, $C_i(\ell) = C_{i,deg}$. That is because we assume that certification at level of assurance ℓ ignores all tasks with a lower criticality level. At runtime, if a task with criticality level greater than χ_i requires more resources than initially allocated to it, then to preserve schedulability, τ_i may run in degraded mode with execution profile $C_{i,deg}$.

The motivation behind defining a degraded execution profile, $C_{i,deg}$, is that in safety-critical applications, tasks typically cannot be aborted due to safety reasons. Several mixed-criticality scheduling policies in literature assume, nonetheless, that if a higher criticality task requires at runtime more resources than initially assigned to it (according to some optimistic resource allocation), then all lower criticality tasks can be aborted from that point on, permanently or temporarily (see study [BD16]). In our work, we assume that each task has a minimal functionality that *must* be executed under all circumstances so that no catastrophic effect occurs in the system. The corresponding execution requirements ($C_{i,deg}$) must be fulfilled by any mixed-criticality scheduling policy.

Finally, we define $\mathcal{D}ep(\mathcal{V}, \mathcal{E})$, a directed acyclic graph representing dependencies among tasks with equal periods. Each node $\tau_i \in \mathcal{V}$ represents a task of τ. A weighted edge $e \in \mathcal{E}$ from τ_i to τ_k implies that within a period the job of τ_i must precede that of τ_k. The weight $w(e)$ denotes the minimum time that must elapse from the completion of τ_i's execution until the activation of τ_j. If $w(e) = 0$, τ_j can be scheduled at earliest right after τ_i. We refer to $w(e)$ as the *minimum distance constraint*. Dependency graphs are a common consideration in scheduling to model e.g., data dependencies among tasks. In our work, we introduce the minimum distance constraint for dependent tasks. This is necessary to model scheduling constraints that stem from inter-cluster communication through a NoC in cluster-based architectures. Such constraints are discussed in Section 3.3.2 and 3.6.1.

Note that the considered task model differs from the one introduced in Section 2.3.2 in that it contains more detailed information on the tasks' execution (min. bound on execution time, min./max. bound on memory access number), it defines a degraded execution profile for lower-criticality tasks and it models possible dependencies among tasks.

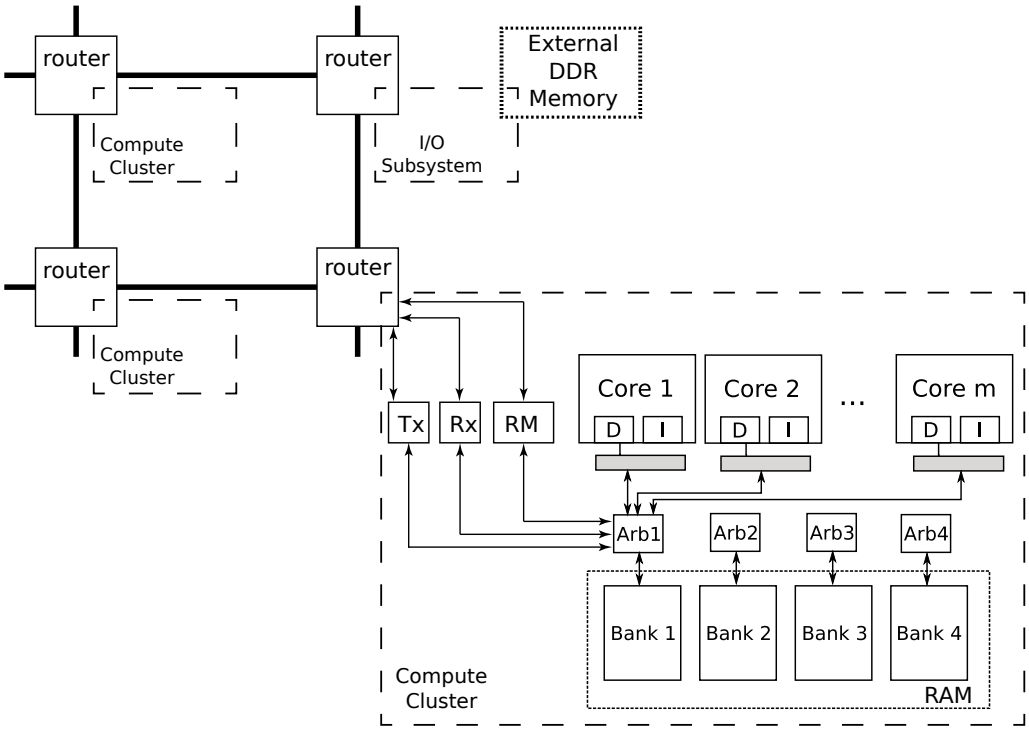


Figure 3.1: Shared memory architecture.

3.3.2 Resource-Sharing Platform Model

We consider a cluster $\mathcal{P}$ of m processing cores, $\mathcal{P} = \{p_1, \ldots, p_m\}$. Here, the cores are identical but there are no obstacles to extend our approach to heterogeneous platforms. The mapping of the task set τ to the cores in $\mathcal{P}$ is defined by function $\mathcal{M}_\tau : \tau \rightarrow \mathcal{P}$. Note that $\mathcal{M}_\tau$ is *not* given, but it will be determined by our approach in Section 3.6.

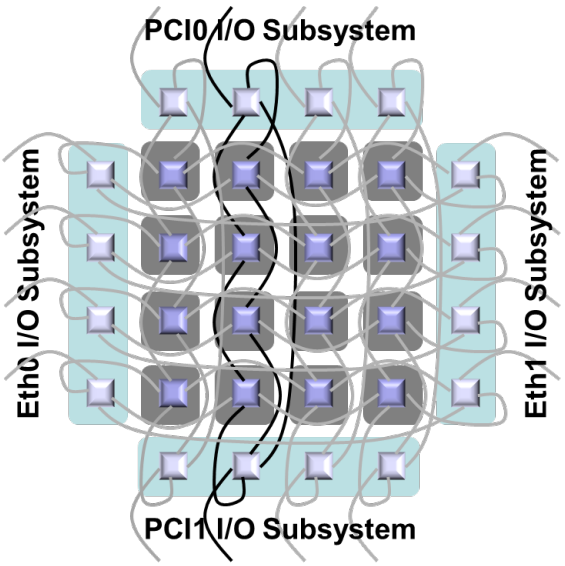
Each core in $\mathcal{P}$ has access to a private cache memory (we restrict our interest to data caches, denoted by 'D' in Figure 3.1), to a shared RAM memory and to an external DDR memory, which is local to another cluster. The shared cluster memory is organized in several banks. Each bank must have a sequential address space (not interleaved among banks). Under this assumption, two concurrently executed tasks on different cores can perform parallel accesses to the shared memory without delaying each other provided that they access different banks. We assume that each memory bank has a dedicated request arbiter. Also, each core has a private path (bus) to the shared memory. The private paths of the cores are connected to all bank arbiters, as depicted abstractly (for Arb1) in Figure 3.1. For the bank arbitration, we consider the class of round-robin-based policies, potentially with higher priority for some bank masters other than the cores in $\mathcal{P}$ (if such exist), e.g., the Rx interface in Figure 3.1. We assume that only one core can access a bank at a time and that once granted, a bank access is completed within a fixed time interval, T_{acc} (same for read/write operations and for all banks). In the meantime, pending

requests to the same bank from other cores stall execution on their cores until they are served.

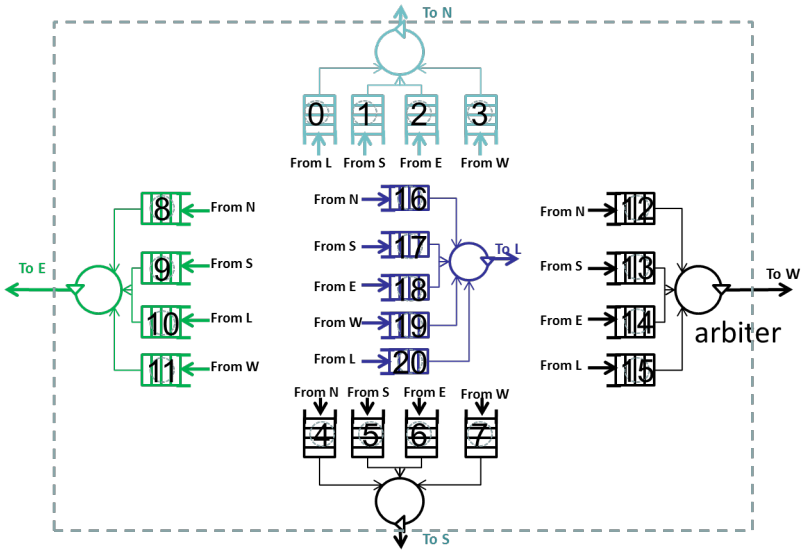
Additionally, the cores in $\mathcal{P}$ have access to external memories of remote clusters, which they access through a Network-on-Chip (NoC). Contention may occur in the NoC on router level every time two flows (virtual channels) need to be routed to the same outgoing link. Again, the assumed arbitration policy is round-robin at packet level.

The discussed abstract architecture model fits very well commercial manycore platforms, such as the Kalray MPPA-256 [dDvAPL14] and the STHorm/P2012 [MBF+12]. In the remainder of the chapter we motivate our models and methods based on the former architecture; therefore we look at it into greater detail in the following. Note that our response time analysis in Section 3.5 is valid for hardware platforms without timing anomalies, such as the fully timing compositional architecture which is defined in [WGR+09]. On such architectures, a locally worse behavior (e.g., a cache miss instead of a cache hit) cannot lead to a globally better effect for a task (e.g., reduced worst-case response time). The absence of timing anomalies allows the decoupling of execution and communication times during timing analysis. For a more detailed discussion on the property of timing compositionality, the interested reader is referred to [WGR+09] and for a rigorous definition of the term in resource-sharing systems to [HRW13], respectively. The MPPA-256 cores are fully timing compositional [dDvAPL14]. Note that our response time analysis and design optimization methods can be extended to cover other arbitration policies too, e.g., the first-ready first-come-first-serve, which is a common policy on COTS multicores [KdNA+14], on condition that the assumption of timing compositionality still holds.

Kalray MPPA Architecture. The Kalray MPPA-256 Andey processor integrates 256 processing cores and 32 resource management cores (denoted by 'RM' in Figure 3.1 and 3.3), which are distributed across 16 compute clusters and four I/O sub-systems. Each compute cluster includes 16 processing cores and one resource management core, each with private instruction and data caches. The processing cores and the resource management core implement the same VLIW architecture. However, the resource management core is distinguished by its connection to the NoC interfaces. Each I/O sub-system includes four resource management cores that share a single data cache, and no processing cores. Application code is executed on the compute clusters (processing cores), whereas the I/O sub-systems are dedicated to the management of external DDR memories, Ethernet I/O devices, etc. Each compute cluster and I/O sub-system owns a private address space. The DDR memory is only visible in the address space of the resource management cores of the I/O sub-system. Communication and synchronization among compute clusters and I/O sub-systems is



(a) MPPA-256 D-NoC topology.



(b) MPPA-256 D-NoC router model.

Figure 3.2: MPPA-256 D-NoC topology and router model.

supported by two explicitly routed, parallel networks-on-chip, the data (D-NoC) and the control (C-NoC) network-on-chip. Here, we consider only the D-NoC. This is dedicated to high-bandwidth data transfers and may operate with guaranteed services, thanks to non-blocking routers with flow regulation at the source nodes [LMJ+09], which is an important feature for the deployment of safety-critical applications.

Figure 3.2(a) presents an overview of the MPPA-256 architecture and the D-NoC topology. Each square in the figure corresponds to a switching node and interface of the D-NoC, for a total of 32 nodes: one per compute

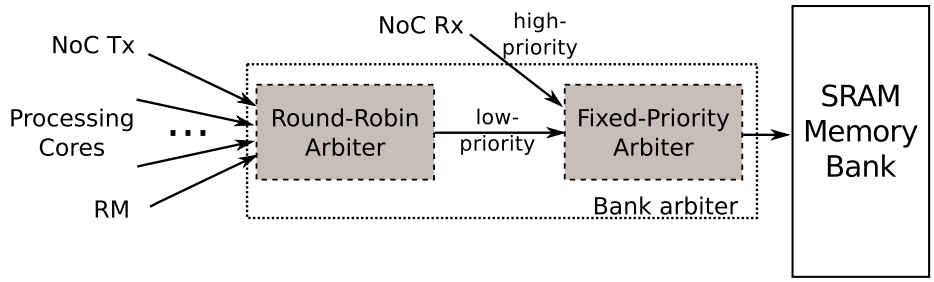


Figure 3.3: Memory bank request arbitration in an MPPA-256 cluster.

cluster (16 internal nodes) and four per I/O sub-system (16 external nodes). The I/O sub-systems are depicted on the four sides. The NoC topology is based on a 2D torus augmented with direct links between I/O sub-systems. Figure 3.2(b) depicts the internal structure of a D-NoC router. The MPPA-256 routers multiplex flows originating from different directions (input ports). Each originating direction (north N, south S, west W, east E, local node L) has its own FIFO queue at the output interface, so flows interfere on a node only if they share a link (output port) to the next node. This interface performs a round-robin arbitration at the packet granularity between the FIFOs that contain data to send on a link. NoC traffic through a router interferes with the memory buses of the underlying I/O sub-system or compute cluster only if the NoC node is a destination for the transfer.

Each of the compute clusters and the I/O sub-systems have a local on-chip memory, which is organized in 16 independent banks with a dedicated access arbiter for each bank. In the compute clusters, this arbiter always grants access to data received (Rx) from the D-NoC. That is, if an access request from D-NoC Rx arrives, it will be immediately served after the current access to the memory bank is completed. The remaining bandwidth is allocated to two groups of bus masters in round-robin fashion. The first group comprises the resource management core, the debug support unit and a DMA engine dedicated to data transmission (Tx) over the D-NoC. The second group is composed by the processing cores. Inside each group, the allocation policy is also round-robin. In practice, one may abstract the arbitration policy of memory banks as illustrated in Figure 3.3, where the debug support unit is omitted for simplicity. Within a compute cluster, the memory address mapping can be configured either as interleaved or as sequential. In the sequential address configuration, each bank spans 128 KB consecutive addresses. This is the assumed configuration in our work. By using linker scripts, one can statically map private code and data of each processing onto the different memory banks. This guarantees that no interference between processing

cores occurs on the memory buses or the arbiters of the memory banks. Such memory mapping optimization to eliminate inter-core interference will be considered in Section 3.6.

When a processing core from a compute cluster requires access to an external DDR memory, this is achieved through the I/O sub-systems, since compute clusters do not have direct access to the external memories. Each I/O sub-system includes a DDR memory controller, which arbitrates the access among different initiators according to a round-robin policy. The initiators include, among others, the D-NoC interfaces and the resource management cores of the I/O sub-systems.

Communication Protocol between a Compute Cluster and an I/O sub-system. In the remainder of this chapter, we consider the processing cores of one MPPA-256 compute cluster as the core set $\mathcal{P}$, introduced earlier in the abstract model. The cores in $\mathcal{P}$ share access to the SRAM memory banks of the cluster and can transfer data from/to an external DDR memory over the D-NoC. For the data transfer, we consider a specific protocol. This is illustrated in Figure 3.4 and described below:

- For each periodic task $\tau_i \in \tau$, which requires data from an external memory, there is a preceding task $\tau_{init,i}$ with the same period and criticality level, which initiates the data transfer. Specifically, $\tau_{init,i}$ communicates with a dedicated listener task which is executed in an I/O sub-system with access to the target DDR. $\tau_{init,i}$ sends a notification to the listener, including all relevant information for the DDR access, e.g., base address in DDR, data length, base address of allocated space in cluster memory. To send the notification, $\tau_{init,i}$ activates the cluster's NoC Tx interface. The transfer is asynchronous, i.e., $\tau_{init,i}$ can complete its execution after sending the notification, without expecting any acknowledgement of reception. We denote the maximum time required for the transmission of the notification packet(s) over the D-NoC as *worst-case notification time* (WCNT). This time is computed by our response time analysis method is Section 3.5.2.

- Upon reception of the notification, the remote listener: (i) decodes the request, (ii) allocates a D-NoC Tx DMA channel and a flow regulator, (iii) sets up the D-NoC Tx DMA engine with the transfer parameters, and (iv) initiates the transfer. From there the DMA engine will transmit the data through the D-NoC to the target compute cluster. We denote the maximum required time interval for actions (i)-(iv) as *worst-case remote set-up time* (WCRST). We assume that the WCRST can be derived based on measurements on the target platform.

- The transmitted packets follow a pre-defined route on the D-NoC

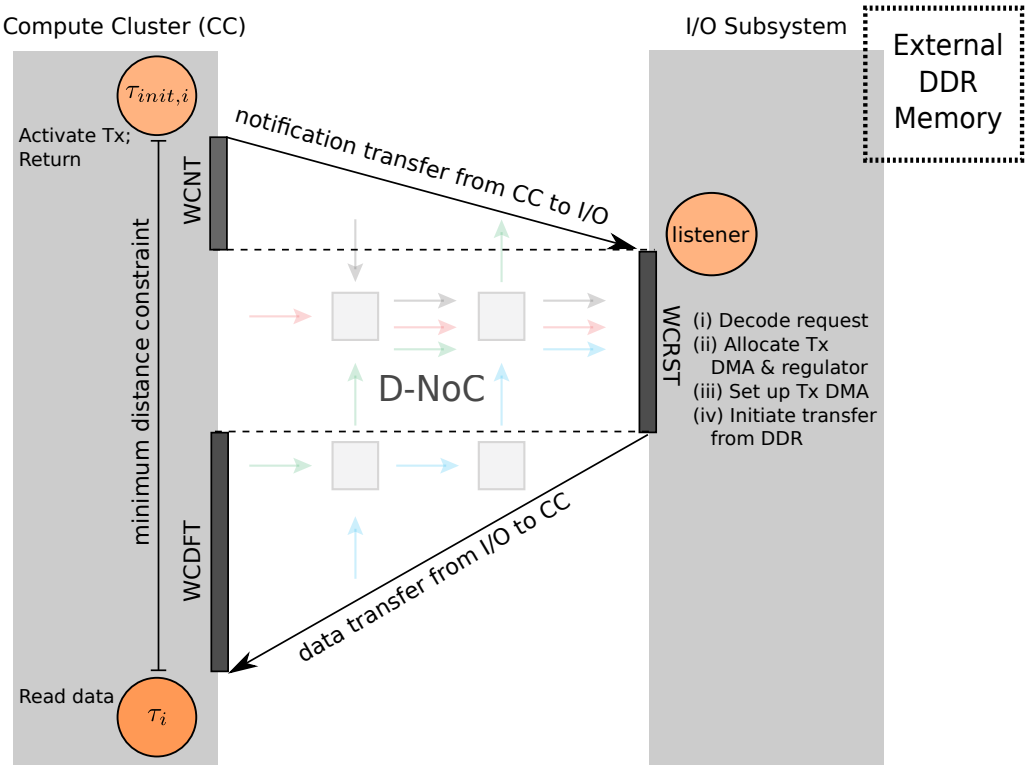


Figure 3.4: Communication protocol for reading data from external DDR memory.

before they are written to the local cluster memory by the cluster's NoC Rx interface. We denote the maximum time required for the data transmission over the D-NoC as *worst-case data fetch time* (WCDFT). We show how to compute this time for pre-routed and regulated flows in Section 3.5.2.

Note that for the considered protocol, $\tau_{init,i}$ should be scheduled early enough so that the required data are already in the cluster memory when τ_i is activated. This implies that for every pair of $(\tau_{init,i}, \tau_i)$ in τ, where $\tau_{init,i}$ initiates a data transfer from a remote memory for τ_i to use these data, an edge between $\tau_{init,i}$ and τ_i must exist in the dependency graph $\mathcal{D}ep$. The edge is weighted by the minimum distance constraint, which in this case, equals the sum of the worst-case notification time, WCNT, the worst-case remote set-up time, WCRST, and the worst-case data fetch time, WCDFT.

The communication protocol has been described for data transfer between a compute cluster and an I/O sub-system. Note, however, that a similar procedure takes place also for inter-cluster data exchange. Note also that for sending data to a remote memory, a task can directly initiate the transfer through the cluster's NoC Tx interface or by setting up a DMA transfer over the D-NoC. The transfer is assumed asynchronous and no handshaking with the remote cluster is required.

Notation	Meaning	Source
System Model - Section 3.3		
τ	Task set	Fixed
K	Number of criticality levels	Fixed
T_i, χ_i	Period and criticality level of task τ_i	Fixed
$C_i(\ell)$	Execution profile with lower and upper bounds on execution time (e_i) and number of memory (μ_i) of τ_i at level $\ell \leq \chi_i$	Fixed
$C_{i,deg}$	Execution profile of τ_i at level of assurance $\ell > \chi_i$	Fixed
$\mathcal{D}ep$	Dependency Graph	The edges (dependencies) are fixed, but their weight is determined from NoC analysis (Section 3.5.2)
$\mathcal{P}$	Set of processing cores on a target cluster	Fixed
T_{acc}	Memory access latency	Fixed
$\mathcal{M}_\tau$	Mapping of tasks of τ to cores of $\mathcal{P}$	Optimized in Section 3.6
Remote Fetch Protocol - Section 3.3.2		
$\tau_{init,i} \rightarrow \tau_i$	Initiating task for remote data fetch, task using fetched data	Fixed
WCNT	Worst-case time for transfer of notification from $\tau_{init,i}$ to listener in remote cluster	Computed in Section 3.5.2 (Eq. (3.12))
WCRST	Worst-case time for set-up of DMA transfer in remote cluster	Based on measurements
WCDFT	Worst-case time for complete transfer of data from remote cluster	Computed in Section 3.5.2 (Eq. (3.12))
FTTS Scheduling - Section 3.4		
H	FTTS cycle	Computed as hyper-period of τ
$\mathcal{F}$	Set of FTTS frames	Computed based on frame lengths
L_f	Length of FTTS frame f	Selected manually
$barriers(f,l)_k$	Worst-case length of k-th sub-frame in frame f at level ℓ	Computed for a given $\mathcal{M}_\tau$, $\mathcal{M}_{mem}$ in Section 3.5.1 (Eq. (3.6))
Response Time Analysis - Section 3.5		
$\mathcal{I}$	Memory interference graph	Consisting of $\mathcal{E}_1, \mathcal{E}_2$
$\mathcal{E}_1$	Mapping of tasks to memory blocks	Fixed
$\mathcal{E}_2$	Mapping of memory blocks to memory banks	Optimized in Section 3.6
$\mathcal{M}_{mem}$	Equivalent to $\mathcal{E}_2$	Optimized in Section 3.6
D	Mutual delay matrix	Computed in Section 3.5.1.1 and 3.5.1.2
`Rx`	Special node in $\mathcal{I}$ indicating a high-priority NoC Rx access	Added to $\mathcal{I}$ in Section 3.5.1.2
δ	Weight of edge between `Rx` and accessed memory block(s)	Computed in Section 3.5.2 (Eq. (3.13))
$WCRT_i(f,l)$	Worst-case response time of τ_i in frame f at level ℓ	Computed in Section 3.5.1.1 (Eq. (3.4))
$CWCRT_{p,k}(f,l)$	Worst-case response time of tasks executing on core p in the k-th sub-frame of frame f at level ℓ	Computed in Section 3.5.1.1, updated in Section 3.5.1.2
Design Optimization - Section 3.6		
$\|barriers\|_3$	3rd norm of *barriers* for all $f \in \mathcal{F}, \ell \in \{1,\ldots,K\}$	Computed for each candidate $\mathcal{M}_\tau$ (Eq. (3.6))
T_0	Initial temperature	Parameter of SA algorithm
a	Temperature decreasing factor	Parameter of SA algorithm
T_{final}	Final temperature	Parameter of SA algorithm
$time_{max}$	Time budget	Parameter of SA algorithm

Table 3.1: Important notation as defined/computed in each section.

3.3.3 Mixed-Criticality Scheduling Requirements

Under the above system assumptions, we seek a *correct* scheduling strategy for the mixed-criticality task set τ on $\mathcal{P}$, which will enable *composable* and *incremental certifiability*. We define below the properties of correctness, composable and incremental certifiability, which are key elements for a successful and economical certification process.

Definition 3.1. *A scheduling strategy is correct if it schedules a task set τ such that the provided schedule is admissible at all levels of assurance. A schedule of τ is admissible at level ℓ if and only if:*

- *the jobs of each task τ_i, satisfying $\chi_i \geq \ell$, receive enough resources between their release time and deadline to meet their real-time requirements according to execution profile $C_i(\ell)$,*
- *the jobs of each task τ_i, satisfying $\chi_i < \ell$, receive enough resources between their release time and deadline to meet their real-time requirements according to execution profile $C_{i,deg}$.* □

The term *resources*, in this context, refers to both processing time and communication time for accessing the shared memory and NoC.

Definition 3.2. *A scheduling strategy enables composable certifiability if all tasks of a criticality level ℓ are temporally isolated from tasks with lower criticality, for all $\ell \in \{1, \ldots, K\}$. Namely, the execution and access activities of a task τ_i must not delay in any way any task with criticality level greater than χ_i.* □

The requirement for composability enables different certification authorities to certify task subsets of a particular criticality level ℓ even without any knowledge of the tasks with lower criticality in τ. This is important when several certification authorities need to certify not the whole system, but individual parts of it. Each authority needs information on the scheduling of tasks with higher criticality level than the one considered. Such information can be provided by the responsible authorities for the higher-criticality task subsets.

Definition 3.3. *A scheduling strategy enables incremental certifiability if the real-time properties of the tasks at all criticality levels $\ell \in \{1, \ldots, K\}$ are preserved when new tasks with lower criticality level are added to the system.* □

This property implies that if the schedule of a task set τ is certified as admissible, the certification process will not need to be repeated if new, lower-criticality tasks are added later to the system. This is reasonable, since repeating the certification process of already certified tasks if the system is designed incrementally results in excessive costs.

Note that the above notion of correctness (Definition 3.1) is not new in mixed-criticality scheduling theory. On the other hand, the requirements for composable and incremental certifiability seem to be

crucial in safety-critical domains, e.g., avionics, for reducing the effort and cost of certification. Nonetheless, they are usually not considered explicitly in mixed-criticality scheduling literature (see study [BD16]). Exceptions are the works that are based on temporal partitioning as defined in the ARINC-653 standard [ARI03], such as the approach of Tamas-Selicean and Pop [TSP11] and the works that use servers for performance isolation among applications of different criticality levels, such as the approach of Yun et al. [YYP+12].

3.4 Flexible Time-Triggered Scheduling

In the following, we present the Flexible Time-Triggered and Synchronization-based (FTTS) mixed-criticality multi-core scheduling policy. FTTS is an IS-compliant policy, namely it allows only tasks with the same criticality level to be executed at any time. It is partitioned and non-preemptive, but it can support fixed preemption points (task splitting). In this section, we assume that an FTTS schedule for a particular task set and platform is given. For the given schedule, we describe the runtime behavior of the scheduler and introduce useful notation. We show how to determine an FTTS schedule (when it is not given) later, in Section 3.6.1.

The FTTS scheduling policy combines time and event-triggered task activation to enforce the IS constraint. A global FTTS schedule repeats over a *scheduling cycle* equal to the hyper-period H of the tasks in τ, i.e., the least common multiple of the periods. The scheduling cycle consists of fixed-length *frames* (set $\mathcal{F}$). Each frame is divided further into K flexible-length *sub-frames*. A sub-frame contains tasks of a single criticality level. The beginning of frames and sub-frames is synchronized among all cores. Frames start at predefined time points. The frame lengths can differ, but they are upper bounded by the minimum period in τ. Each sub-frame (except the first of a frame) starts once all tasks of the previous sub-frame complete execution across all cores. Synchronization is achieved dynamically via a barrier mechanism, for the sake of efficient resource utilization. The sub-frames within a frame are ordered in decreasing order of their criticality. Within a sub-frame, tasks are scheduled sequentially on each core following a predefined order, namely every task is triggered upon completion of the previous one.

Example 3.1. *An illustration of an FTTS schedule is given in Figure 3.5 for seven tasks with hyper-period $H = 200$ ms. Figure 3.5 depicts two consecutive scheduling cycles. The solid lines define the frames and the dashed lines the sub-frames, i.e., potential points where barrier synchronization is performed. The FTTS schedule has a cycle of $H = 200$ ms and is divided into four frames of equal lengths (50 ms), each with $K = 2$ sub-frames: the first for criticality 2 (high)*

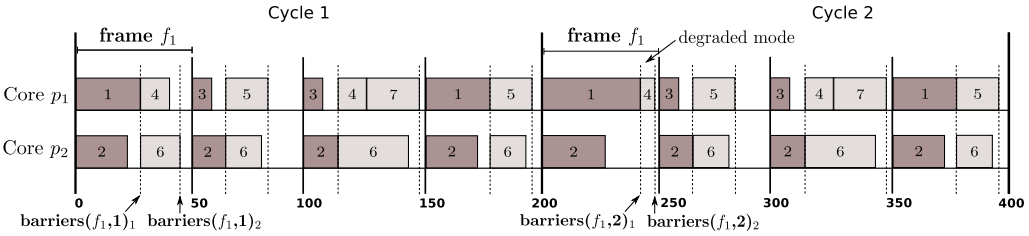


Figure 3.5: FTTS schedule for 2 cycles (dark annotation: criticality level 2, light: criticality level 1).

and the second for criticality 1 (low), respectively. A scheduling cycle includes H/T_i invocations of each task τ_i, i.e., the number of jobs of τ_i that arrive within a hyper-period.

At runtime, the length of each sub-frame varies based on the different execution times and accessing patterns that the executed tasks exhibit. For example, in Figure 3.5, the first sub-frame of f_1 finishes earlier when τ_1, τ_2 run w.r.t. their level-1, i.e., low-criticality profiles (cycle 1) than when at least one task runs w.r.t. its level-2, i.e., high-criticality profile (cycle 2). Despite this dynamic behavior, the worst-case sub-frame lengths can be computed offline for a given FTTS schedule by applying worst-case response time analysis under memory contention (see Section 3.5). Function $barriers : \mathcal{F} \times \{1, \ldots, K\} \rightarrow \mathbb{R}^K$ defines the worst-case length of all sub-frames in a frame, at a particular level of assurance. We denote the worst-case length of the k-th sub-frame of frame f at level ℓ as $barriers(f, \ell)_k$. Note that the k-th sub-frame of f contains tasks of criticality level $(K - k + 1)$. Also, ℓ corresponds to the highest level execution profile that the tasks of f exhibit at runtime. For $\ell > 1$, execution in certain sub-frames of f (with index $k > 1$) may be degraded.

Note that we use the notion of dynamic barriers for efficiency. On one hand, if sub-frames started at fixed time points, they would have to be dimensioned for the worst-case execution profiles of the tasks, so that all possible execution scenarios are covered. This is a common practice, e.g., when dimensioning the timing partitions in ARINC-653 architectures, where only the execution profile at each task's own criticality level is considered [TSP11]. This approach can be very inefficient since the higher criticality tasks may never reveal the corresponding execution profiles in practice. Resources, however, are reserved for them, leading to large idle times, during which the platform resources cannot be used by tasks of another criticality level. On the other hand, barrier synchronization occurs dynamically depending on the execution scenarios revealed at runtime, thus enabling efficient resource utilization.

Runtime behavior. Given an admissible FTTS schedule and the *barriers* function, the scheduler manages task execution on each core within each frame $f \in \mathcal{F}$ as follows (initially, $\ell_{max} = 1$):

- For the k-th sub-frame, the scheduler triggers sequentially the corresponding jobs following the predefined order. Upon completion of the jobs' execution, it signals the event and waits until the remaining cores reach the barrier.
- Let the elapsed time from the beginning of the k-th sub-frame until the barrier synchronization be t. Given ℓ_{max}:

$$\ell_{max} = \max \left\{ \underset{\ell \in \{1,\dots,K\}}{\operatorname{argmin}} \left\{ t \leq barriers(f,\ell)_k \right\}, \ell_{max} \right\}, \tag{3.1}$$

  the scheduler will trigger jobs in the next sub-frame such that tasks with criticality level lower than ℓ_{max} run in degraded mode.
- The two previous steps are repeated for each sub-frame, until the next frame is reached.

Note that the decision on whether a task will run in degraded mode affects only the current frame.

Admissibility. Let an FTTS schedule be constructed such that all H/T_i jobs of each task $\tau_i \in \tau$ are scheduled on the same core, in frames between their release times and deadlines and all dependency constraints hold. The FTTS schedule is ℓ-admissible if and only if it fulfills the following condition:

$$\sum_{k=1}^{K} barriers(f,\ell)_k \leq L_f, \quad \forall f \in \mathcal{F}, \tag{3.2}$$

where L_f denotes the length of frame f. If the condition holds for all frames $f \in \mathcal{F}$, all scheduled jobs can meet their deadlines at level of assurance ℓ. If the condition holds for all levels $\ell \in \{1, \cdots, K\}$, it follows that the FTTS schedule is admissible according to Definition 3.1. That is, it can be accepted by any certification authority at any level of assurance and the scheduling strategy is correct.

Composable and Incremental Certifiability. If different certification authorities certify task subsets of different criticality levels, then for composable certifiability (Definition 3.2), the authorities of lower-criticality task subsets need information on the resource allocation for the higher-criticality task subsets. For the FTTS scheduling strategy, this information is fully represented by function *barriers*. Hence, FTTS enables composable certifiability. Similarly, it enables incremental certifiability (Definition 3.3), since new tasks with lower criticality levels can be added to the system if there is sufficient slack time at the end of the FTTS frames. In this case, the new tasks do not affect the real-time properties of the tasks that were already scheduled in the system. It follows from the above that the computation of function *barriers* is necessary to evaluate if an FTTS schedule is admissible, but also as an interface among

certification authorities and system designers. In the next section, we show how to compute this function step-by-step, considering all possible task interferences for a given FTTS schedule.

3.5 Worst-Case Response Time Analysis

This section describes how to compute function *barriers* for a given FTTS schedule. For the computation of *barriers*, we need to bound the worst-case length of each sub-frame of the FTTS schedule at every level of assurance $\ell \in \{1, \ldots, K\}$. For this, first we perform worst-case response time (WCRT) analysis for every single task that is scheduled within the sub-frame. Second, based on the results of the first step, we derive the worst-case response time of the sequence of tasks which is executed on every core within the sub-frame (per-core WCRT, CWCRT). Once the last value is computed for all cores in $\mathcal{P}$, the worst-case sub-frame length follows trivially as the maximum among all per-core WCRTs.

The challenge in the above procedure lies in the computation of an upper bound for the response time of a task in a specific sub-frame. Note that for the timing compositional architectures which we consider, such as the MPPA-256, it is safe to bound the WCRT of a task by the sum of its worst-case execution (CPU) time and the worst-case delay it experiences due to memory accessing and communication [WGR+09]. The worst-case execution time of each task τ_i is known at different levels of assurance as part of its execution profile $\mathbf{C}_i$. However, to bound the second WCRT component, one needs to account for the interference on the shared cluster memory, i.e., interference from tasks running in parallel and from the NoC interface, when some of them try to access the same memory bank simultaneously. Therefore, to derive the WCRT of a task in a specific sub-frame, we need to model the possible interference scenarios based on the tasks that are concurrently executed and the NoC traffic patterns and then, to analyze the worst-case delay that such interference can incur to the execution of the task under analysis.

Section 3.5.1 shows how to bound the worst-case delay a task can experience on the shared memory path. For this, we consider as inputs: the mapping $\mathcal{M}_\tau$ of task set τ to the cores in $\mathcal{P}$, the mapping of task data to memory banks, the characteristics (patterns) of the incoming NoC traffic, and the memory access latency T_{acc}. Section 3.5.2 describes a method for NoC analysis, based on the network and real-time calculus [LBT01, TCN00], which enables us to compute a bound on all NoC incoming traffic patterns at the shared memory of a cluster. This bound is an input to the analysis of Section 3.5.1.

3.5.1 Bounding Delay on Cluster Memory Path

Within an FTTS sub-frame, we identify two sources of delay that a task may experience on the memory path based on the platform model of Section 3.3.2:

I. Blocking on a memory bank arbiter due to contention from other access requesters, specifically any other processing core or the NoC Tx DMA interface. Since contention is resolved among these requesters in a round-robin fashion, the task under analysis will have to wait for its turn in the round-robin cycle to be granted access to the memory bank.

II. Blocking on a memory bank arbiter due to contention from the NoC Rx interface. This requester has higher priority when accessing the memory, so in the worst case, the task under analysis will have to wait for all accesses of the NoC Rx interface to be served before it can gain access to the memory bank.

In the following, we model interference on the shared memory in the form of a *memory interference graph*. Based on this graph, we compute the maximal delay that each task can cause to another when they are executed in parallel and in the presence of incoming traffic from the NoC Rx interface. This way, we are able to estimate the WCRT of each task within the FTTS sub-frame and subsequently, the global worst-case sub-frame length.

3.5.1.1 Memory Interference among Requesters with Equal Priorities

To model the inter-task interferences due to contention on a round-robin-arbitrated memory controller, we introduce a graph representation, called the *memory interference graph* $\mathcal{I}(\mathcal{V}, \mathcal{E})$. We define $\mathcal{V} = \mathcal{V}_\tau \cup \mathcal{V}_{BL} \cup \mathcal{V}_B$, where $\mathcal{V}_\tau$ represents all tasks in τ (running on processing cores and NoC Tx), $\mathcal{V}_{BL}$ represents all memory blocks BL accessed by τ, i.e., the tasks' instructions, data and communication buffers, and $\mathcal{V}_B$ represents all banks B of the shared memory. Each memory block node is annotated with a corresponding size in bytes. Respectively, each memory bank node is annotated with the bank's capacity in bytes. $\mathcal{I}$ is composed by two sub-graphs: (i) the bipartite graph $\mathcal{I}_1(\mathcal{V}_T \cup \mathcal{V}_{BL}, \mathcal{E}_1)$, where an edge $e \in \mathcal{E}_1$ from $\tau_i \in \mathcal{V}_T$ to $bl_j \in \mathcal{V}_{BL}$ with weight $w(e)$ implies that task τ_i performs at maximum $w(e)$ accesses to memory block bl_j per execution, and (ii) the bipartite graph $\mathcal{I}_2(\mathcal{V}_{BL} \cup \mathcal{V}_B, \mathcal{E}_2)$, where an edge $e \in \mathcal{E}_2$ from $bl_j \in \mathcal{V}_{BL}$ to $b_k \in \mathcal{V}_B$ denotes the allocation of memory block bl_j in exactly one memory bank b_k. Note that the weighted sum over all outgoing edges of a task τ_i equals the memory access bound of its execution profile at its own criticality level, i.e., $\mu_i^{max}(\chi_i)$. The weights can be, however, reduced if WCRT analysis is performed for lower levels of assurance, $\ell < \chi_i$. In this case, the weighted sum over the outgoing edged of τ_i equals the (more

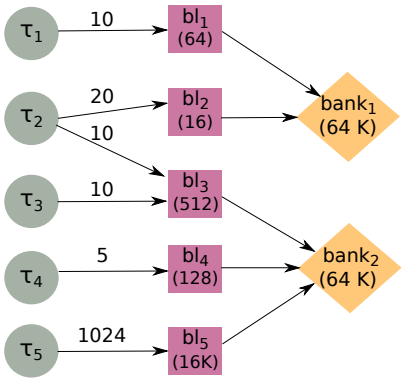


Figure 3.6: Memory Interference Graph $\mathcal{I}$ for a dual-bank memory.

	τ_1	τ_2	τ_3	τ_4	τ_5
τ_1	0	$10{\cdot}T_{acc}$	0	0	0
τ_2	$10{\cdot}T_{acc}$	0	$10{\cdot}T_{acc}$	0	0
τ_3	0	$10{\cdot}T_{acc}$	0	0	0
τ_4	0	0	0	0	$5{\cdot}T_{acc}$
τ_5	0	0	0	$5{\cdot}T_{acc}$	0

Table 3.2: Mutual delay matrix D for round-robin arbitration policy for graph of Figure 3.6.

optimistic) $\mu_i^{max}(\ell)$, which enables tighter WCRT analysis for the specific level of assurance.

Definition 3.4. *Tasks τ_i and τ_j are interfering if and only if $\exists k, l, r \in \mathbb{N}^+$: $(\tau_i, bl_k) \in \mathcal{E}_1$, $(\tau_j, bl_l) \in \mathcal{E}_1$ and $(bl_k, b_r) \in \mathcal{E}_2$, $(bl_l, b_r) \in \mathcal{E}_2$, i.e., they access blocks in the same memory bank.* □

Example 3.2. *Figure 3.6 presents a memory interference graph for a set of five tasks, accessing in total five memory blocks. The memory blocks can be allocated to two banks. Ellipsoid, rectangular and diamond nodes denote tasks, memory blocks and banks, respectively. Note that for the depicted mapping of memory blocks to banks, tasks τ_1 and τ_2 are interfering, whereas τ_1 and τ_3 or τ_4 or τ_5 are not. Interfering tasks can delay each other when executed in parallel.*

In the general problem setting, the mapping of memory blocks to banks, $\mathcal{M}_{mem} : BL \rightarrow B$ ($\mathcal{E}_2$ of $\mathcal{I}$), is not known, but derived by our optimization approach (see Section 3.6). Here, however, for the WCRT analysis we assume that it is fixed. Based on it, we introduce the *mutual delay matrix*, D. D is a two-dimensional matrix ($n \times n$), where $D_{i,j}$ specifies the maximum delay that task τ_i can suffer when executed concurrently with τ_j. $D_{i,j}$ is positive if τ_i and τ_j ($i \neq j$) are (i) of the same criticality level, i.e., potentially concurrently executing in an FTTS schedule, *and* (ii) interfering, i.e., accessing memory blocks in at least one common bank.

For the computation of D, we need to consider the bank arbitration policy, which in the case of MPPA-256 and for the memory requesters that we consider is round-robin. For the round-robin policy, each access request from a task τ_i can be delayed by at most one access from any other concurrently executed task that can read/write from/to the same memory bank which τ_i is targeting. That is because we assume that each core has at most one pending request at a time (Section 3.3). In other words, τ_i can be maximally delayed by a concurrently executed task τ_j for the duration of τ_j's accesses to a shared memory bank, provided that τ_j's accesses are

not more that τ_i's accesses to this bank. If τ_i and τ_j share access to more than one memory bank, the sum of potential delays across the banks has to be considered. This yields:

$$D_{i,j} = \sum_{\substack{b,bl:(\tau_i,bl)\in\mathcal{E}_1 \\ \wedge(bl,b)\in\mathcal{E}_2}} \sum_{\substack{bl':(\tau_j,bl')\in\mathcal{E}_1 \\ \wedge(bl',b)\in\mathcal{E}_2}} \min\left\{w((\tau_i,bl)), w((\tau_j,bl'))\right\} \cdot T_{acc}\ . \tag{3.3}$$

Example 3.3. *For the memory interference graph of Figure 3.6, we assume that tasks τ_1, τ_2, τ_3 are of criticality level 2, whereas τ_4 and τ_5 of criticality level 1. In this case, Table 3.2 presents the mutual delay matrix D for the memory interference graph. Matrix D represents the worst-case mutual delays when τ_1, τ_2, τ_3 are executed in parallel in the same FTTS sub-frame. Tasks τ_4 and τ_5 are assumed to run in parallel, too.*

We use matrix D to compute the worst-case length of the k-th sub-frame of an FTTS frame f at level of assurance ℓ. According to the notation introduced in Section 3.4, the computed length corresponds to $barriers(f,\ell)_k$. First, we compute the WCRT of every task τ_i executed in the k-th sub-frame of frame f at level of assurance ℓ as:

$$WCRT_i(f,\ell) = e_i^{max}(\ell) + \mu_i^{max}(\ell) \cdot T_{acc} + d_i(f,\ell), \tag{3.4}$$

namely, as the sum of its worst-case execution time, the total access time of its memory accesses under no contention, and the worst-case delay it encounters due to contention. This last term, $d_i(f,\ell)$, is defined as:

$$d_i(f,\ell) = \min\left\{ \sum_{\tau_j\in parallel(\tau_i,f)} D_{i,j},\quad \mu_i^{max}(\ell)\cdot(m-1)\cdot T_{acc} \right\}, \tag{3.5}$$

where function $parallel : \tau \times \mathcal{F} \rightarrow S_\tau$ defines a set of tasks $S_\tau \subseteq \tau$ that are executed in parallel to task τ_i (on different cores) in frame f and m is the number of interfering requesters with equal priorities. Note that $\mu_i^{max}(\ell)\cdot(m-1)\cdot T_{acc}$ is a safe upper bound on the delay that a task can suffer due to contention under round-robin arbitration. In Eq. (3.5), we take the minimum of the two terms to achieve a more accurate estimation. This is useful in cases e.g., where (some of) the parallel executed tasks with τ_i are scheduled sequentially on a single core. It is then possible that not all of them can delay τ_i on the memory path. In such cases, the second bound may be tighter than the one based on matrix D.

Example 3.4. *For a demonstration of the use of the above equations, let us consider the FTTS schedule of Figure 3.5. In the 1st sub-frame of frame f_1, tasks τ_1 and τ_2 with criticality level $\chi_1 = \chi_2 = 2$ are executed in parallel on $m = 2$ processing cores. According to the definition of function parallel, it holds that $parallel(\tau_1, f_1) = \{\tau_2\}$ and $parallel(\tau_2, f_1) = \{\tau_1\}$. The accessing behavior of tasks τ_1 and τ_2 is described by the memory interference graph of Figure 3.6. The*

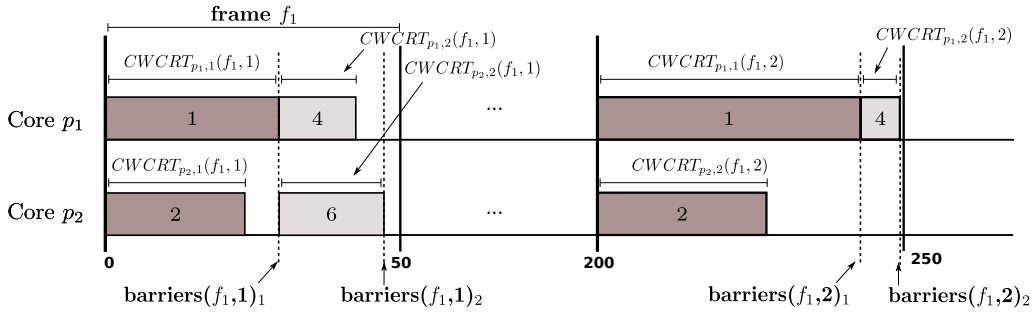


Figure 3.7: Computation of $barriers(f_1, \ell)_k$ for $\ell = \{1, 2\}$ and $k = \{1, 2\}$ for the FTTS schedule of Figure 3.5.

depicted weights on the edges of the memory interference graph, i.e., the number of accesses that each task performs to the respective memory blocks, are derived at level of assurance $\ell = 2$. Based on the graph of Figure 3.6, tasks τ_1 and τ_2 can interfere only on $bank_1$ because τ_1 accesses block bl_1 and τ_2 accesses block bl_2, with both blocks being mapped to the same memory bank $bank_1$. Given this, Eq. (3.3) yields $D_{1,2} = D_{2,1} = 10 \cdot T_{acc}$. In other words, task τ_1 can delay τ_2 at most 10 times when accessing the shared memory, by issuing interfering access requests to the same bank. The same also holds for the maximal delay that τ_2 can cause to τ_1. These results can be seen in the mutual delay matrix D, which is already given in Table 3.2. At a next step, by applying Eq. (3.5), we compute the worst-case delay that task τ_1 can experience due to memory contention in frame f_1 of the FTTS schedule of Figure 3.5, at level of assurance $\ell = 2$:

$$d_1(f_1, 2) = \min\left\{D_{1,2}, \mu_1^{max}(2) \cdot (2-1) \cdot T_{acc}\right\} = \min\left\{10 \cdot T_{acc}, 10 \cdot T_{acc}\right\}.$$

Similarly for task τ_2,

$$d_2(f_1, 2) = \min\left\{D_{2,1}, \mu_2^{max}(2) \cdot (2-1) \cdot T_{acc}\right\} = \min\left\{10 \cdot T_{acc}, 30 \cdot T_{acc}\right\}.$$

Hence, $d_1(f_1, 2) = d_2(f_1, 2) = 10 \cdot T_{acc}$. Namely, the WCRT of both tasks is augmented by $10 \cdot T_{acc}$ as a result of the inter-core interference on the shared memory.

Once the WCRT of all tasks in the k-th sub-frame of frame f are computed at all levels of assurance $\ell \in \{1, \ldots, K\}$, we derive the WCRT of the task sequence on each core p, $CWCRT_{p,k}(f, \ell)$, by summing up the WCRTs of the tasks that are mapped on p in the particular sub-frame. For an illustration of the notation used, please refer to Figure 3.7. It follows trivially that:

$$barriers(f, \ell)_k = \max_{1 \le p \le m} \left\{CWCRT_{p,k}(f, \ell)\right\}. \tag{3.6}$$

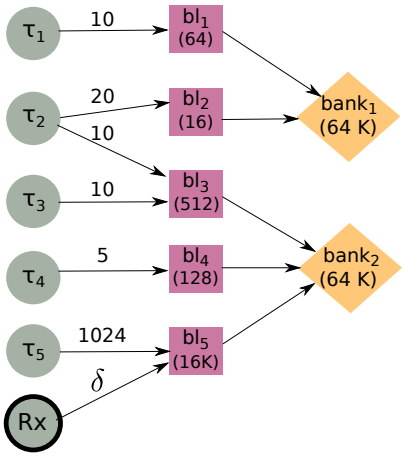


Figure 3.8: Memory Interference Graph $\mathcal{I}$ for a dual-bank memory with higher-priority interference from the NoC Rx requester.

	τ_1	τ_2	τ_3	τ_4	τ_5
τ_1	0	$10{\cdot}T_{acc}$	0	0	0
τ_2	$10{\cdot}T_{acc}$	$\boldsymbol{\delta \cdot T_{acc}}$	$10{\cdot}T_{acc}$	0	0
τ_3	0	$10{\cdot}T_{acc}$	$\boldsymbol{\delta \cdot T_{acc}}$	0	0
τ_4	0	0	0	$\boldsymbol{\delta \cdot T_{acc}}$	$5{\cdot}T_{acc}$
τ_5	0	0	0	$5{\cdot}T_{acc}$	$\boldsymbol{\delta \cdot T_{acc}}$

Table 3.3: Mutual delay matrix D for round-robin arbitration with higher priority for Rx for graph of Figure 3.8.

3.5.1.2 Memory Interference from Requesters with Higher Priority

When introducing the memory interference graph in Section 3.5.1.1, we implicitly assumed that all memory blocks, i.e., task instructions, private data and communication buffers, fit into the memory banks of the shared cluster memory so that the mapping of blocks to banks can be decided offline and no remote access to other compute clusters or I/O sub-systems is required. However, in realistic applications, such as the flight management system which is used for evaluation in Section 3.7, there may be tasks which need access to databases or generally, complex data structures that do not fit in the shared memory of a compute cluster. We assume that these data structures are stored in the external DDR memories and parts of them (e.g., some database entries), which can fit into the cluster memory together with the data of the remaining tasks, are fetched whenever required.

One possible implementation of a remote data fetch protocol has been described for the MPPA-256 platform and similar many-core architectures in Section 3.3.2. Here, we focus on its last step, namely the actual transfer of data packets from the I/O sub-system to the cluster over the NoC. During the data transfer, the NoC Rx interface tries to write to the shared cluster memory every time a new packet arrives at the cluster. Therefore, any task in the cluster attempting to access the memory bank where the remote data are stored will experience blocking due to the higher priority of the NoC Rx interface (see Figure 3.3). In the worst case, the task will stall for the duration of the whole remote transfer before it is granted access to the memory bank.

To model the interference from the NoC Rx interface: **First**, we extend the memory interference graph, as shown in Figure 3.8. The task with

the bold outline represents the DMA transfer from the I/O sub-system (resp. another compute cluster) to the compute cluster. There can be arbitrarily many tasks representing DMA transfers. The weight δ of the newly added edge from `Rx` to the target memory block can be derived as the minimum between (i) the total number of fetched packets and (ii) the maximum number of packets that can be fetched over the NoC in the time interval of one FTTS frame. To compute δ for a particular data flow, we need information about the flow regulation, the flow route and the NoC configuration. We show how to use this information to derive δ in Section 3.5.2 (Eq. (3.13)).

Second, we update the mutual delay matrix D by setting entries $D_{j,j}$ to $\delta \cdot T_{acc}$ for all tasks τ_j that are interfering with `Rx`, as shown in Table 3.3. This applies to all tasks independently of their criticality level or whether they run in parallel to `Rx`, and it expresses that an interfering task τ_j can be delayed by δ higher-priority requests any time it executes. The update helps during memory mapping optimization to distribute the memory block(s) to which `Rx` writes to different banks compared to all remaining memory blocks.

Third, we update the computed in Section 3.5.1.1 per-core WCRTs, $CWCRT_{p,k}(f, \ell)$ for all $p \in \mathcal{P}$. Recall that $CWCRT_{p,k}(f, \ell)$ in a given FTTS schedule denotes the sum of WCRTs of the tasks that are mapped on core p in the k-th sub-frame of frame f, given the task execution profiles at level of assurance ℓ. In the following, we consider the communication protocol between a compute cluster and an I/O subsystem, as described in Section 3.3.2. For every task $\tau_i \in \tau$, which uses remote data from the DDR, and its preceding task $\tau_{init,i}$, which initiates the transfer from the I/O sub-system, let f_{prec} and f_{succ} be the FTTS frames in which $\tau_{init,i}$ and τ_i are scheduled, respectively. Since tasks τ_i and $\tau_{init,i}$ have the same criticality level, χ_i, it follows that they are scheduled in the k-th sub-frame of frames f_{prec} and f_{succ}, respectively, where $k = K - \chi_i + 1$. For instance, in a dual-criticality system ($K = 2$), if τ_i and $\tau_{init,i}$ have criticality level $\chi_i = 2$, they will be scheduled in the 1st sub-frame of their corresponding FTTS frames. For the update of $CWCRT_{p,k}(f, \ell)$, we distinguish two cases, depending on whether frames f_{prec} and f_{succ} are equal or not. Particularly, for every core $p \in \mathcal{P}$:

Case 1 If $f_{prec} = f_{succ}$ *and* in the k-th sub-frame of f_{prec} there are tasks on p scheduled between $\tau_{init,i}$ and τ_i, which are interfering with `Rx`, then $CWCRT_{p,k}(f_{prec}, \ell)$ is increased by $\delta \cdot T_{acc}$, for all $\ell \in \{1, \ldots, K\}$. In other words, if there is at least one task executing between $\tau_{init,i}$ and τ_i, which can access a common memory bank as `Rx`, this (these) task(s) can be delayed by the higher priority NoC Rx data transfer by up to $\delta \cdot T_{acc}$. This is accounted for by increasing the $CWCRT_{p,k}(f_{prec}, \ell)$ accordingly.

Case 2 If $f_{prec} \neq f_{succ}$, then for each sub-frame k' from the k-th sub-frame

of f_{prec} up to and including the k-th sub-frame of f_{succ}: if there are tasks on p other than $\tau_{init,i}$, τ_i, which are interfering with Rx, then $CWCRT_{p,k'}(f', \ell)$ for the including FTTS frame f': $f_{prec} \leq f' \leq f_{succ}$ is increased by $\delta \cdot T_{acc}$, for all $\ell \in \{1, \ldots, K\}$. The intuition is similar as in the previous case. If any sub-frame between the one where $\tau_{init,i}$ is scheduled and the one where τ_i is scheduled includes tasks that are accessing a common bank as Rx, then this (these) task(s) can be delayed by the higher priority NoC Rx data transfer by up to $\delta \cdot T_{acc}$. Note, however, that in every frame f': $f_{prec} \leq f' \leq f_{succ}$, such an increase to $CWCRT_{p,k'}(f', \ell)$ happens only once, for the first sub-frame k' that includes interfering tasks with Rx. This is done for tighter analysis[1].

Example 3.5. *As an illustration of the per-core WCRT updates of the third step above, consider again the FTTS schedule of Figure 3.5. Suppose that τ_4 initiates a remote data transfer for τ_5, which reads the fetched data. Both tasks have criticality level $\chi_4 = 1$. For the first instance of the dependent tasks, $f_{prec} = f_1$ (frame where τ_4 is scheduled), $f_{succ} = f_2$ (frame where τ_5 is scheduled) and $k = 2$ (corresponding sub-frame within the above frames). Since $f_{prec} \neq f_{succ}$, for the per-core WCRT updates we have to consider all FTTS sub-frames starting from the 2nd sub-frame of f_1 up to and including the 2nd subframe of f_2. In this case, there are three such sub-frames to be considered. We assume that the memory accessing behavior of tasks τ_1 to τ_5 and the DMA transfer Rx are modelled by the memory interference graph of Figure 3.8. Tasks τ_6 and τ_7 of the FTTS schedule are not modelled in this graph because they perform no accesses to the shared memory. Given these assumptions, it follows that $CWCRT_{p_1,2}(f_1, \ell)$ and $CWCRT_{p_2,2}(f_1, \ell)$ for the 2nd sub-frame of f_1 and $\ell = \{1, 2\}$ remain unchanged. This is because on core p_1 no other task than τ_4 is scheduled, so the DMA transfer cannot cause any delay in this sub-frame on this core. Also on core p_2, the scheduled task τ_6 is not interfering with Rx, so the DMA transfer cannot delay its execution. In contrast, $CWCRT_{p_1,1}(f_2, \ell)$ and $CWCRT_{p_2,1}(f_2, \ell)$ for the 1st sub-frame of f_2 and $\ell = \{1, 2\}$ are increased by $\delta \cdot T_{acc}$. This is because the scheduled tasks τ_3 (on core p_1) and τ_2 (on core p_2) are interfering with Rx (accessing the same memory bank $bank_2$). Finally, $CWCRT_{p_1,2}(f_2, \ell)$, $CWCRT_{p_2,2}(f_2, \ell)$ for the 2nd sub-frame of f_2 remain unchanged, since no task other than τ_5 is scheduled on p_1 and the unique task on p_2, τ_6, is not interfering with Rx.*

After the discussed updates are performed, the computation of function *barriers* follows from Eq. (3.6) for the updated $CWCRT_{p,k}(f, \ell)$ values.

[1]Given the definition of δ, Rx cannot perform more than δ high-priority memory accesses within one single frame. Therefore, it is too pessimistic to increase $CWCRT_{p,k'}(f', \ell)$ for several sub-frames k' of the same frame f'. This would lead to a potential increase of $\sum_{k=1}^{K} barriers(f', \ell)_k$ by multiples of $\delta \cdot T_{acc}$.

3.5.1.3 Tighter Response Time Analysis

In Section 3.5.1.1 and 3.5.1.2, we derived closed-form expressions for the worst-case sub-frame lengths of the FTTS schedule (function *barriers*). Although several sources of pessimism were avoided, there may be still cases where the computed bounds are not tight. For instance, if the memory accesses of the tasks follow certain patterns (dedicated access phases, non-overlapping in time) such that even if two tasks are executed in parallel, they cannot interfere on the memory path, then the given bounds do not reflect this knowledge. In such cases, more accurate response time analysis can be provided by the method presented in Chapter 4, which uses a state-based model of the system with timed automata [AD94] and model checking to derive the task WCRTs. The system model in Chapter 4 specifies shared-memory multicores with equal-priority requesters, however it can be easily extended to model also the incoming traffic from the NoC, as computed in Section 3.5.2.

Since a model checker explores exhaustively all feasible resource interference scenarios, the above method can have a high complexity. Therefore, during design optimization (Section 3.6), where we need to compute function *barriers* for often thousands of potential FTTS schedules, it is preferred to use the WCRT bounds as derived earlier. We can then apply the method of Chapter 4 to the optimized FTTS solution to refine the computation of *barriers*. Also, if no admissible FTTS schedule can be found during optimization, the same method can be applied to the best encountered solutions, as the more accurate computation of *barriers* may reveal admissible schedules.

3.5.2 Bounding Delay for Data Transfers over NoC

This section shows how to characterize the incoming NoC traffic at the shared cluster memory. Based on the incoming traffic model, we compute upper bounds on (i) the delay for transferring a given amount of packets over a NoC and (ii) the number of accesses that the NoC Rx interface performs to the shared cluster memory in a given time interval. The first result (Eq. (3.12)) is used for defining the minimum distance constraint between a task initiating a remote data transfer, $\tau_{init,i}$, and a task using the fetched data, τ_i, in the remote fetch protocol of Section 3.3.2. The second result (Eq. (3.13)) is used as a parameter of the memory interference graph, representing the maximal interference from the NoC Rx interface, as discussed in Section 3.5.1.2.

We consider an explicitly routed NoC with wormhole switching and assume that each traffic stream uses a dedicated predetermined virtual channel throughout the NoC which is in line with previous approaches, see [ZSO+13]. Each NoC node is a router and also a flow source/sink. Routers contain only FIFO queues with one set of queues per outgoing link, with round-robin arbitration. Routers are work-conserving, i.e., not

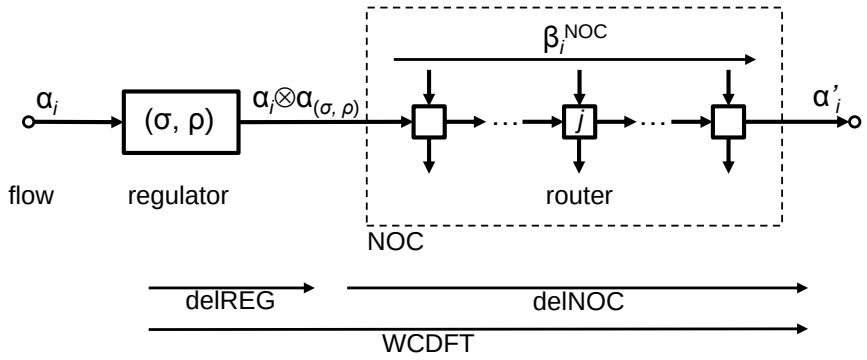


Figure 3.9: Modelling the packet flow i through a (σ, ρ)-regulator and a sequence of routers j.

idling if data are ready to be transmitted.

All flows are (σ, ρ) regulated at the sources [LMJ+09]. The parameters of the regulators are selected such that performance guarantees are provided for all flows and no FIFO queue in a router can overflow. Therefore, stalls due to backpressure flow control are not present. However, we assume that stalls due to switch contention are present, i.e., when packets from different input ports or virtual channels compete for the same output port. Such assumptions simplify the presented NoC analysis. However, if necessary, backpressure stalls can be easily integrated by using existing results [Cha00, TS09, ZSO+13].

Since we consider hard real-time guarantees on the NoC, we have to show that each network packet in a certain flow is delivered to its destination within a fixed deadline. For the analysis, we use the theory of Network and Real-time calculus [Cru91, LBT01, TCN00]. It is a theory of deterministic queuing systems for communication networks and scheduling of real-time systems. Network calculus has been applied in recent works and its effectiveness has been validated for the analysis of NoC-based systems, see [QLD, QLD10, ZSO+13]. The theory analyzes the flow of packet streams through a network of processing and communication resources in order to compute worst-case backlogs, end-to-end delays and throughput. The overall modelling approach is shown in Figure 3.9.

A General Packet Stream Model. Packet streams are abstracted by the function $\alpha(\Delta)$, i.e., an upper arrival curve which provides an upper bound on the number of packets in *any* time interval of length $\Delta \in \mathbb{R}$, where $\alpha(\Delta) = 0$ for all $\Delta \leq 0$. Arrival curves substantially generalize conventional stream models such as sporadic, periodic or periodic with jitter. Note that a packet here is defined as a fixed-length basic unit of network traffic. Variable-length packets can be viewed as a sequence of fixed-length packets.

A General Resource Model. The availability of processing or communication resources is described by the function $\beta(\Delta)$, a lower

service curve which provides a lower bound on the available service in *any* time interval of length $\Delta \in \mathbb{R}$, where $\beta(\Delta) = 0$ for all $\Delta \leq 0$. The service is expressed in an appropriate workload unit compatible to that of the arrival curve, e.g., packets.

Packet stalls at routers can happen when packets from different input ports or virtual channels compete for the same output port. We assume a round-robin arbiter for every router j where each flow i is given a fixed slot of size s_i, e.g., proportional to the maximum packet size for this flow. The packet size is defined as the number of (fixed-length) packets of a flow. The accumulated sizes of all flows going through a router j can be expressed as $s^j = \sum_{i \text{ flows through } j} s_i$. If the lower service curve that the router can provide is denoted as β^j, then the lower service curve under round-robin arbitration for a flow with slot size s_i is [ZSO+13]:

$$\beta_i^j(\Delta) = \frac{s_i}{s^j}\beta^j(\Delta - (s^j - s_i)) \ , \tag{3.7}$$

which expresses the fact that in the worst-case a packet may always have to wait for $s^j - s_i$ time units before its slot becomes available.

A Resource Model for a Network. When a packet flow traverses a system of multiple interconnected components, one needs to consider the service curve provided by the system as a whole, i.e., the system service curve is a concatenation of the individual service curves [LBT01, TS09]. For example, the concatenation of two routers 1 and 2 with lower service curves β_i^1 and β_i^2 for a flow i can be obtained as:

$$\beta_i^{1,2}(\Delta) = \beta_i^1 \otimes \beta_i^2(\Delta) \ ,$$

where $\otimes$ is the min-plus algebra convolution operator that is defined as:

$$(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \ .$$

As a result, the cumulative service β_i^{NOC} for a flow i is the convolution of all individual router service curves on the path of the flow through the network:

$$\beta_i^{\text{NOC}}(\Delta) = \Big(\bigotimes_{i \text{ flows through } j} \beta_i^j \Big)(\Delta) \ . \tag{3.8}$$

Flow Regulator. A flow regulator with a (σ, ρ) shaping curve delays packets of an input flow such that the output flow has the upper arrival curve $\alpha_{(\sigma,\rho)}(\Delta) = (\rho \cdot \Delta + \sigma)$ for all $\Delta > 0$, independent of the timing characteristics of the input flow, and it outputs packets as soon as possible without violating the upper bound $\alpha_{(\sigma,\rho)}$.

Delay Bounds. A packet stream constrained by an upper arrival curve α_i is first regulated by a (σ, ρ) regulator and then it traverses a network that offers a cumulative lower service curve β_i^{NOC} of the routers on the

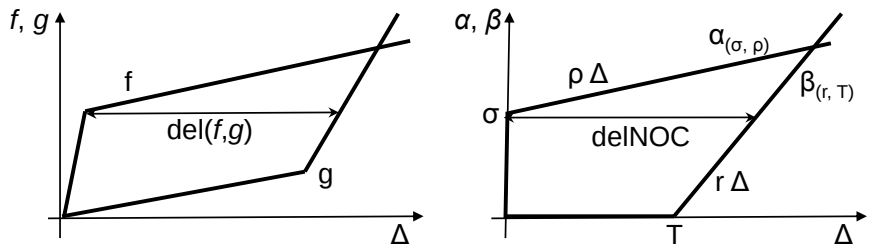


Figure 3.10: Delay bound defined as the maximum horizontal distance illustrated for an arrival curve of a (σ, ρ) regulated flow and a single router providing a rate-latency service curve $\beta^l_{r,T}$.

packet path. It is well known from the network and real-time calculus that the maximum packet delay is related to the maximal horizontal distance between functions (see Figure 3.10) which is defined as:

$$\mathrm{del}(f, g) = \sup_{\lambda \geq 0} \{\inf\{\tau \geq 0 : \ f(\lambda) \leq g(\lambda + \tau)\}\} .$$

Now, the worst-case delay at the regulator delREG experienced by a packet from a flow i constrained by an arrival curve α_i and regulated by a (σ, ρ) flow regulator can be computed as follows [WMT06]:

$$\mathrm{delREG} = \mathrm{del}(\alpha_i, \alpha_{(\sigma,\rho)}) . \tag{3.9}$$

The output of the regulator is constrained by $(\alpha_i \otimes \alpha_{(\sigma,\rho)})(\Delta)$ and therefore, the worst-case packet delay delNOC for flow i within the NoC that has a cumulative lower service β_i^{NOC} can be determined as:

$$\mathrm{delNOC} = \mathrm{del}(\alpha_i \otimes \alpha_{(\sigma,\rho)}, \beta_i^{\mathrm{NOC}}) . \tag{3.10}$$

Example 3.6. *An example of worst-case delay computation is shown in Figure 3.10. We consider a single router which serves a single flow constrained by an upper arrival curve $\alpha_{(\sigma,\rho)}$. The NoC consists of a single router that provides to the flow a lower rate-latency service curve $\beta_{r,T}(\Delta) = r(\Delta - T)$ for $\Delta > T$, and $\beta_{(r,T)}(\Delta) = 0$ otherwise. The arrival curve implies that the source can send at most σ packets at once, but not more than ρ packets per cycle in the long run, while the service curve implies a pipeline delay of T for a packet to traverse the router and an average service rate of r packets per cycle. As shown in Figure 3.10, the worst-case delay bound corresponds to the maximum horizontal distance between the upper output arrival curve of the flow regulator and the lower service curve of the NoC.*

Output Flow Bounds. When a packet stream constrained by arrival curve α_i is regulated by a (σ, ρ) regulator and traverses a network that offers a cumulative lower service curve β_i^{NOC}, the processed output flow is bounded by α_i' computed as follows [WTVL06]:

$$\alpha_i'(\Delta) = ((\alpha_i \otimes \alpha_{(\sigma,\rho)}) \oslash \beta_i^{\mathrm{NOC}})(\Delta) , \tag{3.11}$$

where $\oslash$ is the min-plus algebra deconvolution operator that is defined as:

$$(f \oslash g)(\Delta) = \sup_{\lambda \geq 0}\{f(\Delta + \lambda) - g(\lambda)\} .$$

Data Transfer Delay. Finally, we compute an upper bound on the total delay for transferring a buffer of data using multiple packets, e.g., the maximum delay for transferring 4KB of data from external memory over a NoC. The availability of data can be modelled as an upper arrival curve which has the form of a step function: $\alpha_i(\Delta) = B$ for $\Delta > 0$, where B is the total amount of data measured as the number of packets. Then an upper bound on the total delay can be computed as a sum of the delays computed by Eq. (3.9) and (3.10), where the first equation bounds the delay experienced by all of the data at the regulator, i.e., the delay for the regulator to transmit all of the data, and the second equation bounds the delay for transferring the last packet of the data through the NoC.

In other words, a bound on the total delay for transferring a buffer of B packets regulated by a (σ, ρ) regulator over a NoC that provides a cumulative lower service curve of β_i^{NOC} can be computed as:

$$\text{WCDFT} = \text{delREG} + \text{delNOC} , \tag{3.12}$$

where $\alpha_i(\Delta) = B$ for $\Delta > 0$, delREG and delNOC are defined in Eq. (3.9) and (3.10), respectively, and WCDFT denotes the worst-case data fetch time, as originally defined in the remote fetch protocol of Section 3.3.2.

Note that the above model is valid under the assumption that during the data transfer, the regulator does not stall because there are no packets available for transmission. For instance, in the case of external memory data fetch, the memory controller should be able to insert packets fast enough into the buffer of the regulator. Moreover, the regulator should not experience stalls due to backpressure.

Bound on NoC Rx Memory Accesses within an FTTS Frame. For representing interference from the NoC Rx interface within a compute cluster, in Section 3.5.1.2 we introduced the value δ, which bounds the number of memory accesses that the NoC Rx can perform within a time frame f. For the time interval of the frame f, denoted as L_f, δ can be not greater than $\alpha'(L_f)$ (Eq. (3.11)), but also not greater than the total number of packets in the transmitted buffer, B. Therefore,

$$\delta = \min\left\{\alpha'(L_f), B\right\} . \tag{3.13}$$

The D-NoC in the Kalray Platform. The models and methods described above are compatible with the many-core Kalray MPPA-256 platform. In the following, we give a short summary of flow regulation for the MPPA NoC, which is based on (σ, ρ) regulators. Precisely, in the MPPA-256 processor, each connection is regulated at the source node by a packet

shaper and a traffic limiter in tandem. This regulator can be configured via two parameters, both defined in units of 32-bit flits: (i) a *window length* (T_w), which is set globally for the NoC node and (ii) the *bandwidth quota* (N_{max}), which is set separately for each regulator. At each cycle, the regulator compares the length of a packet scheduled for injection plus the number of flits sent within the previous T_w cycles to N_{max}. If not greater, the packet is injected at the rate of one flit per cycle.

The (σ, ρ) parameters can be set at the source node through T_w and N_{max} (all measured in units of 32-bit flits, including header flits). We link these parameters with the (σ, ρ) model by observing that $\rho = N_{max}/T_w$. This corresponds to the fact that no regulator may let more than N_{max} flits pass over any duration T_w. On the other hand, the regulator is allowed to emit continuously until having sent N_{max} flits within exactly N_{max} cycles. This defines a point on the $\rho + \sigma$ linear time function and by regression, the value of the function at time $t = 0$ (corresponding to σ) is found to be $\sigma = N_{max}(1 - N_{max}/T_w)$. Note that $\sigma \geq 0$. For a more detailed presentation of the MPPA NoC flow regulation, the interested readers are referred to [dDvAPL14].

The MPPA NoC routers multiplex flows originating from different directions. Each originating direction has its own FIFO queue at the output interface, so flows interfere on a node only if they share a link to the next node. This interface performs a round-robin arbitration at the packet granularity between the FIFOs that contain data to send on a link. The NoC routers have thus been designed for simplicity while inducing a minimal amount of perturbations on (σ, ρ) flows. An additional benefit of this router design is that eliminating backpressure from every single queue through (σ, ρ) flow regulation at the source effectively prevents deadlocks. Therefore, it is not necessary to resort to specific routing techniques such as turn-models. Selecting (σ, ρ) parameters for all flows is treated as an optimization step at design time, which however is outside the scope of this chapter.

3.6 Design Optimization

Section 3.4 presented the runtime behavior of the FTTS scheduler and Section 3.5 the response time analysis for tasks that are scheduled under FTTS and can experience blocking delays on the shared memory path of a cluster by concurrently executing tasks in the cluster or by the incoming traffic from the NoC. In both sections, we assumed a given FTTS schedule, with known task mapping to cores and data mapping to memory banks. In this section, we discuss the problem of actually finding an FTTS schedule while optimizing resource utilization in our system.

The problem can be formulated as follows. **Given** (i) a periodic mixed-criticality task set τ with dependency graph $\mathcal{D}ep$, (ii) a cluster consisting

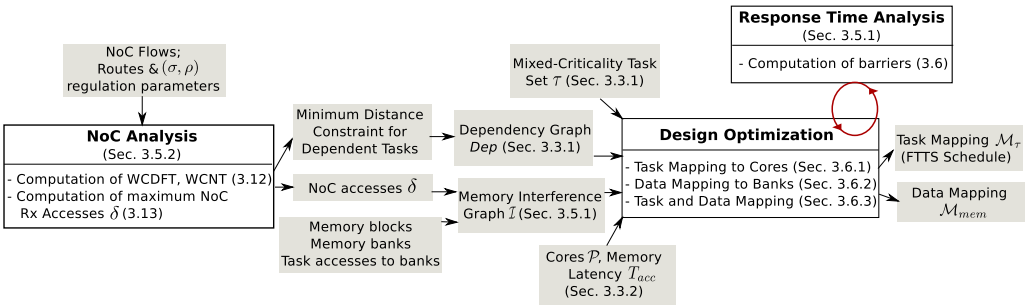


Figure 3.11: FTTS Design Flow. The white blocks represent the main tasks of the design flow, resp. the main contributions of the chapter. The grey blocks represent inputs/outputs to/from the design tasks.

of processing cores $\mathcal{P}$ with access to a banked memory, (iii) the memory interference graph $\mathcal{I}$ with undefined edge set $\mathcal{E}_2$, (iv) the memory access latency T_{acc}; **Determine** the mapping $\mathcal{M}_\tau : \tau \rightarrow \mathcal{P}$ of tasks to processing cores and the mapping $\mathcal{M}_{mem} : BL \rightarrow B$ ($\mathcal{E}_2$ of $\mathcal{I}$) of memory blocks to banks **such that**:

- all tasks meet their mixed-criticality real-time requirements at all levels of assurance,
- the worst-case sub-frame lengths are *minimized*, which leads to a balanced workload distribution across the cores and minimal schedulability loss due to the IS constraint,
- the minimum distance constraints of the dependency graph $\mathcal{D}ep$ are not violated, and
- the memory bank capacities are not surpassed.

The mapping $\mathcal{M}_\tau$ defines both the spatial partitioning of tasks among the cores in $\mathcal{P}$ as well as the timing partitioning into frames and the execution order on each individual core. These three aspects (spatial, timing partitioning, relative execution order) determine fully an FTTS schedule for task set τ.

For each of the two considered optimization problems ($\mathcal{M}_\tau$ in Section 3.6.1 and $\mathcal{M}_{mem}$ in Section 3.6.2), we assume an existing solution to the other one. Finally, we show how to solve both optimization problems in an integrated manner (Section 3.6.3). To facilitate reading, please refer to Figure 3.11, which depicts the inputs and outputs of the optimization procedure as well as the flow of analyses (NoC analysis, response time analysis) and information (task set, platform model, etc.) which enable us to determine some of the inputs, e.g., the memory interference graph, and to evaluate the visited solutions during optimization.

3.6.1 Task Mapping $\mathcal{M}_\tau$ Optimization

The problem of optimal task mapping on multiple cores is known to be NP-hard, resembling the combinatorial bin-packing problem. To reduce the complexity of finding an optimal mapping $\mathcal{M}_\tau$, we propose a heuristic method which is based on simulated annealing (SA) [KGV83]. In its general form, the SA algorithm seeks the global minimum of a given cost function in a state space. It begins with an arbitrary solution (state) and it considers a series of random transitions based on a neighbourhood function. At each step, if the neighbouring state S' is of lower cost than the current state S, SA accepts the transition to S'. Else, SA accepts the transition with probability $e^{-(Cost(S')-Cost(S))/T}$, where T is a positive constant, commonly known as temperature. SA is only one of the methods that can be applied for the design optimization. If the optimization problem itself was the focus, one could consider also e.g., constraint solvers or other non black box heuristics. This is, however, outside the scope of this chapter.

Our optimization approach starts by generating a random task mapping solution, resp. FTTS schedule for the given task set τ. Specifically, it selects the FTTS cycle as the hyper-period H of tasks in τ and also, the FTTS frame lengths depending on the task periods (the greatest common divisor of the periods is used for all frames unless otherwise specified by the system designer). For every task $\tau_i \in \tau$, it selects arbitrarily a core on which the task will be mapped. Then, it computes the number of jobs that are released by task τ_i within a hyper-period H ($\frac{H}{T_i}$) and the range of FTTS frames in which each job can be scheduled, such that it is executed between its release time and absolute deadline. For every job of τ_i, it selects arbitrarily an FTTS frame from the allowed range. This procedure is repeated for all tasks in τ. The constraints that must be respected during the generation of the initial FTTS schedule are:

- All jobs of the same task are scheduled on the same core.
- For every dependency $\tau_i \rightarrow \tau_j$ in the dependency graph $\mathcal{D}ep$, the jobs of the two tasks are scheduled on the same core, with a job of τ_j being scheduled in the same or a later frame than the corresponding job of τ_i within their common period. If they are scheduled in the same frame, the job of τ_j must succeed that of τ_i. The sum of the best-case execution times of the jobs that are scheduled in between τ_i and τ_j and the lengths of the intermediate frame(s) (if any exist between the two jobs) must be no lower than their minimum distance constraint, i.e., the weight of the corresponding edge in $\mathcal{D}ep$.

Note that this is the procedure that a system designer would follow to generate a random FTTS schedule for a given task set τ. It provides no guarantee on the schedule admissibility. If at this initial step no solution

can be found to satisfy the above criteria, the search is aborted, i.e., τ is considered non-schedulable on $\mathcal{P}$.

Once an initial mapping solution is determined, the optimizer applies simulated annealing [KGV83] to explore the design space for task mapping. Particularly, new solutions are found by randomly selecting a task $\tau_i \in \tau$ and applying one of two possible variations with given probabilities: (i) re-mapping all jobs of τ_i (and its dependent tasks in $\mathcal{D}ep$) to a different core or (ii) re-allocating one randomly selected job of τ_i to a different FTTS sub-frame or to a different position within the same sub-frame. Design space exploration is restricted to solutions that satisfy the dependency constraints in $\mathcal{D}ep$. The exploration terminates when it converges to a solution or a computational budget is exhausted.

A task mapping solution is considered optimal if all jobs meet their deadlines at all levels of assurance, i.e., the schedule is admissible, and the worst-case sub-frame lengths are minimized, implying a balanced workload distribution. Based on these requirements, we define the cost function of the optimization problem as:

$$Cost(S) = \begin{cases} c_1 = \max_{f \in \mathcal{F}} \{\max_{\ell \in \{1,\dots,K\}} late(f,\ell)\} & \text{if } c_1 > 0 \\ c_2 = \|barriers\|_3 & \text{if } c_1 \leq 0, \end{cases} \tag{3.14}$$

where $late(f,\ell)$ is the difference between the worst-case completion time of the last sub-frame of f and the length of f at level of assurance ℓ:

$$late(f,\ell) = \sum_{i=1}^{K} barriers(f,\ell)_i - L_f \ . \tag{3.15}$$

If $late(f,\ell) > 0$, the tasks in f cannot complete execution by the end of the frame for their ℓ-level execution profiles. Therefore, with this cost function, we initially guide design space exploration towards finding an admissible solution. When such a solution is found, cost c_1 becomes negative or 0. Then, c_2, i.e., the 3-norm of all sub-frame lengths, $\forall f \in \mathcal{F}, \forall \ell \in \{1,\dots,K\}$, is used to minimize the worst-case lengths of all sub-frames. The 3-norm of a vector x with n elements (here, positive real numbers) is defined as $\|x\|_3 := \left(\sum_{i=1}^{n} |x_i|^3\right)^{1/3}$. We selected the particular value to map (represent) the vector with the *barriers* values for all $f \in \mathcal{F}$ and $\ell \in \{1,\dots,K\}$, as we empirically found this to be the best among other considered norms, such as the average, the maximum, the sum or the Euclidean norm. Namely, the selected norm provides a trade-off between reducing the worst-case sub-frame lengths (to ensure schedulability) and enabling progress in the optimization via improving the average-case lengths. However, alternative norms, such as the ones mentioned above can be also used in our cost function (3.14). Note that during exploration, the *barriers* function is computed for each visited solution, as discussed in Section 3.5. For the WCRT analysis, the memory mapping $\mathcal{M}_{mem}$ is assumed to be known.

The task mapping optimization method can be easily extended to account for fixed task preemption points, mapping constraints, solution ranking, etc. We discuss the case of fixed preemption points below.

Extension: Preemption Points. In certain cases, some sort of preemption is indispensable for schedulability, esp. when one considers task sets with one or more computationally intensive tasks, which may not "fit" in any frame of a global FTTS schedule. Enabling a preemptive scheduling strategy, where each task can start executing in one sub-frame and continue over several frames (in the respective sub-frames of its criticality level) would not be efficient because:

- The computationally intensive tasks would be allowed to run up to the end of each frame in which they are scheduled, thus preventing any other tasks with the same or lower criticality level from being executed. This behavior would affect not only tasks on the same core, but also on the remaining cores due to the Isolation Scheduling constraint.
- The fact that these tasks could be preempted at any possible point of execution makes accurate interference analysis impossible, since the execution profiles of the tasks (including memory accesses) cannot be extracted for any possible partial execution.

To avoid these problems, we use the concept of *fixed preemption points*. A task may have a certain amount of well-defined preemption points, so that execution profiles for the corresponding partial executions can be extracted. We specify each "preemptable" task by a list of alternative executions, e.g., one with no preemption, one with 1 preemption point, etc. In each case, the partial executions are defined as chains of d ependent tasks with the same period, e.g., a chain with one task or 2 dependent tasks, respectively. The alternatives are given as input to our optimization algorithm. An admissible FTTS schedule must eventually include one of the alternative executions of the related tasks. Note that this extension introduces a new possible variation of a solution during design space exploration. Namely, a new solution can be found by randomly selecting a task chain and substituting it for one of its alternative executions.

3.6.2 Memory Mapping $\mathcal{M}_{mem}$ Optimization

The goal of memory mapping optimization is to determine a static allocation of the task instructions, private data and communication buffers (memory blocks) BL to banks B of the shared memory ($\mathcal{E}_2$ of memory interference graph $\mathcal{I}$), so that the timing interferences of tasks when accessing the memory are minimized. For a given task mapping $\mathcal{M}_\tau$, this leads to a minimization of the worst-case sub-frame lengths (*barriers*). A constraint of the optimization problem is that the total size of the

allocated memory blocks in a bank must not surpass the bank capacity. This constraint holds e.g., for the memory mapping in Figure 3.6.

For this problem, we adopt a heuristic method based on simulated annealing, similar to the task mapping optimization. The method is described in Algorithm 5. It receives as inputs an initial temperature T_0, a temperature decreasing factor $a \in (0,1)$, the maximum number of consecutive variations with no cost improvement that can be checked for a particular temperature $Fail_{max}$, a stopping criterion in terms of the final temperature T_{final}, and a stopping criterion in terms of search time (computational budget) $time_{max}$. It returns the best encountered solution(s) in the given time.

The algorithm starts with an arbitrary initial solution S, satisfying the bank capacity constraints. If function GenerateInitialSolution() can provide no such solution, exploration is aborted (lines 1–4). Otherwise, design space exploration is performed by examining random variations of the memory mapping. Particularly, function Variate() selects randomly a memory block and remaps it to a different memory bank such that no bank capacity constraint is violated (line 10). The new solution S' is accepted if $e^{-(Cost(S')-Cost(S))/T}$ is no lesser than a randomly selected real value in (0,1) (lines 11–13). The cost of S' is, also, compared to the minimum observed cost, $Cost_{min}$. If it is lower than $Cost_{min}$, the new solution and its cost are stored even if transition to S' was not admitted (lines 15–19). The temperature T of the simulated annealing procedure is reduced geometrically with factor a. Reduction takes place every time a sequence of $Fail_{max}$ consecutive solutions are checked, none of which improves $Cost_{min}$. After temperature reduction, exploration continues from the so-far best found solution (S_{cur_best}) (lines 22–26). Design space exploration terminates when the lowest temperature T_{final} is reached or the computational budget $time_{max}$ is exhausted.

Memory mapping affects the WCRT of a task τ_i by defining which of the tasks that can be executed in parallel with it are also interfering with it. The less interfering tasks, the lower the delay τ_i experiences when accessing the shared memory. Therefore, to evaluate a memory mapping solution we select a cost function which reflects the increase in task WCRT due to interference on the shared memory banks. The cost function is based on the mutual delay matrix D, which was defined in Section 3.5.1.1, and has two alternative definitions.

One alternative to solve the optimization problem is to compute (part of) the *Pareto set* of memory mapping solutions with minimal interference between any two tasks with the same criticality level. The intuition behind this approach is that we try to minimize simultaneously all elements of the mutual delay matrix D, namely all blocking delays that a task can cause to any other task (with the same criticality). This problem can be seen as a multi-objective optimization problem with the n^2 elements of matrix D as individual cost functions. For this set of objectives, we compute

Algorithm 5: Modified Simulated Annealing for Memory Mapping $\mathcal{M}_{mem}$.

Input: T_0, a, $Fail_{max}$, T_{final}, $time_{max}$
Output: $\bar{S}_{best}$

```
1:  S ← GenerateInitialSolution()
2:  if S = null then
3:      return null
4:  end if
5:  S̄_best ← {S}, S_cur_best ← S, Cost_min ← Cost(S)
6:  T ← T_0
7:  FailCount ← 0
8:  time ← StartTimer()
9:  while time < time_max and T > T_final do
10:     S' ← Variate(S)
11:     if e^{-(Cost(S')-Cost(S))/T} ≥ Random(0,1) then
12:         S ← S'
13:     end if
14:     UpdateBestSolutions(S')
15:     if Cost(S') < Cost_min then
16:         S_cur_best ← S'
17:         Cost_min ← Cost(S')
18:         FailCount ← 0
19:     else
20:         FailCount ← FailCount + 1
21:     end if
22:     if FailCount = Fail_max then
23:         T ← a · T
24:         S ← S_cur_best
25:         FailCount ← 0
26:     end if
27: end while
```

the Pareto set of memory mapping solutions. Algorithm 5 maintains such a set $\bar{S}_{best}$ of non-dominated solutions. In particular, a newly visited mapping solution S' with matrix D' is inserted into set $\bar{S}_{best}$ if it has a lower value for at least one element of D' than the corresponding element of any solution in $\bar{S}_{best}$. If a solution $S \in \bar{S}_{best}$ is dominated by S', i.e., S' has lower or equal values for all elements of D, then S is removed from the set. This update is performed by function UpdateBestSolutions() (line 14).

The second alternative is to define the scalar cost function D_{avg} as the *average* over all elements of matrix D, i.e., the average delay that tasks with the same criticality level cause to each other when interfering on shared banks. Then we can seek the best solution in terms of D_{avg}. In this case, $\bar{S}_{best}$ contains only one solution characterized by the minimum

encountered D_{avg}.

3.6.3 Integrated Task and Memory Mapping Optimization

The problems of optimizing $\mathcal{M}_\tau$ and $\mathcal{M}_{mem}$ are inter-dependent. Namely, design space exploration for the optimization of the task mapping requires information on the memory mapping for computing function *barriers*. Similarly, matrix D, which defines the cost of a memory mapping solution, can be refined for a particular task mapping depending on the tasks that can be executed in parallel. In the following, we outline two alternative approaches towards an integrated optimization solution.

I. **Task mapping optimization for each memory mapping in Pareto set.** As discussed previously, one can compute using Algorithm 5 part of the Pareto set $\bar{S}_{best}$ of memory mapping solutions that minimize the interference between any two tasks of the same criticality level. These solutions consider that all tasks with the same criticality level are potentially executed in parallel (*worst-case task mapping*). The next step is to solve the task mapping optimization problem for each memory mapping in the set $\bar{S}_{best}$. Finally, the combination of solutions which minimizes $||barriers||_3$ is selected.

II. **Iterative task and memory mapping optimization.** Since the complexity of computing the Pareto set solutions for the memory mapping optimization problem can be prohibitive, one can select an iterative solution to the two problems. Then, for each visited solution during design space exploration for $\mathcal{M}_\tau$, a memory mapping optimization is also performed to find the solution with minimized cost $||barriers||_3$. Here, we use the cost function D_{avg}.

It cannot be said that one method clearly outperforms the other in terms of efficiency. Empirical evaluation has shown that depending on the sizes of the search spaces of the task mapping and memory mapping optimization problems, one algorithm can perform faster than the other.

3.7 A Case Study: Flight Management System

To evaluate the proposed design optimization approaches, we use an industrial implementation of a flight management system [DFG+14]. This application was the central use-case of the European Certainty project [cer]. The purpose for the evaluation is first, to show applicability of our optimization methods and demonstrate the results of the response time analysis for the optimized task and memory mapping solution. Second, we investigate the effect of various platform parameters, such as the memory access latency and the number of memory banks, or

design choices, such as the selection of routes and (σ,ρ) parameters for the NoC flows, on the schedulability of the application under FTTS. The optimization framework has been implemented in Java and the evaluation was performed on a laptop with a 4-core Intel i7 CPU at 2.7 GHz and 8 GB of RAM.

Flight Management System. The flight management system (FMS) from the avionics domain is responsible for functionalities such as the localization of an aircraft based on periodically acquired sensor data, the computation of the flightplan that guides the auto-pilot, the detection of the nearest airport, etc. We look into a subset of the application, consisting of 14 periodic tasks for sensor reading, localization and computation of the nearest airport. Seven are characterized by safety level DAL-B (we map it to criticality level 2, i.e., high) and seven by safely level DAL-C (we map it to criticality level 1, i.e., low) based on the DO-178C standard for certification of airborne systems [RTC12]. The periods of the tasks vary among 200 ms, 1 sec and 5 sec, as shown in Table 3.4. Based on the periods, we select the cycle and the frame length of the FTTS schedule as $H = 5$ sec and $L_f = 200$ ms for all $f \in \mathcal{F}$, respectively. The worst-case execution times of the tasks were derived through measurements on a real system or for few tasks for which the code was not available (e.g., $\tau_{init,13}$) based on conservative estimations. A discussion on how the worst-case execution time parameters of the FMS tasks can be derived can be found in the technical report [Cer14a]. For the level-2 profiles $C_i(2)$ of tasks τ_i with $\chi_i = 2$, we augment the worst observed execution times by a factor of 5. Similarly, for the memory accesses, we consider conservative bounds based on the known memory footprints for the tasks and derive the $C_i(2)$ parameters by multiplying these bounds by 5. Factor 5 is selected arbitrarily to augment the worst-case task parameters and thus, increase the safety margins. Such augmentation of the worst-case parameters seems a common industrial practice for safety-critical applications. The best-case execution time and access parameters are taken equal to 0 due to lack of more accurate information. Last, the degraded profiles $C_{i,deg}$ of tasks τ_i with $\chi_i = 1$ correspond to no execution, i.e., $C_{i,deg} = (0, 0, 0, 0)$. The task periods, criticality levels, level-1 and level-2 worst-case execution times and memory accesses as well as the memory blocks that each task accesses and the maximum number of accesses at the task's own criticality level are shown in Table 3.4.

To model the memory accessing behavior of the tasks according to Section 3.5.1, we define a memory interference graph $\mathcal{I}$ with the following memory blocks: one block per task with size equal to the size of its data as measured on the deployed system and one block per communication buffer with known size, too. This yields in total 27 memory blocks.

The FMS requires access to a navigation database with a memory footprint of several tens of MB. Particularly task τ_{13}, which is responsible

Purpose	Task τ_i	CL χ_i	Period T_i (ms)	Level-1 (2) e_i^{max} (ms)	Level-1 (2) μ_i^{max}	Max. Accesses to Mem. Blocks
Sensor data acquisition	τ_1	2	200	11 (55)	213 (1065)	b_1 (100), b_2 (425), b_4 (70), b_6 (190) b_8 (190), b_{10} (90)
	τ_2	1	200	20 (0)	117 (0)	b_3 (10), b_4 (107)
	τ_3	1	200	18 (0)	129 (0)	b_5 (10), b_6 (119)
	τ_4	1	200	18 (0)	129 (0)	b_7 (10), b_8 (119)
	τ_5	1	200	20 (0)	129 (0)	b_9 (10), b_{10} (119)
Localization	τ_6	2	200	7 (35)	145 (725)	b_2 (425), b_{11} (100), b_{12} (100), b_{13} (90) b_4 (10)
	τ_7	2	1000	6 (30)	56 (280)	b_{13} (90), b_{14} (100) b_{15} (90)
	τ_8	2	5000	6 (30)	57 (285)	b_{15} (17), b_{16} (100), b_{17} (90), b_{19} (78)
	τ_9	2	1000	6 (30)	57 (285)	b_{17} (90), b_{18} (100), b_{21} (78), b_{22} (17)
	τ_{10}	1	200	20 (0)	130 (0)	b_{12} (120), b_{24} (10)
	τ_{11}	1	1000	20 (0)	113 (0)	b_{19} (103), b_{25} (10)
	τ_{12}	1	200	20 (0)	113 (0)	b_{21} (103), b_{26} (10)
Nearest Airport	τ_{13}	2	1000	48 (192)	1384 (6920)	b_{17} (90), b_{20} (100), b_{23} (1610), b_{27} (5120)
	$\tau_{init,13}$	2	1000	2 (10)	18 (90)	b_{17} (90), Triggers Rx to b_{27} (403)

Table 3.4: Flight Management System specification.

for the computation of the nearest airport, needs read-access to certain entries (up to 4 KB of data). In the following, we assume that the database is maintained in an external DDR memory and the required data are fetched to the local memory of a cluster, where the FMS application is executed. The data transfer is initiated by the preceding task $\tau_{init,13}$, which has the same criticality level and period as τ_{13}. The memory block corresponding to the database data is bl_{27}. We add a high-priority task $\texttt{Rx}_{13}$ to $\mathcal{I}$ to indicate the remote data transfer. $\texttt{Rx}_{13}$ is connected to the memory block b_{27} via an edge with weight δ.

Note that the FMS contains no task dependencies other than that between the task requesting the database entries for the computation of the nearest airport, $\tau_{init,13}$, and the task that performs the computation, τ_{13}. In the following, we show how to compute the minimum distance constraint between $\tau_{init,13}$ and τ_{13} in the dependency graph $\mathcal{D}ep$ as well as the weight δ for the edge from the node $\texttt{Rx}_{13}$ to bl_{27} in the memory interference graph $\mathcal{I}$.

NoC Flow Routing and Regulation. Task τ_{13} reads upon each activation 4 KB of data from the database. The data are transferred from the remote cluster with access to the DDR over the D-NoC in packets of 4 Bytes. Namely, a transfer of 1024 packets must be executed between any two successive executions of τ_{13}. We assume that this flow is (σ, ρ) regulated

at the I/O sub-system, with σ = 10 packets and ρ = 2000 packets/sec. The (σ, ρ) parameters are selected arbitrarily here, such that they are reasonable for the required amount of transferred data and they allow the transfer over the NoC to be completed within a period of task τ_{13}. The flow routing is fixed and passes through *two* D-NoC routers. On the first router, the flow can encounter interference from *one* more flow on the output link. Respectively, on the second router, the flow interferes with *three* more flows on the output direction. The clock frequency on the chip is 400 MHz. The routers forward the packets over the D-NoC links at a rate of 1 packet/cycle, equiv. 400,000,000 packets/sec. Given the above assumptions and by applying Eq. (3.12) and (3.13) of Section 3.5.2, we derive the worst-case data fetch time, WCDFT = 511.4 ms, and the maximum number of packets fetched during 200 ms (duration of an FTTS frame), δ = 403, respectively.

Based on the remote fetch protocol of Section 3.3.2, to specify the minimum distance constraint between task $\tau_{init,13}$ and τ_{13}, we need to know besides WCDFT, the worst-case notification time, WCNT, for the transfer of the notification from $\tau_{init,13}$ to the remote listener task in the I/O sub-system as well as the worst-case remote set-up time for the data transfer, WCRST. WCNT can be derived in a similar way as WCDFT. Assuming that $\tau_{init,13}$ sends only one packet (4 Bytes) to the remote cluster, following the exact same route as the flow from the I/O sub-system to the cluster, it follows from Eq. (3.12) that WCNT = 0.4 ms. Regarding WCRST, a conservative bound is assumed to be given, WCRST = 25 ms (here, arbitrary selection). Summarizing on the above results, the dependency between $\tau_{init,13}$ and τ_{13} in the dependency graph $\mathcal{D}ep$ is weighted with the minimum distance constraint: w =(0.4 + 25 + 511.4) ms = 536.8 ms.

Platform Parameters. For the deployment of the FMS, we consider a target platform resembling a cluster of the MPPA-256 platform. In particular, $\mathcal{P}$ includes 8 processing cores with shared access to 8 memory banks of 128 KB each. We consider round-robin arbitration on the memory arbiters with higher priority for the NoC Rx interface, according to the description in Section 3.3.2. Once a memory access is granted, the fixed memory latency is T_{acc} = 55 ns. The memory latency bound has been empirically estimated on the MPPA-256 platform using benchmark applications. Note, however, that it is not necessarily a safe bound for the MPPA-256 memory controller.

3.7.1 Design Optimization and Response Time Analysis

With the first experiment we intend to evaluate the applicability and efficiency of the optimization framework of Section 3.6 w.r.t. the deployment of the FMS on a compute cluster. The scheduling policy in the cluster is FTTS with a cycle of H = 5000 ms, consisting of 25 frames with length 200 ms each, based on the periods of our task

Frame	Core	Sub-frame 1	Sub-frame 2	Frame	Core	Sub-frame 1	Sub-frame 2
f_1 [0,200]	p_1	$\tau_{init,13}$	τ_{10}, τ_2, τ_3	f_2 [200,400]	p_1	-	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5		p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5
f_3 [400,600]	p_1	-	τ_{10}, τ_2, τ_3	f_4 [600,800]	p_1	τ_{13}	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5, τ_{11}		p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5
f_5 [800,1000]	p_1	τ_7, τ_9	τ_{10}, τ_2, τ_3	f_6 [1000,1200]	p_1	$\tau_{init,13}$	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_5, τ_{12}		p_2	τ_6, τ_1	τ_4, τ_5, τ_{12}
f_7 [1200,1400]	p_1	τ_7	τ_2, τ_{10}, τ_3	f_8 [1400,1600]	p_1	τ_9	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5		p_2	τ_6, τ_1	τ_4, τ_5, τ_{11}, τ_{12}
f_9 [1600,1800]	p_1	-	τ_{10}, τ_3, τ_2	f_{10} [1800,2000]	p_1	τ_{13}	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5		p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5
f_{11} [2000,2200]	p_1	$\tau_{init,13}$	τ_{10}, τ_2, τ_3	f_{12} [2200,2400]	p_1	-	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_{12}, τ_5, τ_9		p_2	τ_6, τ_1	τ_4, τ_{11}, τ_{12}, τ_5
f_{13} [2400,2600]	p_1	τ_7	τ_{10}, τ_2, τ_3	f_{14} [2600,2800]	p_1	τ_{13}	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_{12}, τ_4, τ_5		p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5
f_{15} [2800,3000]	p_1	τ_9	τ_{10}, τ_2, τ_3	f_{16} [3000,3200]	p_1	-	τ_{10}, τ_2, τ_3
	p_2	τ_1, τ_6	τ_4, τ_{12}, τ_5		p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5
f_{17} [3200,3400]	p_1	$\tau_{init,13}$, τ_9	τ_{10}, τ_2, τ_3	f_{18} [3400,3600]	p_1	τ_7, τ_8	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5, τ_{11}		p_2	τ_6, τ_1	τ_{12}, τ_4, τ_5
f_{19} [3600,3800]	p_1	-	τ_{10}, τ_2, τ_3	f_{20} [3800,4000]	p_1	τ_{13}	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_5, τ_{12}		p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5
f_{21} [4000,4200]	p_1	-	τ_{10}, τ_2, τ_3	f_{22} [4200,4400]	p_1	$\tau_{init,13}$, τ_9	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5		p_2	τ_6, τ_1	τ_{13}, τ_4, τ_5, τ_{11}
f_{23} [4400,4600]	p_1	-	τ_{10}, τ_2, τ_3	f_{24} [4600,4800]	p_1	τ_7	τ_{10}, τ_2, τ_3
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5		p_2	τ_6, τ_1	τ_5, τ_{12}, τ_4
f_{25} [4800,5000]	p_1	τ_{13}	τ_{10}, τ_2, τ_3				
	p_2	τ_6, τ_1	τ_4, τ_{12}, τ_5				

Table 3.5: Optimized task mapping $\mathcal{M}_\tau$ for FMS on a 2-core, 2-bank subset of a compute cluster.

Bank	Mapped memory blocks
1	b_1 to b_{15}, b_{24}
2	b_{16} to b_{23}, b_{25} to b_{27}

Table 3.6: Optimized memory mapping $\mathcal{M}_{mem}$ for FMS on a 2-core, 2-bank subset of a compute cluster.

set. Each FTTS frame is divided into two sub-frames, since the FMS has $K = 2$ criticality levels. We configure the simulated annealing search (Algorithm 5 of Section 3.6) for both task and memory mapping optimization with parameters: $a = 0.8$, $Fail_{max} = 100$, T_0 based on the average cost change of 100 random solutions, $T_{final} = 0.1$, $time_{max} = 30$ min. For the task mapping $\mathcal{M}_\tau$ optimization, the probabilities of selecting a sub-frame or core variation to find new solutions are 0.85 and 0.15, respectively. The memory mapping optimization $\mathcal{M}_{mem}$ uses as objective function the average value of the delay matrix D_{avg} and is performed for each visited task mapping solution during design space exploration, i.e., according to the integrated solution II of Section 3.6.3. The overall optimization goal is to maximize the slack time at the end of the frames (equiv. minimize $||barriers||_3$), which indicates a maximal exploitation of computation parallelism and memory accessing parallelism.

We consider all possible configurations with one to eight processing

cores and one to eight memory banks for the deployment of the FMS. The optimized task and memory mapping solution which yields the minimum value for the objective function $||barriers||_3$ is found for the configuration with *two* processing cores and *two* memory banks. Selecting more memory banks or more cores is not beneficial since it does not lead to a solution of lower cost, namely a solution with more slack time at the end of the frames. This is important information for a system designer, who tries not only to design a safe system, but also to allocate the minimal amount of resources.

For the configuration with two cores and two memory banks, the optimization framework returns an admissible FTTS schedule after evaluating 4919 task and memory mapping combinations and converging to one within 4.3 minutes. The optimized task and memory mapping are shown in Table 3.5 and 3.6, respectively. Note that the dependency between tasks $\tau_{init,13}$ and τ_{13} is respected and that the two tasks are scheduled in different frames on the same core such that the minimum distance constraint (536.8 ms) is not violated. For the optimized solution, function *barriers* can be computed based on the memory interference graph $\mathcal{I}$ and the memory latency T_{acc}, as described in Section 3.5.1. The values of *barriers*, i.e., the worst-case sub-frame lengths for every FTTS sub-frame and level of assurance, as computed by our optimization framework, are shown in Table 3.7. Note that for every FTTS frame, the sum of barriers for its two sub-frames, under both levels of assurance, is not greater than the size of the frames, i.e., 200 ms. This shows that the admissibility condition of Eq. (3.2) is valid, which yields the FTTS schedule admissible.

3.7.2 Effect of Platform Parameters and Design Choices on FTTS Schedulability

With the second experiment we intend to evaluate the sensitivity of our optimization approach when certain parameters, e.g., the number of memory banks, the memory access latency T_{acc}, the incoming traffic from the NoC, the number of available processing cores vary. We evaluate schedulability of the FMS application for the alternative configurations (combinations of the above parameters) based on the cost $||barriers||_3$ of the optimized task and memory mapping solution in each case. For the definition of metric $||barriers||_3$, see the discussion on the cost function (3.14) in Section 3.6.1. The lower the cost of the optimized solution, the higher the probability that an admissible FTTS schedule for the considered configuration exists. In all following scenarios, the optimizer converges to a task and memory mapping solution in less than 7 minutes. For the simulated annealing algorithm, we use the same parameters as in the previous section.

First, we evaluate the effect of the *memory access latency* T_{acc} on the

Frame f	Level-1 *barriers*$(f,1)$		Level-2 *barriers*$(f,2)$	
	Subframe1	Subframe2	Subframe1	Subframe2
f_1	18	58.1	90.1	0
f_2	18	58.1	90.1	0
f_3	18	78.1	90.1	0
f_4	48.1	57.9	192	0
f_5	18	58.1	90.1	0
f_6	18	58.1	90.1	0
f_7	18	58.1	90.1	0
f_8	18	78.1	90.1	0
f_9	18	58.1	90.1	0
f_{10}	48.1	57.9	192	0
f_{11}	18	58.1	90.1	0
f_{12}	18	78.1	90.1	0
f_{13}	18	58.1	90.1	0
f_{14}	48.1	57.9	192	0
f_{15}	18	58.1	90.1	0
f_{16}	18	58.1	90.1	0
f_{17}	18	78.1	90.1	0
f_{18}	18	58.1	90.1	0
f_{19}	18	58.1	90.1	0
f_{20}	48.1	57.9	192	0
f_{21}	18	58.1	90.1	0
f_{22}	18	78.1	90.1	0
f_{23}	18	58.1	90.1	0
f_{24}	48.1	57.9	192	0
f_{25}	18	58.1	90.1	0

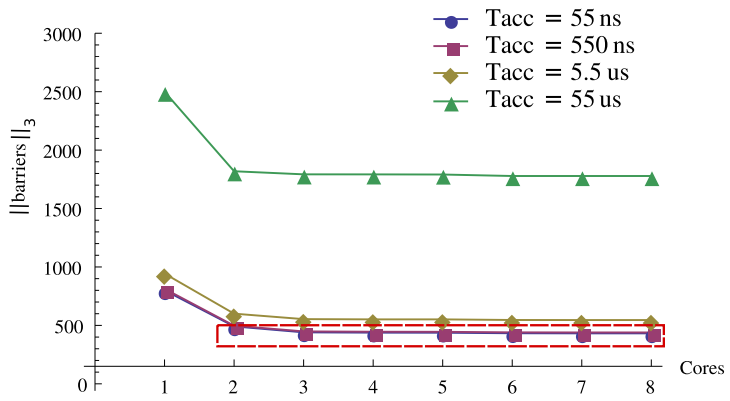
Table 3.7: Computation of *barriers* for $\mathcal{M}_{mem}$ (Table 3.5), $\mathcal{M}_{mem}$ (Table 3.6), T_{acc} = 55ns, memory interference graph $\mathcal{I}$.

FMS schedulability. We assume that the value of T_{acc} varies within {55 ns, 550 ns, 5.5 us, 55 us}. We perform design space exploration after fixing the number of memory banks to two. Figure 3.12(a) shows how the schedulability metric, $||barriers||_3$, changes for the FMS as the number of available cores increases from one to eight, for different T_{acc} values. For each combination of T_{acc} and number of cores, the depicted point in Figure 3.12(a) corresponds to the best found solution by our optimization framework. The value on the y-axis represents the 3-norm $||barriers||_3$ for the optimized task and memory mapping solution. The points within the dashed rectangle correspond to schedulable implementations, namely to combinations of T_{acc} and number of cores m for which the optimized mapping solution is admissible according to Definition 3.1. The points that do not fall into this rectangle correspond to implementations for which the optimizer could not converge to any admissible solution within the given time budget of 30 minutes. We observe that, like in the previous section, the FMS schedulability under FTTS increases or remains stable as the number of cores increases. This is partly explained by the low task set

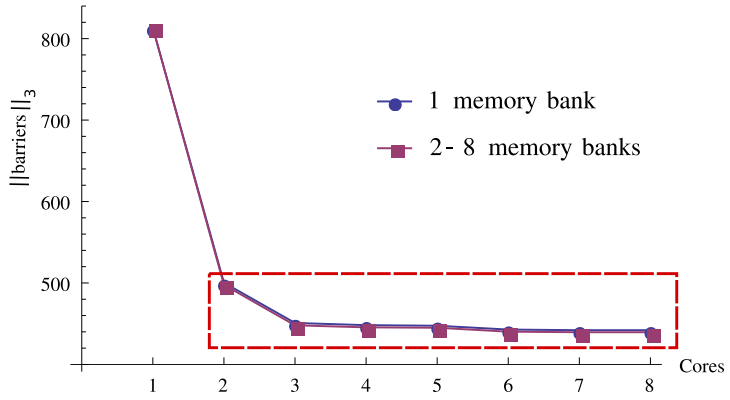
utilization of the FMS. Two processing cores suffice to find an admissible FTTS schedule, whereas more cores are not beneficiary. Moreover, the effect of T_{acc} in schedulability is significant. For $T_{acc} \in \{5.5 \text{ us}, 55 \text{ us}\}$ no admissible schedule can be found even when all cores of the cluster are utilized. This is an important indicator that in shared-memory multi-core and many-core platforms, the increase in number of cores must be followed by a simultaneous increase in memory bandwidth (reduction of T_{acc}) or mechanisms for the reduction of memory contention, for real exploitation of task parallelism.

Second, we evaluate the effect of *data partitioning* on the FMS schedulability. We fix T_{acc} to 550 ns and perform design space exploration for all cluster configurations with one to eight memory banks and one to eight cores. Figure 3.12(b) shows the schedulability metric (cost $||barriers||_3$ of the optimized task and memory mapping solution) as the number of cores increases, for the configurations with one bank (blue line) or more than one banks (magenta line). Deploying more than two memory banks does not improve the optimized solutions. This can be partly explained by the low memory utilization of the FMS (fraction of cluster memory required for task data and communication buffers). Also, note that the cost of the solutions for one bank are only marginally worse than those for several banks. By carefully examining the optimized solutions, we conclude that in cases where no flexibility exists w.r.t. memory mapping, the optimizer tends to select the task mapping by maximally distributing the tasks across the FTTS frames (not letting empty frames), such that a minimal set of tasks are executed in parallel in the same frame and hence, interfere on the memory bank. The periods of the FMS tasks and the considered dimensioning of the FTTS schedule (25 frames over $H = 5$sec) help in this direction, since many tasks have a high degree of freedom in the range of frames to which they can be mapped. We conclude that for the FMS, the combined task and memory mapping optimization performs efficiently, in the sense that the optimizer exploits maximally the flexibility in solving one problem (task mapping) when the flexibility of the second problem (memory mapping) is limited.

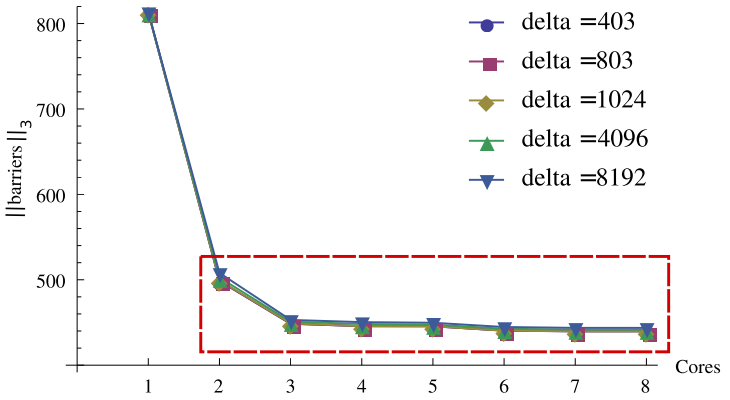
Finally, we evaluate the effect of the *incoming NoC traffic* at the cluster memory on the FMS schedulability. We assume that by selecting different regulation parameters and/or NoC routes for the data flow that is requested by task $\tau_{init,13}$, we can affect the maximum number of NoC Rx accesses to the local memory, δ, such that it varies within {403, 803, 1024, 4096, 8192}. We fix T_{acc} to 550 ns and the number of memory banks to 2. The FMS schedulability metric (cost $||barriers||_3$ of the optimized task and memory mapping solution) for increasing number of cores and for the different δ values is shown in Figure 3.12(c). Again, schedulability is not severely affected by increased incoming NoC traffic. This is achieved in that the optimizer isolates the memory block corresponding to the fetched database entries (b_{27}) so that no or very few FMS tasks are interfering with



(a) FMS schedulability for variable T_{acc}.



(b) FMS schedulability for variable number of banks.



(c) FMS schedulability for variable number of NoC Rx memory accesses in an FTTS frame δ.

Figure 3.12: Effect of platform and design parameters on FMS schedulability under the FTTS policy.

the higher-priority $\mathtt{Rx}_{13}$ requester, thus exploiting the memory accessing parallelism that the two memory banks enable. This way, the WCRT of the FMS tasks becomes immune to changes of δ. This observation justifies the benefits of the combined task and memory mapping optimization, where the interference of the tasks on shared platform resources is considered.

3.8 Comparison to Existing Mixed-Criticality Scheduling Policies

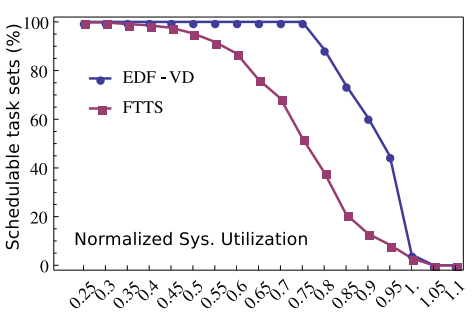
The benefit of using FTTS in the presence of shared platform resources, e.g., memory banks and networks-on-chip, has been discussed and evaluated in Section 3.7. In this section, we evaluate the efficiency of the FTTS policy in finding admissible schedules against state-of-the-art scheduling policies that have been proposed for mixed-criticality systems. Specifically, we intend to evaluate the limitations posed by the (flexible) time-triggered implementation of FTTS and their impact on schedulability. Recall that a comparison between FTTS and other IS-compliant policies has been already presented for synthetic task sets with harmonic periods in Section 2.6.4. Here, we compare FTTS to more dynamic, non IS-compliant state-of-the-art MC scheduling policies, particularly the EDF-VD algorithm for single-core [BBD+12] and its GLOBAL variant for multi-core systems [LB12]. Since these algorithms do not consider resource sharing, comparison is based upon synthetic task sets that require no memory or NoC accesses.

For task set generation we use the algorithm of [LB12] (TaskGen, Figure 4 in [LB12]) for 2 criticality levels. Per-task utilization U_i is selected uniformly from $[U_L, U_H] = [0.05, 0.75]$ and the ratio Z_i of the level-2 utilization to level-1 utilization is selected uniformly from $[Z_L, Z_H]$ =[1,8]. The probability that a task τ_i has $\chi_i = 2$ is set to $P = 0.3$. Period T_i is randomly selected from the set $\{100, 200, 300, 400, 500\}$. Because FTTS cannot handle dynamic preemption, if the assigned execution time of a task is larger than the minimum frame length of the FTTS scheduling cycle, the task is split into sub-tasks through fixed preemption points. The number of preemption points is selected such that each sub-task can "fit" within the FTTS frame with minimal length.

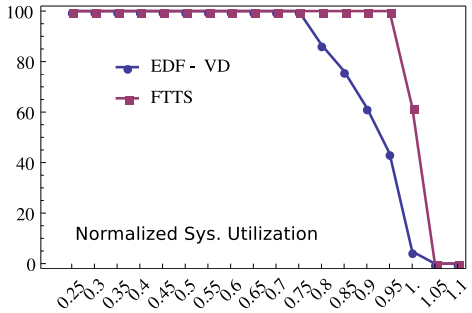
Figures 3.13(a)-3.13(d) (FTTS vs. EDF-VD) and Figures 3.14(a)-3.14(d) (FTTS vs. GLOBAL) show the fraction of task sets that are deemed schedulable by the considered algorithms as a function of the ratio U_{sys}/m (normalized system utilization). U_{sys} is defined in [LB12] as follows:

$$U_{sys} := \max\left\{U_{\mathrm{LO}}^{\mathrm{LO}}(\tau) + U_{\mathrm{HI}}^{\mathrm{LO}}(\tau), U_{\mathrm{HI}}^{\mathrm{HI}}(\tau)\right\}, \tag{3.16}$$

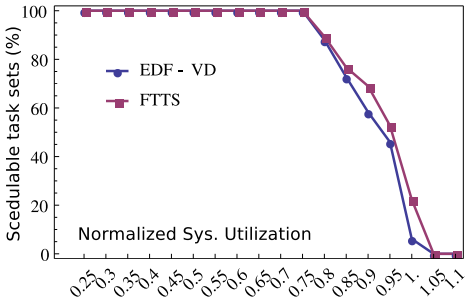
where $U_x^y(\tau)$ represents the total utilization of the tasks with criticality level x for their $y-$level execution profiles (LO≡1, HI≡2). Note that the normalized utilization increases from 0.25 to 1.10 in steps of 0.05. For each utilization point in the graphs, 100 or 1000 randomly generated task sets (as annotated in respective figures) are considered. To check schedulability of each randomly generated task set for FTTS, we use the optimization framework of Section 3.6.1 and check condition (3.2) for the optimized solution. For the design space exploration, we use the same configuration for the simulated annealing algorithm as in Section 3.7 and a



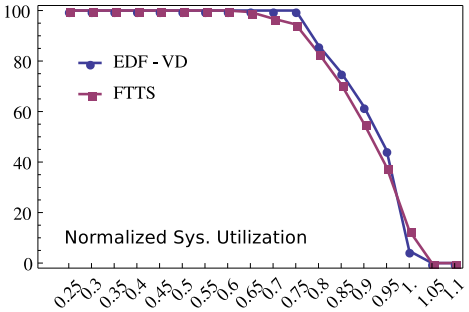
(a) $T_i \in \{100, 200, 300, 400, 500\}$.



(b) $T_i \in \{100\}$.



(c) $T_i \in \{200, 400\}$.



(d) $T_i \in \{200, 400, 800\}$.

Figure 3.13: Schedulable task sets (%) vs. normalized system utilization for FTTS and EDF-VD ($m = 1$), $U_L = 0.05, U_L = 0.75, Z_L = 1, Z_L = 8, P = 0.3$, 1000 task sets per utilization point.

time budget of 10 minutes. In all cases, however, the optimizer converged to a solution in less than 5 minutes. Additionally, to check schedulability of each randomly generated task set for EDF-VD and GLOBAL, we check the sufficient conditions from [BBD+12, LB12]. These conditions are given below.

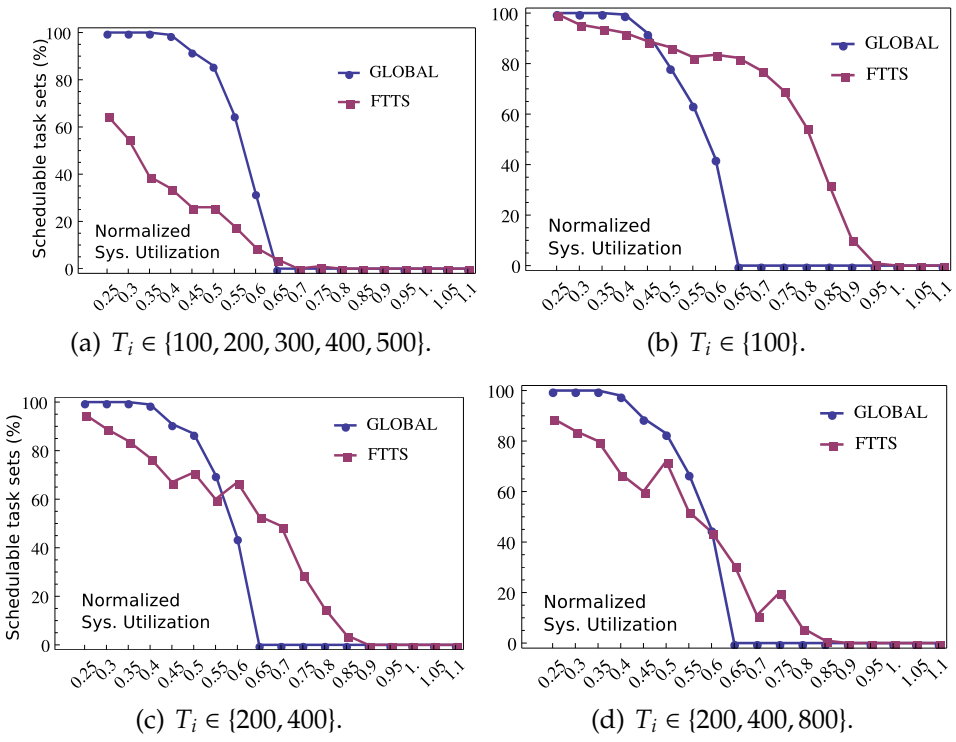
- For EDF-VD on single cores [BBD+12]:

$$U_{\mathrm{HI}}^{\mathrm{HI}}(\tau) + U_{\mathrm{LO}}^{\mathrm{LO}}(\tau) \cdot \frac{U_{\mathrm{HI}}^{\mathrm{LO}}(\tau)}{1 - U_{\mathrm{LO}}^{\mathrm{LO}}(\tau)} \leq 1. \tag{3.17}$$

- For GLOBAL on multicores with m cores [LB12]:

$$U_{\mathrm{LO}}^{\mathrm{LO}}(\tau) + \min\left(U_{\mathrm{HI}}^{\mathrm{HI}}(\tau), \frac{U_{\mathrm{HI}}^{\mathrm{LO}}(\tau)}{1 - 2 \cdot U_{\mathrm{HI}}^{\mathrm{HI}}(\tau)/(m+1)}\right) \leq \frac{m+1}{2}, \tag{3.18}$$

On single-core systems, FTTS faces two limitations compared to EDF-VD, i.e., the fixed preemption points and the time-triggered frames. EDF-VD is more flexible with scheduling task jobs as they arrive and can preempt them any time. The results of Figure 3.13(a) show that as the utilization increases, EDF-VD can schedule 0 up to 52.9% (U_{sys} =



(a) $T_i \in \{100, 200, 300, 400, 500\}$. (b) $T_i \in \{100\}$.

(c) $T_i \in \{200, 400\}$. (d) $T_i \in \{200, 400, 800\}$.

Figure 3.14: Schedulable task sets (%) vs. normalized system utilization for FTTS and GLOBAL ($m = 4$), $U_L = 0.05, U_L = 0.75, Z_L = 1, Z_L = 8, P = 0.3$, 100 task sets per utilization point.

0.85) more MC task sets than FTTS (on average, EDF-VD has 17.9% higher schedulability than FTTS). The impact of the FTTS limitations on schedulability becomes even clearer if we repeat the experiment such that these limitations are avoided. This happens when all tasks have the same period ($T_i = 100$), hence the FTTS cycle consists only of 1 frame. The corresponding results in Figure 3.13(b) exhibit now reverse trends, with FTTS being able to schedule up to 57.2% ($U_{sys} = 1.0$) more task sets than EDF-VD (on average, FTSS has 10.5% higher schedulability than EDF-VD). In fact, if we consider safety-critical applications with harmonic task periods, such as the FMS of Section 3.7, the performance of FTTS is comparable to that of EDF-VD. This can be seen in Figure 3.13(c) and 3.13(d), where the task periods for the generated task sets are selected uniformly from sets $\{200, 400\}$ and $\{200, 400, 800\}$, respectively. In the case of two harmonic periods, FTTS can schedule up to 16.2% (on average 2.2%) more tasks sets than EDF-VD. In the case of three periods, FTTS can schedule up to 8.1% more task sets ($U_{sys} = 1$), but on average across all utilization points, it schedules 1.2% less task sets than EDF-VD. The comparable performance of the two policies in terms of schedulability for equal or harmonic task periods is a significant outcome, given that FTTS was designed targeting at temporal isolation rather than efficiency.

On multicores, we expect GLOBAL to perform more efficiently than FTTS not only because of the previously discussed advantages, but also because it enables task migration. Namely, several jobs of the same task can be scheduled on different cores and a preempted job can be resumed on a different core. The results of Figure 3.14(a) show that the effectiveness of GLOBAL in finding admissible schedules for the generated task sets is up to 65% higher (U_{sys} = 0.40) than for FTTS. Recall, however, that the increased efficiency comes at the cost of ignoring the timing effects of shared resources which are not negligible especially in the presence of task migrations. If the limitations of FTTS are avoided as before, the results (Figure 3.13(b)) are again reversed. Then, FTTS schedules up to 82.3% (U_{sys} = 0.65) more task sets than GLOBAL (on average, FTTS has 20.8% higher schedulability than GLOBAL). Figure 3.14(c) and 3.14(d) show the schedulability vs. utilization trends when the task periods are selected from the harmonic sets $\{200, 400\}$ and $\{200, 400, 800\}$, respectively. In the case of two harmonic periods, can schedule up to 53% (on average 3.8%) more tasks sets than GLOBAL. In the case of three periods, it can schedule up to 31% more task sets, but on average across all utilization points, it schedules 3.6% less task sets than GLOBAL. Therefore, schedulability is again comparable between the two policies when the task periods are harmonic.

It follows that FTTS, despite its imposed limitations for achieving temporal isolation, e.g., the lack of dynamic preemption, the static partitioning of tasks among cores and the fixed-length frames, can actually compete with state-of-the-art scheduling algorithms which were designed for efficiency. In other words, FTTS not only enables Isolation Scheduling to ease the process of certification, but it is also a competent solution for efficient (processing, memory, communication) resource utilization in mixed-criticality environments.

3.9 Summary

This chapter presented a unified analysis approach for computing, memory and communication scheduling. The approach targets modern cluster-based many-core architectures with two shared resource classes: a shared multi-bank memory within each cluster and a network-on-chip (NoC) for inter-cluster communication and access to external memories. To model such architectures and the communication flows through the NoC, we used the Kalray MPPA-256 architecture as reference and introduced a protocol for inter-cluster communication with formally provable timing properties. For the scheduling of mixed-criticality applications on cluster-based architectures, we proposed a policy based on flexible time-triggered and synchronization scheduling (FTTS), which complies with the Isolation Scheduling model of Chapter 2. For this

policy, we presented a worst-case response time analysis which (i) models interference on the shared memory of a cluster by concurrently executing tasks in the cluster and by incoming traffic from the NoC, (ii) bounds safely and tightly the end-to-end delays for data transfers through the NoC, (iii) models the incoming traffic from the NoC in the form of arrival curves from the real-time calculus, (iv) integrates the results of the memory interference analysis and the NoC analysis. Moreover, we proposed design exploration methods targeting at the optimization of resource utilization within a cluster at the levels of computing (core utilization), memory (exploitation of internal memory structure for data partitioning) and communication (management of incoming traffic from a NoC). The applicability and efficiency of the optimization approach were demonstrated for an industrial implementation of a flight management system. Additionally, the FTTS scheduling policy was compared in terms of schedulability to state-of-the-art policies for mixed-criticality systems. The results showed a comparable or even higher schedulability than existing approaches for harmonic workloads.

Later, Chapter 5 presents an implementation of the FTTS scheduler on the Kalray MPPA-256 platform. The methods developed here for system-level design optimization are then put into practice to enable the efficient and timing-predictable deployment of the flight management system and other industrial-representative applications. This implementation will allow us to refine and validate the proposed worst-case response time analysis.

4

A Dedicated Execution Model for Tighter Interference Analysis

Multi-core architectures offer the potential for drastically increasing the performance of embedded real-time systems. However, the potential is usually not fully exploited because of contention on shared platform resources. Concurrently executed tasks mutually block their accesses to the shared resources, causing non-deterministic delays which are difficult to bound. Chapter 2 presented a scheduling model for eliminating such timing interference among tasks with different safety criticality levels. Chapter 3 presented a method for bounding the effects of resource contention on the response time of concurrently executed tasks (with the same criticality level) on manycores with shared memory and network-on-chip infrastructure. The proposed worst-case response time analysis is applicable to shared resources that are round-robin arbitrated. Its extension to more complex arbitration schemes is, however, not trivial and may result in excessive pessimism. In fact, even for round-robin arbitration, the analysis of Chapter 3 may be pessimistic in certain cases, since it is based on the assumption of worst-case interference for every single resource access of a task, which may not be realistic depending on the accessing patterns of the concurrently executed tasks. In order to reduce this pessimism, we propose in this chapter a dedicated execution model and a state-based method for worst-case response time analysis.

Specifically, we consider real-time tasks composed of superblocks, i.e., sequences of computation and resource access phases. Resource accesses are synchronous (blocking), causing execution on the processing core to stall until the access is served. For such systems, we present a state-based modelling and analysis approach based on timed automata which

can model accurately different resource arbitration schemes. Based on it, we compute safe bounds on the worst-case response times of tasks. The scalability of the approach is increased significantly by abstracting several cores and their tasks with one arrival curve, which represents their resource accesses and computation times. This curve is then incorporated into the timed automata system model. The accuracy and scalability of the approach are evaluated with benchmark applications and a real-world automotive application, and compared against simulation-driven and state-of-the-art analytic approaches.

4.1 Introduction

Multi-core systems become increasingly popular as they allow performance gains by exploiting parallelism without sacrificing too much on power consumption or cost. However, the reduction in cost is achieved by sharing resources among the processing cores. Such shared resources can be buses, caches, scratchpad memories, main memories, DMA engines. In safety-critical embedded systems, such as controllers in the Automotive Open System Architecture (AutoSAR) [aut] or the flight management system of the previous chapter, accesses to the shared resources can be non-periodic and bursty, which may result in missed deadlines in the worst case. Therefore, a designer needs to consider the interference due to contention on the shared resources in order to verify the real-time properties of the system. At the same time, the interference-induced delays need to be *tightly* bounded to avoid extreme resource over-provisioning and hence, platform under-utilization.

Performing timing analysis for such systems is challenging because the number of possible interleavings of resource accesses from the different processing cores can be very large. Analytic approaches for bounding the worst-case response times (WCRT) under resource contention, such as the method presented in Section 3.5, have the advantage of scalability. Since they are based on closed-form expressions, they can be applied to analyze systems with an arbitrarily large number of cores or tasks. On the other hand, they are often based on over-approximations which result in overly pessimistic WCRT estimations, particularly in cases of state-based arbitration mechanisms like first-come-first-serve (FCFS) or round-robin (RR). This shortcoming is of concern, as industrial standards like the FlexRay bus protocol [fle] or the first-ready first-come-first-serve (FR-FCFS) arbitration policy [RDK+00, NALS06] for DDR memory controllers, explicitly exploit state-dependent behaviors. In this chapter, we rely on the formalism of timed automata [AD90] in order to model accurately the behavior of such state-dependent arbiters and we use model checking to derive tight WCRT bounds.

Contrary to analytic approaches, model checking is based on

exhaustive search of a state space and can reveal the actual worst-case response time of a task without over-approximations. Additionally, the expressiveness of timed automata enables an accurate modelling of any given resource arbitration scheme. These advantages come at the cost of analysis complexity, which often makes state-based modelling and analysis not applicable to systems of industrial size. In order to alleviate the state-space explosion problem which is inherent to model checking, we combine in this chapter model checking with two main abstractions. The first abstraction is that tasks follow a dedicated execution model, i.e., they are composed of *superblocks*. The second abstraction is that some of the processing cores can be substituted by simpler models, i.e., *arrival curves* that represent their resource accesses in the time interval domain.

The superblock model for structuring real-time tasks is based on the assumption that tasks are composed of sequentially executed superblocks for which the minimum/maximum number of resource accesses and execution times are known. This model, which fits very well signal-processing and control applications, has been extensively used in several methods [PSC+10, SCT10, SPC+11] for WCRT estimation in resource-sharing multicores with synchronously accessed resources. Different variants of the model are compared in [SPC+10]. They differ mostly in that superblocks may have phases where resource accesses are not required (computation-only phases). Such phase structure can be enforced by a compiler as shown in [PBB+11]. [FDNG+09] shows how the superblock model can be mapped to an AutoSAR system for automotive applications.

Arrival curves as known from network and real-time calculus [TCN00] are used to bound the maximum number of events arriving in any time interval of any given length. Several methods [NSE09, PSC+10, SNE10] have utilized them before to represent the maximum number of resource accesses from a task. The novelty of this chapter is their integration into a timed automata model of the system in order to combine accurate modelling of complex arbitration schemes with analysis scalability.

Our results are applicable to hardware platforms without timing anomalies, such as the fully timing compositional architecture proposed by Wilhelm et al. [WGR+09]. We assume that a task partitioning to cores is predefined and that tasks are scheduled on each core sequentially according to a static schedule, which is the case for example within each sub-frame of an FTTS schedule (Section 3.4).

Contributions. The main contributions of the chapter can be summarized as follows:

- For the proposed system model, we introduce a state-based WCRT analysis approach. Timed automata are used to model concurrent execution of processing cores and their tasks (superblocks) as well as resource access arbitration according to an event-driven (FCFS, RR), time-driven (TDMA) or hybrid (FlexRay) policy. The Uppaal

model checker [BDL04] is then used to derive the exact WCRT of each task in the system.

- To increase the scalability of the approach, we introduce an abstraction based on arrival curves. We show how to compute tight curves that bound the number of resource accesses from each core in the time interval domain. Using the interfaces between real-time calculus and timed automata presented in [LPT10], the timed automata model of the system is reduced by replacing the models of several processing cores with a single model that can generate non-deterministic streams of access requests according to the arrival curves of the abstracted cores.
- We demonstrate the accuracy and scalability of the approach using a set of embedded benchmark applications (EEMBC) and a real-world automotive application. The WCRT bounds derived by the proposed method are compared to those obtained by a simulation framework and to state-of-the-art analytic approaches. The results show that our state-based analysis approach yields safe WCRT estimates and can scale efficiently to a large number of cores, without compromising the accuracy of the WCRT bounds.

Outline. The chapter is organised as follows. Section 4.2 presents related analytic or state-based methods for bounding the worst-case response times of tasks under resource contention. Section 4.3 shortly introduces some of the necessary theory on timed automata and real-time calculus. Section 4.4 defines our system model. Section 4.5 introduces the formal model of a system using timed automata and the new state-based analysis method. Section 4.6 and 4.7 address explicitly the challenge of analysis scalability. Finally, Section 4.8 presents the empirical evaluation of the proposed approach and Section 4.9 summarizes the main results of the chapter.

4.2 Related Work

Performing timing analysis for multi-core systems with shared resources is challenging as the number of possible orderings of access requests that arrive at a shared resource can be exponentially large. Additionally, resource accesses can be asynchronous (non-blocking) such as message passing or synchronous (blocking) such as memory accesses due to cache misses. For the asynchronous accesses, the timing analysis needs to take into account the arrival pattern of accesses from the processing cores and the respective backlogs. In this case, traditional timing analysis methods such as real-time calculus [TCN00] and SymTA/S [HHJ+05] can compute accurate bounds on the worst-case response times (WCRT) of

tasks and the end-to-end delays of accesses. For the synchronous accesses, if we assume in-order execution without support for simultaneous multi-threading (which describes for example execution on the Kalray MPPA-256 processing cores [dDvAPL14]), an access request stalls the execution until the access is completed. This leads an increase of the task's WCRT. This increase exists because, once an access request is released, the task execution cannot be preempted. Moreover, once service of an access request starts, the latter cannot be preempted by other accesses. Bounding the blocking times of tasks under these assumptions is far from trivial, as one has to take into account the currently issued accesses from all other cores and the state of the resource arbiter. In this chapter, like in Chapter 3, we consider synchronous accesses. Note that, in the following, we use the terms 'synchronous' and 'blocking' interchangeably.

Schliecker et al. [SNE10] proposed methodologies to analyze the worst-case delay of a task due to synchronous accesses to shared memory. The authors used event activation models [SNE09] to capture the memory request distances by single tasks or by sequences of tasks that are executed on the same core, assuming that a minimum and maximum number of access requests in particular time windows is known. By considering preemptive fixed-priority scheduling on the processing cores and first-come first-serve (FCFS) arbitration on the shared memory, the worst-case memory delay is derived in an iterative fashion based on the access patterns of higher priority tasks and tasks executing concurrently on other cores. The authors evaluated the approach with a system with two processing cores. Shortcomings of this method which lead to increased pessimism were identified and addressed in the work of Dasari et al. [DAN+11]. By following a similar approach, yet considering the access request distribution of the tasks in a time window (rather than a uniform distribution as in [SNE10]) and non-preemptive scheduling, this work provided tighter bounds on the worst-case delays that a task can experience due to interference on the shared bus to the memory. In [NSE09] Negrean et al. considered the multiprocessor priority ceiling protocol where tasks have globally assigned priorities, and similarly used a system with two processing cores for evaluation. An alternative approach to analyzing the duration of accesses to a time-division-multiple-access (TDMA)-arbitrated resource was presented by Kelter et al. [KFM+11]. The proposed approach is based on abstract interpretation and integer linear programming. It statically computes all possible offsets for access requests within a TDMA arbitration cycle. This way the authors aimed at high accuracy of analysis results and reasonable analysis times. None of the above works considered complex scheduling policies such as FlexRay [fle] or systems with a high number of processing cores, as analyzed in this chapter.

Pellizzoni et al. [PSC+10] and Schranzhofer et al. [SCT10, SPC+11] proposed methods for WCRT analysis in multi-core systems where

the shared resource is arbitrated according to FCFS, round-robin (RR), TDMA or a hybrid time/event-driven strategy which combines TDMA and FCFS/RR. Contrary to previous works, the proposed methodology was shown to scale efficiently. The analysis, however, used over-approximations which can result in extremely pessimistic results, particularly in cases of state-based arbitration mechanisms, like FCFS or RR. This shortcoming is of concern, as modern multi-core architectures tend to exploit complex, state-dependent behaviors. The presented work in this chapter exploits the same model of computation (superblocks). It uses the concept of event arrival curves for capturing the non-deterministic arrivals of service requests at the shared resource. The main difference to previous results can be seen in the fact that our work relies on a state-based modelling and analysis approach, based on model checking. This way we ensure that the system model captures accurately the behavior of the state-dependent resource arbiters and minimizes analysis pessimism.

In later sections, we consider the FlexRay protocol [fle] as an example of a complex state-based arbitration policy, even though it is not originally designated for bus arbitration in shared-memory systems. FlexRay was analyzed by Pop et al. [PPE+08] and by Chokshi et al. [CB10], but these approaches dealt only with the asynchronous case of resource accesses. On the other hand, FlexRay with synchronous accesses has never been modelled with analytic approaches. In [SPC+11], Schranzhofer et al. modelled a hybrid arbitration mechanism as a combination of a static (TDMA) and a dynamic segment, the latter behaving according to FCFS or RR. Model checking enables us to model and analyze FlexRay with synchronous accesses for the first time.

Model checking techniques have been previously applied for timing analysis in resource-sharing multicores by Lv et al. [LYGY10] and Gustavsson et al. [GELP10]. The methods dealt accurately with complex arbitration schemes, however, none of them could scale efficiently beyond two cores due to the state explosion problem. We address this problem by combining state-based and analytic techniques for the modelling and analysis of multi-core systems.

Such techniques were introduced by Lampka et al. [LPT10, LPT12] for the combination of timed automata [AD90] and real-time calculus [TCN00]. Altisen and Moy [AM10] handled the case of synchronous data-flow component models and real-time calculus, while Simalatsar et al. [SRL+11] introduced the coupling of parametric timed automata and real-time calculus on top of an SMT-based analysis technique for deriving regions of parameters for task activation patterns under which a system is scheduled. Our WCRT analysis in multi-core resource-sharing settings is based on the aforementioned coupling of timed automata and real-time calculus [LPT10, LPT12], for which we exploit specific concepts as implemented in the timed model checker Uppaal [BDL04, BY04].

4.3 Background theory

In this section, we briefly introduce some important concepts from the theories of timed automata and real-time calculus which will be needed in the remainder of the chapter. For a more detailed presentation, the reader is referred to the respective literature.

4.3.1 Timed Automata

A timed automaton (TA) [AD90] is a state machine extended with clocks. Let C be a set of clocks and let *ClockCons* be a set of constraints on these clocks. In timed automata, the clock constraints are conjunctions, disjunctions and negations of atomic (clock) guards of the form $x \bowtie n$, where $x \in C, n \in \mathbb{N}_0$ and $\bowtie \in \{<, \leq, >, \geq, =\}$. A TA $\mathcal{T}$ is then defined as a tuple $\mathcal{T} = (Loc, Loc_0, Act, C, \hookrightarrow, I)$, where Loc is a finite set of locations, $Loc_0 \subseteq Loc$ is the set of initial locations, Act is a (finite) set of action (or edge) labels, C is the finite set of clocks, $\hookrightarrow \subseteq Loc \times ClockCons(C) \times Act \times 2^C \times Loc$ is an edge relation and $I : Loc \rightarrow ClockCons(C)$ is a mapping of locations to clock constraints, the latter being referred to as location invariants.

Let the active location be the location in which the TA currently resides. The operational semantics associated with a TA can be informally characterized as follows:

Delay transitions. As long as the location invariants of the active location are not violated, time may progress with all clocks increasing at the same speed.

Edge executions. The traversal of an edge potentially changes the active location; self-loops are possible. The traversal or "edge execution" is instantaneous and possible as long as the source location of the edge is marked active and the guard of the edge evaluates to true. Upon edge executions, clocks can be reset.

This operational semantics yields that for a TA one may observe infinitely many different behaviors. This is because edge executions may occur at any time, namely as long as the edge guard evaluates to true. However, with the concept of clock regions [AD90] it is possible to capture all possible behaviors in a finite state graph, such that timed reachability is decidable. In fact, modern timed model checkers incorporate clock zones [LWYP99] as they often result in a coarser partitioning of the clock evaluation space in comparison to the original definition of clock regions.

In this chapter we employ the timed model checker Uppaal [BDL04], which implements timed safety automata extended with variables. Similarly to clocks, variables can be used within guards of edges and location invariants. Upon an edge execution, a variable can be updated. The used variable values must build a finite set, which, however, does not

need to be known beforehand, i.e., it can be constructed on-the-fly upon the state space traversal.

For modelling execution and resource access in a multi-core system, we use *networks* of cooperating TA. In such a setting, clocks and variables can be shared among the different TA, and dedicated sending and receiving edges are jointly traversed depending on the selected synchronization policy ($1:1$ binary synchronization or $1:n$ broadcast synchronization). For a system model, expressed as a network of TA, the Uppaal model checker enables verification of functional and temporal properties. For conciseness, we omit further details, and refer the interested reader to the literature [AD90, Yov98, BDL04].

4.3.2 Real-time Calculus

Real-time calculus (RTC) [TCN00] is a compositional methodology for system-level performance analysis of distributed real-time systems. We briefly recapitulate the basic concepts that are used in this chapter.

In the context of real-time systems design, the timing behavior of event streams is usually characterized by real-time traffic models. Examples of such typically used models are periodic, sporadic, periodic with jitter, and periodic with burst. RTC provides an alternative characterization of event streams: a pair (α^l, α^u) of arrival curves bounds the number of events seen on the stream for any interval of length $\Delta \in [0, \infty)$. Let $R(t)$ be a stream's cumulative counting function which reports the actual event arrivals for the time interval $[0, t)$. The upper and lower arrival curves bound $R(t)$, i.e., the possible event arrivals in the time interval domain, as follows:

$$\alpha^l(t-q) \leq R(t) - R(q) \leq \alpha^u(t-q) \text{ with } 0 \leq q \leq t. \tag{4.1}$$

As each event from a stream may trigger some behavior within a down-streamed component, arrival curves provide an abstract lower and upper bound on the amount of resources demanded for processing the events within a time interval $\Delta = t-q$. Note also that for a given pair (α^l, α^u) there might be a (possibly infinite) set of event streams, namely all streams for which the counting function satisfies Eq. (4.1).

In this work we restrict our attention to the case of *discrete amounts* of events and a simple burst-model for event arrivals. In RTC, such scenarios can be modelled as staircase curves. In particular, the upper arrival curve $\alpha^u(\Delta)$ can be defined as the staircase function:

$$\alpha^{st}(\Delta) := B + \left\lfloor \frac{\Delta}{\delta} \right\rfloor \cdot s. \tag{4.2}$$

Parameter $B > 0$ models the burst capacity, namely the number of events that can arrive at the same time instant in a stream upper-bounded by $\alpha^{st}(\Delta)$. Parameters δ and s are related to the maximum long-term arrival rate of events in the stream and the step size (y-offset), respectively.

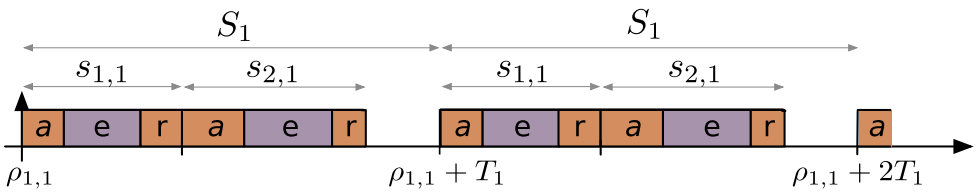


Figure 4.1: Two consecutive processing cycles of superblock sequence $S_1 = \{s_{1,1}, s_{2,1}\}$ which is executed on core p_1 with period T_1 and initial phase $\rho_{1,1}$. Each superblock complies with the dedicated superblock model, consisting of an acquisition, an (access-free) execution and a replication phase. Superblocks are triggered sequentially within each processing cycle.

The timing behavior of streams modelled as staircase arrival curves (Eq. (4.2)) can be correctly and completely modelled with timed automata models as described in [LPT10].

4.4 System Model

This section presents the models of real-time task sets and resource-sharing multi-core architectures that are considered in our analysis and the problem statement.

4.4.1 Task Set and Processing Cores

We consider systems with m processing elements or cores, $\mathcal{P} = \{p_1, \ldots, p_m\}$. The cores in $\mathcal{P}$ support in-order execution without simultaneous multithreading, such that no timing anomalies exist and the property of timing compositionality holds [WGR+09]. The processing cores execute independent tasks but can access a common resource, such as a bus to a shared memory. Accesses to this resource are synchronous (blocking), namely they cause execution to stall until they are served.

We assume a given task partitioning, in which a periodic task set τ_j is assigned to a predefined processing core $p_j \in \mathcal{P}$. All tasks is τ_j have a common period T_j and implicit deadlines, i.e., their deadline is equal to T_j. For simplicity, we assume that the first job of each task arrives at time zero. For structuring the tasks, we exploit the so called superblock model [SCT10]. This phase-based model of computation assumes that each periodic task consists of a sequence of superblocks. The superblocks are non-preemptable execution units with known lower/upper bounds on their computation time and the number of resource accesses that they may require. The number of superblocks per task is fixed and different tasks can have different numbers of superblocks.

For each core p_j, we assume a static, non-preemptive schedule for the superblocks of the assigned task set τ_j. The schedule defines a fixed

order of execution of the superblocks. The superblocks that belong to a single task in τ_j can be ordered sequentially (one after the other) or their execution can be interleaved with superblocks of other tasks. Let the static schedule of superblocks on core p_j be denoted $\mathcal{S}_j = (s_{1,j}, s_{2,j}, \ldots, s_{|\mathcal{S}_j|,j})$, where index i in $s_{i,j}$ defines the execution order (not the corresponding task). This schedule is repeated periodically with period T_j, to which we also refer as processing cycle. Processing cycles may be different among the cores (depending on the periods of the assigned task sets). We assume that T_j is large enough so that all superblocks in $\mathcal{S}_j$ can complete execution within a processing cycle even under resource contention scenarios. Namely, deadline misses are *not* possible by design and successive processing cycles cannot overlap.

In the first processing cycle, the first superblock $s_{1,j}$ of sequence $\mathcal{S}_j$ starts executing at time $\rho_{1,j}$. The superblocks in S_j are repeatedly executed then in $[\rho_{1,j}, \rho_{1,j}+T_j)$, $[\rho_{1,j}+T_j, \rho_{1,j}+2T_j)$ and so forth, with each superblock $s_{i+1,j}$ being triggered upon completion of its predecessor, $s_{i,j}$[1]. For an illustration, Figure 4.1 depicts the execution of superblock sequence $S_1 = \{s_{1,1}, s_{2,1}\}$ on core p_1, where superblocks $s_{1,1}, s_{2,1}$ may belong to the same or different tasks. The starting times of processing cycles on different cores may be synchronized, such that the first superblock in all first processing cycles starts at time 0 ($\rho_{1,j} = 0, \forall j$), or non-synchronized, with $\rho_{1,j} \in [0, T_j)$.

In order to reduce non-determinism with respect to the occurrence of access requests, every superblock is divided into three phases, known as *acquisition*, *execution* and *replication*, which are denoted with a, e, r in Figure 4.1. As an example, let us consider a system with a shared main memory. During the acquisition phase, a superblock reads the required data from the main memory. During the execution phase, computations are performed on the local processing core. During the replication phase, the superblock writes the modified/new data back to the main memory. This is a common model for signal-processing and control real-time applications. For our analysis, we consider in particular the dedicated superblock model, in which resource accesses are restricted to the acquisition and the replication phase and no accesses are allowed in the execution phase. The dedicated superblock structure is the first abstraction proposed in this chapter, since the restriction of resource accesses to dedicated phases leads to increased predictability when analyzing the system's timing behavior. Schranzhofer et al. have shown that the schedulability of the dedicated sequential superblock model dominates that of other models in which accesses may occur any time during execution [SPC+10].

As a result of the discussed structure, a superblock is fully defined by the following parameters: the minimum and maximum number of access requests during its acquisition and replication phase, $\mu_{i,j}^{min,\{a,r\}}$ and $\mu_{i,j}^{max,\{a,r\}}$,

[1] Idle intervals between successive superblocks may also exist.

and the minimum and maximum computation time during its execution phase, $e_{i,j}^{min}$ and $e_{i,j}^{max}$. For simplification, we consider the computation time to initiate the accesses in the acquisition or replication phase, $e_{i,j}^{\{a,r\}}$, equal to zero. If this time is too large, i.e., it cannot be neglected, we divide the corresponding acquisition or replication phases into several superblocks with smaller acquisition/replication and execution phases so that eventually each superblock phase features either computation or accesses only. Note that the terms computation time and execution time are used interchangeably in the following and do not include the time spent on resource accesses or blocking due to contention.

For a superblock with logical branches, the above parameters may be overestimated. In any case, they are assumed to be conservative, i.e., best-case bounds may be too optimistic and upper bounds too pessimistic, such that the worst-case execution time can be safely bounded. We assume that the access request parameters can be derived either by profiling and measurement for soft real-time systems, as shown in [PBCS08], or by applying static analysis and abstract interpretation techniques, as provided, e.g., by the aiT analysis tool [Inf], when hard bounds are necessary. Note that these parameters define the worst-case *execution time* of a superblock when this is executed in isolation, without any interfering superblocks in parallel. Our analysis in Section 4.5 will enable a derivation of the worst-case *response time* of each superblock. Response time is defined as the elapsed time between the execution start and completion of a superblock, and it considers potential delays due to contention on the shared resource. Once the worst-case execution (response) time of all superblocks is known, the worst-case execution (response) time of a task is derived as the sum of the respective times of all constituent superblocks.

The superblock model in practice. For systems employing caches, we rely on the PRedictable Execution Model (PREM) to carefully control when tasks access shared memory. Under PREM, a task's superblocks are specially compiled such that during the acquisition phase, the core accesses either main memory or last level cache to prefetch all required data for the superblock execution into the private core cache. During the subsequent execution phase, the superblock performs computations on the previously fetched data, without incurring any private cache misses. Finally, as first proposed in [BYPC12], during the replication phase, write-backs can be forced for modified cache lines in the private cache [SPC+11]. This ensures that modified data are written back to the shared memory resource. In the PREM implementation [PBB+11] which we employ for the empirical evaluation (Section 4.8), PREM-compliant tasks are produced through modifications of existing applications written in standard high-level languages such as C. Certain constraints are present which limit the applications that can be made PREM-compliant. These constraints, however, are not significantly more restrictive than those imposed by

state-of-the-art static timing analysis tools.

4.4.2 Shared Resource and Arbiter

In the considered systems, superblock execution is suspended every time an access request is issued, until the latter is completely processed by the resource arbiter. Once the arbiter grants access of a request, the accessing time is equal to T_{acc} time units. That is, if a superblock $s_{i,j}$ could access the shared resource at any time, its worst-case execution time would be $e^{max}_{i,j} + (\mu^{max,a}_{i,j} + \mu^{max,r}_{i,j}) \cdot T_{acc}$. It is assumed that access can be granted to at most one request at a time and that processing of an ongoing access cannot be preempted. Once a pending access request is served, either execution on the corresponding core is resumed by performing computations or a new access request is issued or the core remains idle until the start of a new processing cycle. Access to the shared resource is granted by a dedicated arbiter. The implemented arbitration policy can be time-driven, e.g., TDMA, event-driven, e.g., FCFS or RR or hybrid, e.g., the FlexRay bus protocol. A more detailed discussion on the possible arbitration schemes follows.

TDMA Arbiter. In a TDMA arbitration scheme, access to the shared resource is statically organized by assigning time slots to each core, such that only one core can access the resource at a time. TDMA arbitration policies are widely used in timing and safety-critical applications to increase timing predictability and alleviate schedulability analysis, since they eliminate mutual interferences of tasks executing on different cores by construction [RAEP07, AEPR08, SCT10, KFM+11]. A TDMA arbiter uses a predefined cycle of fixed length, which is specified as a sequence of time slots. The time slots can be of variable lengths and there is at least one slot for each core $p_j \in \mathcal{P}$. An access request issued by a superblock on core p_j during the i-th time slot of the TDMA cycle is enabled immediately if the latter slot is assigned to core p_j and the remaining time of the slot suffices to process the new access. Requests that arrive "too late" have to wait until the next allocated slot. To provide meaningful results, we assume that all slots in a TDMA schedule are at least of length T_{acc}.

RR Arbiter. RR-based arbitration can be seen as a dynamic version of TDMA with a varying arbitration cycle. This is because the unused slots of the cycle are skipped whenever the respective cores do not need to access the shared resource. To implement this behavior, the RR arbiter checks repeatedly (circularly) all cores in $\mathcal{P}$, starting with the first identifier, p_1, up to the last one, p_m. If the core with the currently considered identifier p_i has a pending request, access is granted to it immediately. Otherwise, the arbiter checks the next identifier, and so forth.

FCFS Arbiter. The FCFS resource arbiter is responsible for maintaining a first-in first-out (FIFO) queue with the identifiers of the cores which have

a pending access request. Access is granted based on the time ordering of their occurrence, i.e., the oldest request is served first. This scheme guarantees fairness given that, if core p_i issues an access request before core p_j, p_i's request will be served at an earlier point in time, without considering any priorities between the two cores.

FlexRay Arbiter. The FlexRay protocol [fle] has been introduced by a large consortium of automotive manufacturers and suppliers. It enables sharing of a resource (usually interconnection bus among the processing cores of an automotive system), featuring both time and event-driven arbitration. In the FlexRay protocol, a periodically repeated arbitration cycle is composed of a static (ST) and a dynamic (DYN) segment. The ST segment uses a generalized TDMA arbitration scheme, whereas the DYN segment applies a flexible TDMA-based approach. The lengths of the two segments may not be equal, but they are fixed across the arbitration cycles. Both segments are defined as sequences of time slots. The ST segment includes a fixed number of slots with constant and equal lengths, d. Each slot is assigned to a particular core and one or more access requests from this core can be served within its duration. The DYN segment is defined as a sequence of *minislots* of equal length, $\ell \ll d$. The actual duration of a slot depends on whether access to the shared resource is requested by the corresponding core or not: if no access is to be performed, then the slot has a very small length (*minislot length,* ℓ). Otherwise, it is resized to enable the successful processing of the access (*access length*, here equal to T_{acc}). To obtain reasonable results, we assume that each ST slot as well as the DYN segment are at least equal to T_{acc}. The assignment of ST or DYN (mini)slots to the processing cores is static. However, since the introduction of FlexRay 2.0, cycle multiplexing has become also possible for the DYN segment, i.e., some minislots may be assigned to different cores in different cycles, resulting in more than one alternating arbitration schedule.

The above description implies that in the static part of FlexRay, like in TDMA, interference can be neglected due to isolation. In the dynamic part, however, the actual delay of an access is interference-dependent, which makes it difficult to analyze without a state-based approach. The accurate modelling and analysis of such an arbitration policy (and similar policies) for the case of synchronous resource accessing has been one of the major motives of the developed approach in this chapter.

4.4.3 Problem Statement

The problem that we address in this chapter can be formulated as follows. Given the superblock sequences S_j for each core $p_j \in \mathcal{P}$ (incl. all relevant execution and resource access bounds for the superblocks), the corresponding periods T_j, the initial phases $\rho_{1,j}$, the resource access latency T_{acc}, and one of the aforementioned resource arbitration policies

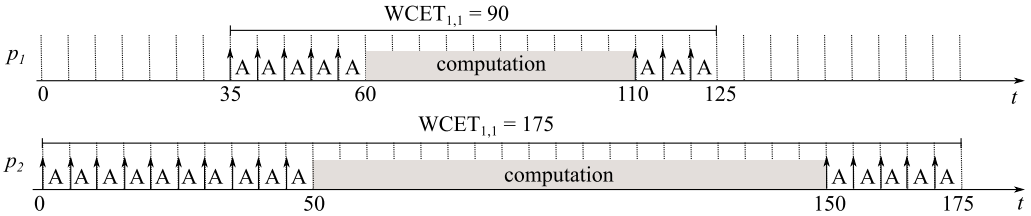


Figure 4.2: Superblocks $s_{1,1}$ and $s_{1,2}$ executed in isolation. 'A' boxes denote accesses to the shared resource with latency $T_{acc} = 5$.

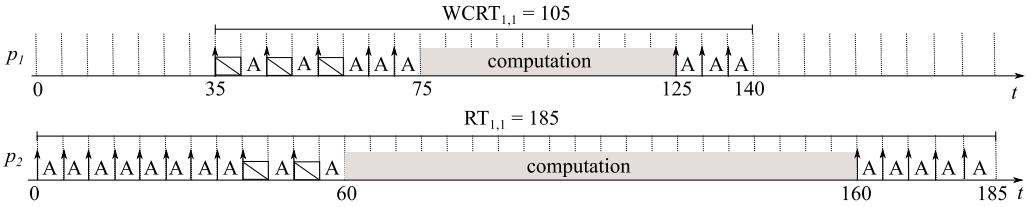


Figure 4.3: Superblocks $s_{1,1}$ and $s_{1,2}$ executed in parallel on cores p_1, p_2 in the first processing cycle. Marked boxes denote blocking time due to contention on the round-robin resource arbiter.

with relevant parameters (e.g., cycle length and slot assignment for a TDMA arbiter), compute a *safe* and *tight* worst-case response time (WCRT) bound for each superblock $s_{i,j} \in S_j$ on each core $p_j \in \mathcal{P}$.

Analysis is performed under the assumption that the total worst-case response time of *all* superblocks $s_{i,j} \in S_j$ cannot be greater than T_j. Namely, all superblocks can be fully executed within a processing cycle. We illustrate with an example some of the challenges that are related to the computation of a tight WCRT bound under resource contention.

Example 4.1. *Consider a system with $m = 2$ processing cores and a shared memory with RR arbitration and access latency $T_{acc} = 5$ time units. Superblock sequence $S_1 = \{s_{1,1}\}$ consists of a single superblock with $\mu_{1,1}^{a,min} = \mu_{1,1}^{a,max} = 5$, $e_{1,1}^{min} = e_{1,1}^{max} = 50$, $\mu_{1,1}^{r,min} = \mu_{1,1}^{r,max} = 3$, $T_1 = 400$, $\rho_{1,1} = 35$. Superblock sequence $S_2 = \{s_{1,2}\}$ also consists of a single superblock with $\mu_{1,2}^{a,min} = \mu_{1,2}^{a,max} = 10$, $e_{1,2}^{min} = e_{1,2}^{max} = 100$, $\mu_{1,2}^{r,min} = \mu_{1,2}^{r,max} = 5$, $T_2 = 400$, $\rho_{1,2} = 0$. Figure 4.2 depicts the worst-case execution time (WCET) of the two superblocks when they are executed in isolation. Based on the superblock parameters and T_{acc}, it follows trivially that $WCET_{1,1} = 90$ and $WCET_{1,2} = 175$. When we consider the parallel execution of $s_{1,1}$ and $s_{1,2}$, a safe WCRT bound for each superblock can be derived in a similar way as in Chapter 3, by assuming worst-case interference on the shared memory (every memory access is delayed by an access from the other core). In this case, for instance, the WCRT of $s_{1,1}$ can be computed as $WCRT_{1,1} = m \cdot T_{acc} \cdot (\mu_{1,1}^{max,a} + \mu_{1,1}^{max,r}) + e_{1,1}^{max} = 130$. Figure 4.3, though, illustrates the actual WCRT of $s_{1,1}$ which is equal to $WCRT_{1,1} = 105$. The reason for the pessimism of the former result is that the acquisition/replication phases of the two superblocks cannot fully overlap (maximally three resource accesses of $s_{1,1}$ can be delayed by accesses from $s_{1,2}$).*

Note that this was a very simple (deterministic) example, with only one superblock per core, equal minimum/maximum bounds for the resource access and computation time parameters, equal processing cycles, and only two cores. If any of these parameters change, e.g., if there is a large number of cores or different minimum/maximum bounds for the superblock parameters, the possible interference scenarios increase rapidly. Enumerating all feasible scenarios for the derivation of a tight WCRT bounds becomes intractable, which is the reason why we typically resort to conservative estimations like before. This, however, comes at the cost of over-provisioning and resource under-utilization.

To avoid the pessimism of analytic approaches for the derivation of WCRT bounds, we present in the next section a state-based method for system modelling and we derive safe and tight WCRT bounds based on model checking.

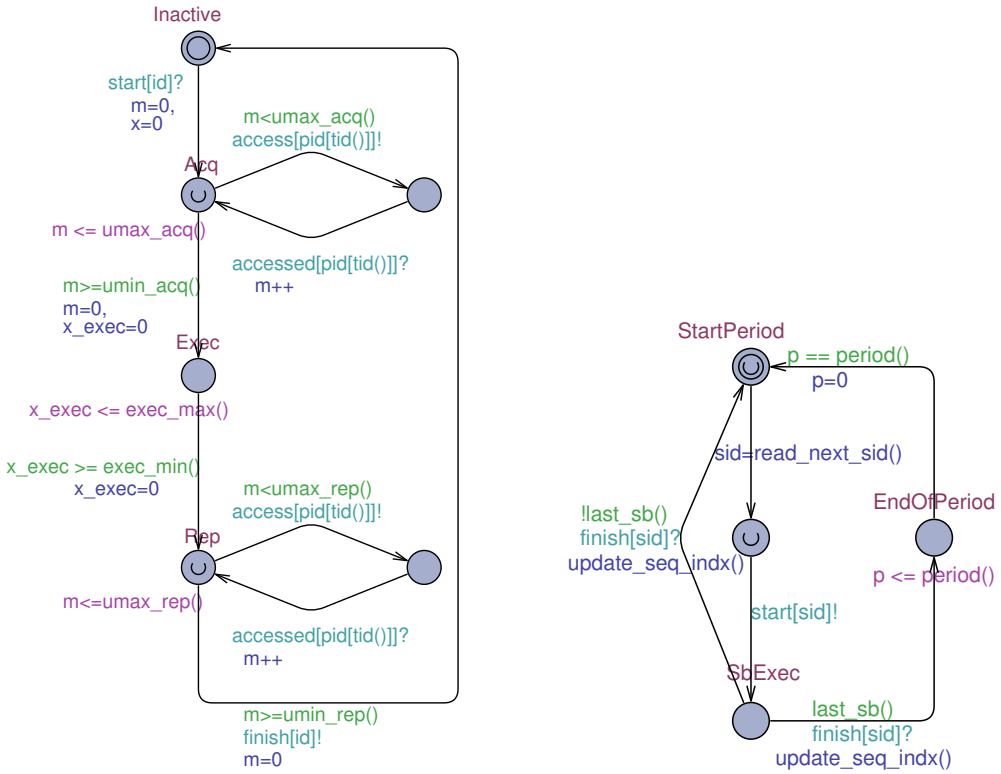
4.5 Worst-case Response Time Analysis using Model Checking

For the state-based analysis of resource contention scenarios in multicores, the system specification of Section 4.4 can be modelled by a network of cooperating timed automata (TA). This section presents the TA that were used to model the system and shows how we can derive tight WCRT bounds for each superblock and verify their safety using the Uppaal timed model checker. The next sections will then propose abstractions to reduce the required verification effort.

4.5.1 Modelling Concurrent Execution

The parallel execution of sequences of superblocks on the processing cores is modelled using instances of two TA, which are denoted as *Scheduler* and *Superblock* in the following. The TA in Uppaal notation are depicted in Figure 4.4. In a system with m cores and a total of n superblocks executed on them, $(n + m)$ TA instances are needed to fully model execution.

Each of the m instances of the *Scheduler* TA (Figure 4.4(b)) enforces the predefined execution order (schedule) of superblocks on its associated core p_j. Whenever a new superblock must be scheduled, the respective instance of TA *Scheduler* emits a `start[sid]` signal, with `start` being an array of channels and `sid` the index of the respective superblock. Due to the underlying composition mechanism, this yields a joint execution (synchronization) of the `start`-labelled edges by the respective instances of the TA *Scheduler* and *Superblock*. Similarly, when the superblock's execution is completed, the two instances of TA *Scheduler* and *Superblock* execute their `finish`-labelled edges, with `finish` being again an array of



(a) Superblock. (b) Static Scheduler.

Figure 4.4: Superblock and Scheduler TA.

channels. Once the last superblock in a processing cycle has terminated (`last_sb()`), the instance of TA *Scheduler* moves to location `EndOfPeriod`, where it resides until the end of the processing cycle (function `period()` returns value T_j). T_j time units after the activation of the first superblock of core p_j, the *Scheduler* TA triggers a new execution sequence of superblocks S_j. For simplicity, the depicted TA does not model the initial phase $\rho_{i,j}$ of S_j (it is assumed zero). However, this can be easily modelled by enforcing the TA to reside in location `StartPeriod` for $\rho_{i,j}$ time units before triggering a new execution of $s_{1,j}$.

A *Superblock* TA (Figure 4.4(a)) models the three phases of each superblock and is parameterized by the lower and upper bounds on access requests and computation times. Once a *Superblock* instance is activated, it enters the `Acq` location, where the respective TA resides until a non-deterministically selected number of resource accesses (within the specified bounds) has been issued and served. Access requests are issued through channel `access[pid]`, whereas notification of their completion is received by the arbiter through channel `accessed[pid]` (see 'loop transitions' over `Acq` in Figure 4.4(a)). For location `Acq`, we use Uppaal's concept of urgent locations to ensure that no computation time passes between successive requests from the same core, which complies with the

specification of our system model. Subsequently, the *Superblock* TA moves to the `Exec` location, where computation (without resource accesses) is performed. The clock `x_exec` measures here the elapsed computation time to ensure that the superblock's upper and lower bounds, e^{max} and e^{min}, are guarded. The behavior of the TA in the following location `Rep` is identical to that modelled with location `Acq` (successive resource accesses).

For the case of a single superblock in S_j, clock `x` is used to measure its total execution time. Checking the maximum value of clock `x` while the TA is *not* in its `Inactive` location allows to obtain the WCRT of the whole superblock. With Uppaal this is done by specifying the lowest value of `WCRT` for which the safety property:

```
A[] Superblock(i).Rep imply Superblock(i).x <= WCRT
```

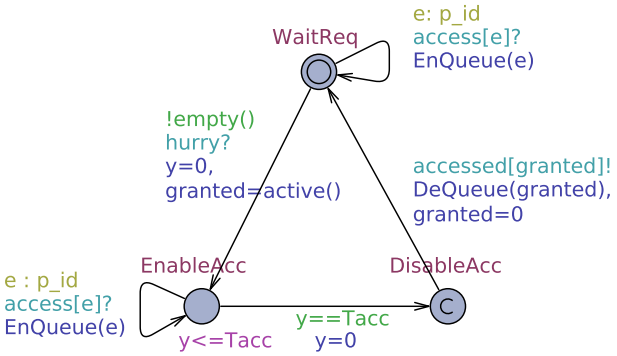
holds[2]. The property implies that location `Rep` is never traversed with `x` showing a clock value larger than `WCRT`. This way we ensure that for all reachable states, the value of superblock s_i's clock `x` is bounded by `WCRT` (safety). To find the lowest `WCRT` satisfying the previous property (tightness), binary search can be applied. Upon termination, the binary search will deliver a safe and tight bound on the superblock's WCRT. Similarly, we can compute a WCRT bound on a whole sequence S_j by adapting the TA of Figure 4.4(a) to model more than 3 phases.

4.5.2 Modelling Resource Arbitration

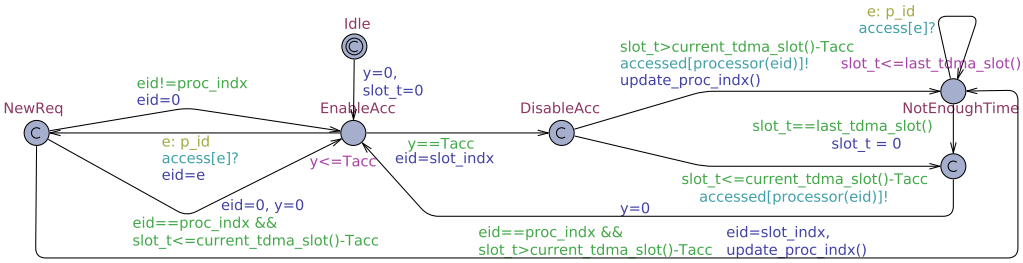
The TA models for the four suggested arbitration policies of Section 4.4.2 are depicted in Uppaal notation in Figure 4.5. Depending on the implemented policy, the respective model is included in the TA network of our system.

The *FCFS* and the *RR Arbiter* share a similar TA, depicted in Figure 4.5(a). Both arbiters maintain a queue with the identifiers of the cores that have a pending access request. In the case of FCFS, this is a FIFO queue with capacity m, since each core can have at maximum one pending request at a time. When a new request arrives, the arbiter identifies its source, i.e., the emitting core, and appends the respective identifier to the tail of the FIFO queue. If the queue is not empty, the arbiter enables access to the shared resource for the oldest request (`active()` returns the queue's head). After time T_{acc}, the access is completed, so the arbiter removes the head of the FIFO queue and notifies the respective *Superblock* instance that the pending request has been processed. For the RR arbiter a bitmap is maintained instead of a FIFO queue. Each position of it corresponds to one of the cores and pending requests are flags with the respective bit set to 1. As long as at least one bit is set, the arbiter parses the bitmap

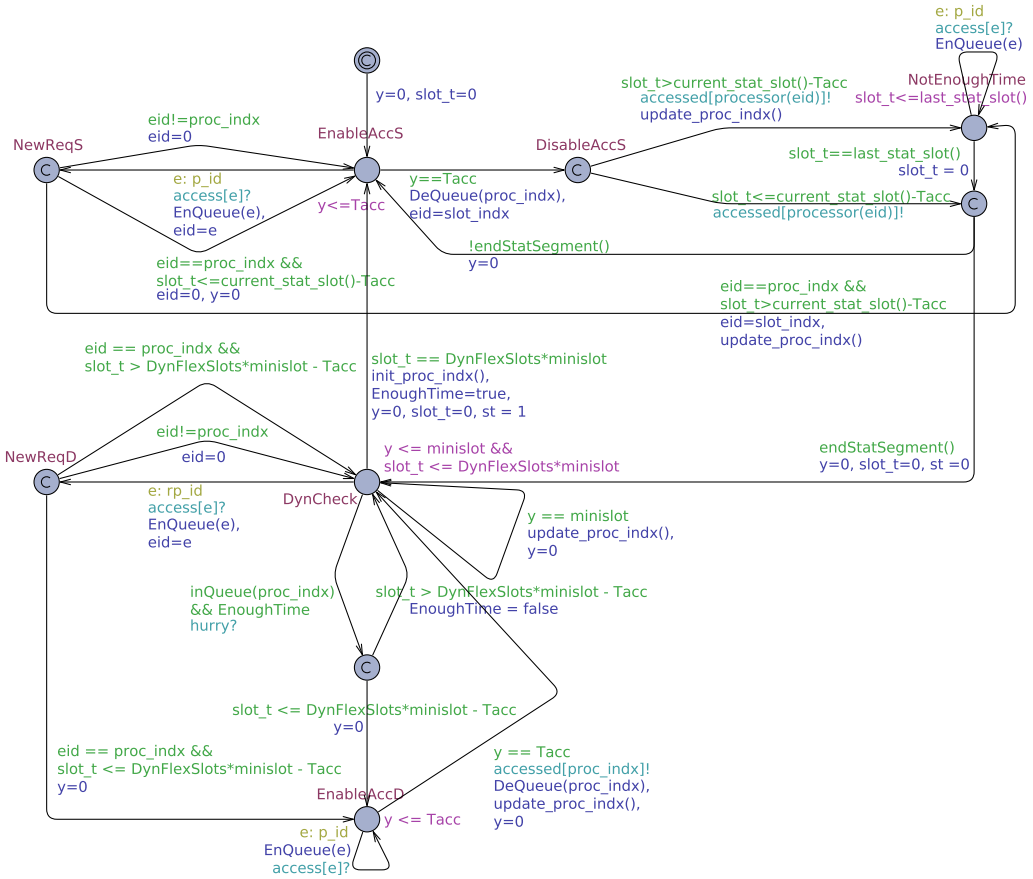
[2]Verification query `sup{Superblock(i).Rep}:Superblock(i).x` could be also used for the same purpose.



(a) FCFS / RR Arbiter.



(b) TDMA Arbiter.



(c) FlexRay Arbiter.

Figure 4.5: Arbiter TA representating different arbitration mechanisms.

sequentially granting access to the first request it encounters (return value of `active()`).

The *TDMA Arbiter* of Figure 4.5(b) implements the predefined TDMA arbitration cycle, in which each core has one or more, sequential or randomly assigned time slots. It is assumed that the cores (*Scheduler* instances) and the arbiter initialize simultaneously such that the first slot on the shared resource and the first superblock on each core start at time 0 (assuming synchronized processing cycles among the cores). The arbiter's clock `slot_t` measures the elapsed time since the beginning of each TDMA slot. When `slot_t` becomes equal to the duration of the current slot, the clock is reset and a new time slot begins. According to this, a new access request from core `eid` is served as soon as it arrives at the arbiter on condition that (i) the current slot is assigned to `eid` and (ii) the remaining time before the slot expires is sufficiently large for the processing of the access. If at least one condition is not fulfilled, the pending request remains in the arbiter's queue and is granted as soon as the next dedicated slot to `eid` begins.

The *FlexRay Arbiter* in Figure 4.5(c) is substantially an extension over the TDMA arbiter. This extension models the DYN segment of the FlexRay arbitration cycle that is executed after each ST segment. Once the ST segment is completed (`EndStatSegment()`), the arbiter checks if the core assigned to the first DYN minislot has a pending request. If this is true (`inQueue(proc_indx)`), the DYN minislot is resized to `Tacc` time units to accommodate the access. Otherwise, the arbiter waits until expiration of the minislot and then, it checks for pending requests from the core assigned to the next minislot. This procedure is repeated until the DYN segment expires. According to this model, during the FlexRay DYN segment, a new access request from core `eid` is served immediately on condition that (i) the current minislot is assigned to `eid` and (ii) the remaining time until the DYN segment expires is at least equal to T_{acc}, so that the DYN segment cannot interfere with the upcoming ST segment. If a condition is not fulfilled, then serving `eid`'s access request is postponed until its following ST or DYN (mini)slot.

In all *Arbiter* TA, new access requests can be received any time, either when the queue is empty or while the resource is being accessed. Multiple requests can also arrive simultaneously.

4.6 Reducing Complexity of Model Checking through Analytic Abstractions

Based on the results of Section 4.5, modelling a resource-sharing multicore requires a network consisting of *m Scheduler*, *n Superblock* and one *Arbiter* TA instances. By verifying appropriate temporal properties in Uppaal,

we can derive WCRT estimates for each superblock (sequence) that is executed on a processing core. However, scaling is related to the number of TA instances as the verification effort of the model checker depends on the number of clocks and clock constants used in the overall model. Particularly, the more the processing cores, the more the required clocks for modelling execution on them, which leads gradually to state space explosion. In this section, we propose safe abstractions for achieving a better analysis scalability.

In the proposed abstractions, only one processing core (core under analysis, CUA) is considered at a time, while all remaining cores which compete against it for access to the shared resource are abstracted away (but not ignored). To model the access requests of abstracted cores, we use arrival curves as defined in the real-time calculus (Section 4.3.2). This way an arrival curve capturing the aggregate interference pattern of the abstracted cores can be computed and then, modelled using TA. Eventually, the network of TA that models our system will include only one *Scheduler*, $|S_i|$ *Superblock* (number of superblocks executing on CUA p_i), one *Arbiter* and two *Request Generator* TA, i.e., the number of TA instances will not depend on the number of abstracted cores.

In the following, Section 4.6.1 presents how to derive an event arrival curve which bounds the resource accesses that are issued by a sequence of superblocks on a single core. Section 4.6.2 shows how to derive an aggregate interference arrival curve based on the individual access request arrival curves of several cores. Finally, Section 4.6.3 shows how to integrate such an interference curve into the TA-based system model.

4.6.1 Abstract Representation of Access Request Patterns

We consider a sequence of dedicated superblocks S_j, executing on processing core p_j with period T_j and accessing a shared resource. The possible resource access patterns depend on the minimum and maximum number of access requests, $\mu_{i,j}^{min}$ and $\mu_{i,j}^{max}$ (sum for acquisition and replication phase) and the minimum and maximum computation time, $e_{i,j}^{min}$ and $e_{i,j}^{max}$ (execution phase) of each superblock $s_{i,j} \in S_j$ as well as the order of superblocks in the sequence. In the following, we introduce a method to represent abstractly the possible access patterns as an upper arrival curve. The latter will provide an upper bound on the number of access requests that can be issued by the corresponding core in any interval of time.

The arrival curve for a core p_j is derived assuming no interference on the shared resource. Namely, the superblock sequence of p_j is analyzed in *isolation*, as if it had exclusive access to the resource. This is an over-approximation as it maximizes the issued resource access requests in the time interval domain. The computation of the arrival curve is performed as follows.

Acquisition $\{\mu^{a,min}, \mu^{a,max}\}$	{3,4}
Execution $\{e^{min}, e^{max}\}$	{50,70}
Replication $\{\mu^{r,min}, \mu^{r,max}\}$	{1,2}
Period T	250
Access Latency T_{acc}	20

Table 4.1: Superblock and shared resource parameters.

4.6.1.1 Computation of an Upper Access Request Trace

In the time domain, different access request traces R_j can be derived for core p_j if one considers all possible numbers of accesses and computation times within the lower/upper bounds of the superblocks in S_j. Note that since p_j is examined in isolation, the inter-arrival time between any two successive accesses issued by the core is equal to the resource access latency, which we denote T_{acc}. That is, every time p_j emits an access request, the access is assumed to be granted immediately, hence being served within T_{acc} time units. Among all possible access request traces, we identify the upper trace R_j^u which is computed by considering the *maximum* number of accesses and the *minimum* computation time for all superblocks of S_j. This trace corresponds to the scenario in which core p_j issues as *many* requests as possible (based on the memory access demand of S_j) as *early* as possible.

Lemma 4.1. *Any feasible access request trace R_j of core p_j satisfies the inequality:*

$$R_j(t) \leq R_j^u(t),\ \forall t \in [0, \infty)\ . \tag{4.3}$$

Proof. Consider a single execution of superblock sequence S_j in the interval $[0, T_j)$ and any time instant $0 \leq t < T_j$. Let trace point $R_j^u(t)$ specify the total number of access requests up to a certain execution point of a superblock $s_{k,j} \in S_j$. Given the deterministic computation method of $R_j^u(t)$ there can be only one such superblock (if at time t all superblocks have been executed, we take $k = |S_j|$). First, we consider the acquisition and replication phases of all superblocks preceding (and incl. up to t) $s_{k,j}$ in sequence S_j. If we compute a trace R_j with less than the maximum access requests for at least one acquisition or replication phase of these superblocks, then R_j will represent less or equal access requests in $[0, t]$ compared to R_j^u. Second, we consider the execution phases of all superblocks preceding (and incl. up to t) $s_{k,j}$. If we compute a trace R_j with a greater than the minimum computation time for at least one execution phase of these superblocks, then emission of $R_j^u(t)$ access requests can be delayed in time, i.e., $R_j^u(t) = R_j(t + \lambda)$ with $\lambda \geq 0$. Thus, we conclude that no feasible access request trace R_j for p_j can represent more accesses than R_j^u in time interval $[0, t]$, where $t \in [0, T_j)$. Since $R_j^u(t)$ is computed periodically, the same argumentation holds for subsequent periods $[T_j, 2T_j), [2T_j, 3T_j)$, namely inequality (4.3) holds $\forall t \in [0, \infty)$. □

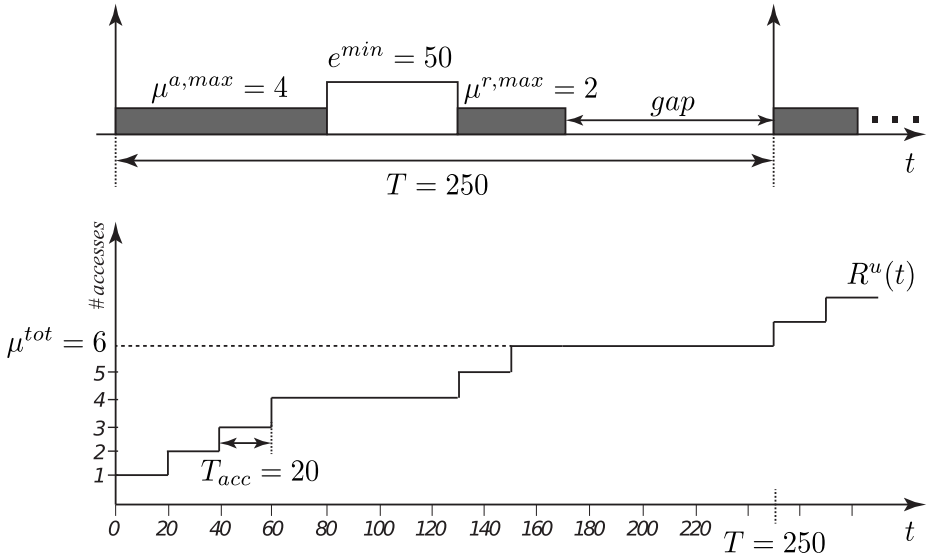


Figure 4.6: Upper access request trace derived from the superblock parameters specified in Table 4.1.

Example 4.2. *To illustrate the construction of R^u_j with an example, we consider a single superblock as specified in Table 4.1. If the superblock is solely executed on a core, then the trace R^u of access requests that this core can emit is given in Figure 4.6, where μ^{tot} is the total maximum number of accesses in one period, i.e., in this example, $\mu^{tot} = \mu^{a,max} + \mu^{r,max}$. In particular, in Figure 4.6, in the interval* $[0, 80)$ *R^u makes* 4 *steps which reflect the maximum number of accesses in the acquisition phase of the superblock. Each access takes the same access latency of $T_{acc} = 20$ which corresponds to the flat segments of length* 20 *after each step in R^u. These acquisition phase accesses are followed by the minimum execution phase of length* 50 *which corresponds to the flat segment in R^u in the interval* $[80, 130)$*. In the interval* $[130, 170)$*, R^u makes* 2 *steps which correspond to the maximum number of accesses in the replication phase. They are followed by a flat segment for the interval* $[170, 250)$ *which is denoted as the gap until the next execution of the superblock can start.*

Determining the length of this gap for the purpose of our analysis will be described next.

4.6.1.2 Lower Bounding the Gap Between Re-executions

In the computed upper access request trace, we can identify the *gap* (idle interval) between the end of the last phase of S_j and the start of the next processing cycle. In the example trace of Figure 4.6, this gap can be computed as $T - (\mu^{tot} \cdot T_{acc} + e^{min})$ as a result of the isolation assumption. However, on the actual multi-core system, where p_j competes against other cores for access to the shared resource, it is in fact possible that the incurred delays will cause the execution of S_j to be extended closer to the end of the processing cycle. Therefore, in the general case, where

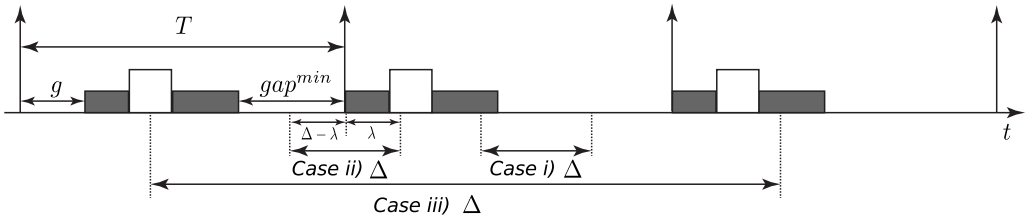


Figure 4.7: Three cases for the position of the considered interval Δ: i) within one processing cycle, ii) starting in one processing cycle and ending in the next processing cycle, iii) spanning more than two processing cycles.

no upper bounds on these delays can be provided, we need to consider a zero-interval as the minimum gap between two successive executions of S_j. Note that the gap cannot be negative because we assume that execution of a superblock sequence always finishes within the current processing cycle.

This estimation can be refined in particular cases. For instance, when the resource is FCFS or RR-arbitrated, in the worst-case every access of S_j can be delayed by all $(m-1)$ competing cores. Since access requests are blocking, every core can have up to one pending access request at a time. Therefore, each access request of S_j can be delayed by at most $(m-1) \cdot T_{acc}$ time units due to the interference of the other cores. As a result, a lower bound on the gap can be computed as follows:

$$gap^{min} = T_j - (m \cdot T_{acc} \cdot \sum_{\forall s_{i,j} \in S_j} (\mu_{i,j}^{a,max} + \mu_{i,j}^{r,max}) + \sum_{\forall s_{i,j} \in S_j} e_{i,j}^{max}) \,. \tag{4.4}$$

4.6.1.3 Deriving the Access Request Arrival Curve

Derivation of p_j's access request arrival curve follows from the computed upper trace R_j^u and the lower bound on the gap gap^{min}. Before we proceed, we need to define the shift operator $\triangleright$ as:

$$(R \triangleright g)(t) = \begin{cases} R(t-g) & , t > \max(g,0) \\ 0 & , 0 \le t \le \max(g,0) \end{cases} \tag{4.5}$$

The arrival curve that will be derived upper bounds the amount of access requests that p_j can emit in *any* time interval Δ. To safely obtain this bound, one has to consider time intervals $[t-\Delta, t)$, with $\Delta \ge 0$ and $t \ge \Delta$, which may start any time after the first triggering of the superblock sequence S_j. Depending on the number of processing cycles over which the intervals may span, we differentiate three cases for the start and the end of interval Δ, as illustrated in Figure 4.7.

Case i) $\mathbf{0 \le t - \Delta \le t \le T_j}$. Here, the intervals $[t-\Delta, t)$ are contained entirely in one processing cycle, as depicted in Figure 4.7. The number of access requests in such an interval is computed simply

as: $R_j(t) - R_j(t-\Delta)$. An upper bound on the access requests of this interval is computed by considering all possible positions for the interval Δ on the upper access request trace R_j^u and taking the maximum as follows:

$$\alpha_{j,i}^u(\Delta) = \max_{t \geq 0} \left\{ R_j^u(t) - R_j^u(t-\Delta) \right\} . \tag{4.6}$$

Lemma 4.2. *There is no feasible access request trace R_j of p_j such that: $R_j(t) - R_j(t-\Delta) > \alpha_{j,i}^u(\Delta)$, where $0 \leq t - \Delta \leq t \leq T_j$ and $\alpha_{j,i}^u$ is computed according to Eq. (4.6).*

Proof Sketch. Suppose that there exists an (adversary) feasible trace R_j, a time instant t_1 and an interval Δ, with $0 \leq t_1 \leq t_1 + \Delta \leq T_j$, such that $R_j(t_1 + \Delta) - R_j(t_1) > \alpha_{j,i}^u(\Delta)$ (reformulation for ease of notation). Let the superblock which is executed at time t_1 according to trace R_j be $s_{k,j}$. We assume that t_1 falls into an acquisition or replication phase of $s_{k,j}$, since $s_{k,j}$ issues no access requests during execution phase (the adversary strategy would not be able to maximize the number of request arrivals in Δ since by default there would be no accesses for at least a part of Δ). Suppose that at time t_1 superblock $s_{k,j}$ has $\hat{\mu}$ remaining access requests (not yet issued) in the current acquisition/replication phase according to trace R_j. Let, now, t_2 be the corresponding time instant at which superblock $s_{k,j}$ has $\hat{\mu}$ remaining access requests in the same phase according to the upper trace R_j^u. Note that t_2 is a well defined point because R_j^u represents at least equal accesses for each acquisition/replication phase as R_j. Hence, if $s_{k,j}$ has $\hat{\mu}$ remaining accesses in a phase according to R_j, a point describing the same situation at R_j^u will surely exist. The initial hypothesis implies that $R_j(t_1+\Delta) - R_j(t_1) > R_j^u(t_2+\Delta) - R_j^u(t_2)$ based on definition (4.6), i.e., R_j represents more access requests in $[t_1, t_1 + \Delta]$ than R_j^u in $[t_2, t_2 + \Delta]$. However, following a similar argumentation for interval $[t_2, t_2 + \Delta]$ as in Lemma 4.1 (where t_2 corresponds to 0 and $t_2 + \Delta$ to t in the proof of Lemma 4.1), we conclude that the above statement cannot be true, i.e., R_j cannot represent more access requests than R_j^u for the superblock phases included in $[t_2, t_2 + \Delta]$. This contradicts the initial hypothesis and proves the lemma. □

Lemma 4.2 simplifies the computation of $\alpha_{j,i}^u$, since this depends on a single access request trace, R_j^u.

Case ii) $\mathbf{0 \leq t - \Delta \leq T_j \leq t \leq 2T_j}$. In this case, the intervals $[t - \Delta, t)$ span over two processing cycles, as they start in one processing cycle and end in the next one. An example is shown in Figure 4.7. Again, the number of access requests in this interval is calculated based on $R_j(t) - R_j(t - \Delta)$.

Let us substitute $\lambda := t - T_j$. Then $R_j(t)$ can be expressed as $R_j(T_j) + R_j(\lambda)$. In order to obtain an upper bound, we use the upper access request trace: $R_j^u(T_j) + R_j^u(\lambda)$. The reason why considering only trace R_j^u is sufficient for deriving a safe upper bound is similar as in case i). The maximum total number of access requests in one processing cycle is a constant, calculated as $\mu_j^{tot,max} = \sum_{\forall s_{i,j} \in S_j} (\mu_{i,j}^{a,max} + \mu_{i,j}^{r,max})$, i.e., we have $R_j^u(T_j) = \mu_j^{tot,max}$. After the substitution, we can express $R_j(t-\Delta)$ as $R_j(T_j + \lambda - \Delta)$. In order to obtain an upper bound, we need to consider the minimum gap between the end of the superblock sequence in the first processing cycle and the start of the superblock sequence in the next processing cycle. For this purpose, we use the shifted to the right upper access request trace $R_j^u \triangleright g$ which takes into account the lower bound on the gap, gap^{min} from Eq. (4.4), where g is computed as follows:

$$g = T_j - (T_{acc} \cdot \sum_{\forall s_{i,j} \in S_j} (\mu_{i,j}^{a,max} + \mu_{i,j}^{r,max}) + \sum_{\forall s_{i,j} \in S_j} e_{i,j}^{min}) - gap^{min} . \tag{4.7}$$

Considering all possible positions of the interval Δ and taking the maximum, we obtain the arrival curve in this case:

$$\alpha_{j,ii}^u(\Delta) = \max_{0 \le \lambda \le \Delta} \left\{ \mu_j^{tot,max} + R_j^u(\lambda) - (R_j^u \triangleright g)(T_j + \lambda - \Delta) \right\} . \tag{4.8}$$

Case iii) $\mathbf{0 \le t - \Delta \le kT_j \le (k+1)T_j \le t}$, $\mathbf{k \ge 1}$. In this case, the intervals $[t - \Delta, t)$ may span over more than two processing cycles, as shown in Figure 4.7. We can observe that this case is similar to the previous one. However, the end of the interval Δ is not in the next processing cycle, but can be in later processing cycles. Therefore, knowing that the maximum number of access requests in one processing cycles is $\mu_j^{tot,max}$ and having K processing cycles between the start and the end of interval Δ, we can use the results from the previous case to obtain an arrival curve as follows:

$$\alpha_{j,iii}^u(\Delta) = \max_{1 \le K \le \left\lfloor \frac{\Delta}{T_j} \right\rfloor} \left\{ \alpha_{j,ii}^u(\Delta - K \cdot T_j) + K \cdot \mu_j^{max,tot} \right\} . \tag{4.9}$$

By combining the individual results of all three cases, we obtain the upper arrival curve α_j^u that upper bounds all access request traces of sequence S_j executing on core p_j for any time interval $\Delta \ge 0$. To this end, we take the maximum of expressions (4.6), (4.8), and (4.9):

$$\alpha_j^u(\Delta) = \max\{\alpha_{j,i}^u(\Delta), \alpha_{j,ii}^u(\Delta), \alpha_{j,iii}^u(\Delta)\} . \tag{4.10}$$

The arrival curve obtained by Eq. (4.10) is a safe upper bound on the access requests that can be issued by a core, but it is more accurate than

the upper bounds derived with earlier methods [PSC+10, GLST12]. The method presented in [PSC+10] introduces inaccuracy of the computation of the arrival curve because for simplicity it assumes zero inter-arrival time of access requests from the core. The method presented here improves on this by considering that the minimum inter-arrival time is bounded by the access latency T_{acc} of the shared resource. Similarly, the method presented in [GLST12] introduces inaccuracy because it considers that the minimum gap gap^{min} can appear between all successive executions of a superblock sequence, effectively shortening the processing cycle of the core, while the method presented here always considers the correct processing cycle.

4.6.2 Derivation of Interference Curve of Multiple Cores

Recall that for our analytic abstraction, we intend to use a single arrival curve to represent the access patterns of several processing cores, when analyzing the WCRT of a particular superblock sequence on a *core under analysis* (CUA). In this case, the interference caused by all the other cores is taken as the sum of their individual arrival curves, which are computed by Eq. (4.10) and by considering the cores in isolation. The sum represents a safe over-approximation of the interference that the cores may cause on the arbiter as we deal with monotone systems, where the number of requests received by the arbiter for any given interval Δ cannot exceed the sum of the issued requests by the interfering cores.

The sum of the arrival curves of all cores except the CUA is mentioned in the following as the interference curve α and it is computed as follows:

$$\alpha(\Delta) = \sum_{p_j \in \mathcal{P} \setminus \{p_{i(CUA)}\}} \alpha_j^u(\Delta)\,. \tag{4.11}$$

4.6.3 Embedding of Interference Curve into TA Model

For embedding the worst-case interference arrival curve α of the abstracted processing cores (Eq. (4.11)) into the TA-based system model, we exploit the results of Lampka et al. [LPT10]. The goal is to translate α into a meaningful set of TA that will model the emission of interfering access requests at such a rate so that α is never violated. In other words, we are looking for a state-based access request generator, capable of emitting all possible event streams that are upper bounded by α.

Initially, to reduce the complexity of the embedding, we over-approximate curve α by a single staircase function[3] $\alpha^{st}(\Delta) \geq \alpha(\Delta), \forall \Delta \in$

[3]Instead of a single staircase curve, α^{st} can also be composed of sets of staircase curves integrated via nested maximum and minimum operations[LPT10, PLT11]. This allows to model more complex curves, however it substantially adds to the complexity of the model checking problem to be solved when determining the WCRT of a superblock.

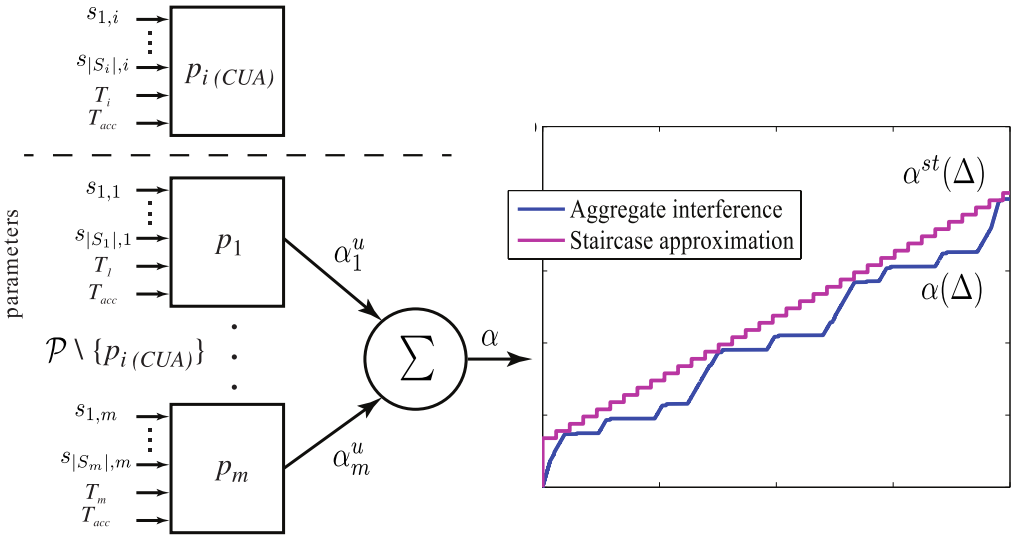
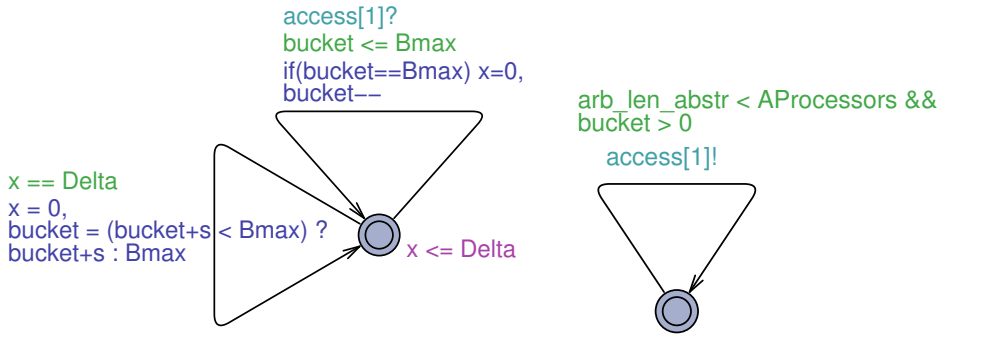


Figure 4.8: RTC interference arrival curve representation.

$[0, \infty)$, as defined in Eq. (4.2). Such a function is illustrated in Figure 4.8. The staircase function is selected so that (i) it coincides with the original α on the long-term rate and (ii) it has the minimum vertical distance to it. The event streams that are bounded by this new arrival curve can be generated by two TA, as depicted in Figure 4.9. These TA are adapted versions of the ones presented in [LPT10], so as to comply with our system specification. The *Upper Bound* TA (UTA) is responsible for guarding the upper staircase function α^{st} (with fixed parameters `Bmax`, `Delta` and `s` corresponding to B, δ, s of Eq. (4.2)), whereas the *Access Request Generator* (ARG) emits access requests "on behalf of" the abstracted cores on condition that UTA permits it.

UTA models partly what is known from the computer networks field as a leaky bucket. When the leaky bucket contains at least one token (unreleased access request), a corresponding event (request emission) can take place. If the leaky bucket is, however, empty, no requests can be emitted before new tokens are generated. The leaky bucket is configured so as to implement the upper staircase function α^{st}. Namely, it has a maximal capacity of B, being full in its initial state. An amount of s new tokens is produced every δ time units and one token is consumed every time a request is emitted by ARG.

Request emission by ARG is enabled as long as (i) at least one token is contained in the leaky bucket and (ii) the current amount of pending interfering requests is lower than the number of abstracted cores (to consider only realistic scenarios). If both conditions are valid, then ARG may issue a new request, without restriction on the time point when the latter occurs (to account for all traces below α^{st}). Note that in the TA of Figure 4.9, '1' refers to the default identifier of the access request generator as seen by the arbiter (representing without distinction any of



(a) Upper Bound (UTA).

(b) Access Request Generator (ARG).

Figure 4.9: Interference generating TA.

the abstracted cores' identifiers).

Note that in the new system model, where the two presented TA substitute all *Scheduler* and *Superblock* TA instances that were previously used to model execution and resource accessing on $m - 1$ cores, the total number of TA instances is independent of the number of cores in the system. To illustrate the reduction in the size of the TA-based system specification, consider a system with 32 cores, each executing a single superblock. If all cores were modelled individually, i.e., each by its own component TA as described in Section 4.5, the complete system model would consist of 65 TA: 32 instances of the TA *Scheduler*, 32 instances of the TA *Superblock* and one instance of the TA *Arbiter*. Therefore, the system model would contain in total 97 clock variables and 128 synchronization channels. By applying the above abstraction, one obtains a system model which contains five component TA only: one instance of the TA *Scheduler*, one instance of the TA *Superblock*, one instance of the TA *Arbiter* and one instance of the TA *Upper Bound* and *Access Request Generator*. In total, this yields a system model with 5 clocks and 6 synchronization channels.

Substitution of the $(m - 1)$ *Scheduler* and the $(n - |S_i|)$ *Superblock* TA instances of the abstracted cores with the presented pair of interference generating TA is expected to alleviate significantly the verification effort for the WCRT estimation of the superblocks executing on CUA (core p_i). This comes with the cost of over-approximation, since the interference arrival curve α provides a conservative upper bound of the interfering access requests in the time-interval domain and a^{st} over-approximates it. Additionally, the interference generating TA emit interfering requests non-deterministically in time, enabling the exploration of certain request streams that are bounded by α^{st} but may never occur in practice. As shown in Section 4.8, though, the pessimism in the WCRT estimates for the superblocks of CUA is limited.

4.7 Further Adaptations to Improve Scalability

Besides the basic abstraction step of Section 4.6, i.e., the interference representation with arrival curves and the modelling of the latter with TA, further abstractions and optimizations of our system specification can be considered. We briefly discuss these in the following, such that the experimental results of Section 4.8 can be reproduced:

1. For system models in which execution on all cores is modelled explicitly and the resource is FCFS or RR-arbitrated, the state space exploration for a superblock's WCRT can be restricted to the duration of one hyper-period of the cores' processing cycles. The hyper-period is defined as the least common multiple of the cycles' periods, $\text{lcm}(T_1, \cdots, T_m)$, and within its duration all possible interference patterns are exhibited. Therefore, deriving a superblock's WCRT by exploring the feasible scenarios in this time window only is safe. Note that a similar simplification can be applied in case of TDMA-arbitrated resources if the hyper-period is redefined to account for the period of the arbitration cycle, $\text{lcm}(T_1, \cdots, T_m, \Theta)$, where Θ denotes the TDMA arbitration cycle.

2. In system specifications where execution on some cores is abstracted and the resource is FCFS or RR-arbitrated, the TA *Superblock* can be simplified to model not periodic execution, but a single execution. Since all feasible interference streams bounded by α^{st} can be explored for the time interval of one superblock execution, the WCRT observed during this interval is a safe WCRT bound. This simplification also eliminates the need for including the *Scheduler* and the remaining *Superblock* instances of the CUA in the TA system model. The same can be applied to systems with a TDMA arbiter if we enumerate and model all possible offsets for the starting time of the superblock within the TDMA arbitration cycle.

3. To bound the WCRT of a superblock, one can add the WCRT of the individual superblock phases. Similarly as before, we can model single acquisition or replication phases and explore all interference streams bounded by α_{st} for the time interval of one phase execution. Based on the arrival curve properties, the obtained WCRT bound for each phase is safe. For the execution phase, one can simply consider the maximum execution time and add this to the previous sum. The total WCRT of the superblock (sum) of the individual phases can be more pessimistic than the WCRT found when the sequence of superblocks is analyzed in a single step. This is because it may not be possible for all phases to exhibit their WCRT in a single superblock execution. Nevertheless, this simplification reduces the verification effort, by dividing the original problem into smaller ones, each analyzed independently.

4. In system models with a TDMA resource arbiter, the interference from the competing cores can be ignored (not modelled) due to the timing isolation that a TDMA scheme offers. Namely, for deriving a superblock's WCRT, the model checker needs to consider all possible relative offsets between the arrival of the CUA's access requests and the beginning of its dedicated slot in the arbitration cycle. The interference from the remaining cores does not affect the superblock's WCRT. The same holds for the static segment of the FlexRay arbitration cycle.

5. In system models with a FCFS or RR resource arbiter, granularity of communication between the *Access Request Generator* and the *Arbiter* TA may become coarser, by letting access requests arrive in *bursts* at the arbiter. For this, the interference generating TA can emit requests in bursts of b, where b is a divider of the maximum burst size B in Eq. (4.2) and $b \leq m - 1$. The arbiter TA can be adapted to store the requests and serve them like in the original system (as if they were emitted one by one). New bursts of interfering requests can be generated any time after the previous b requests have been served. This optimization decreases the need for inter-automata synchronization and also the number of explored traces below α^{st}, since the inter-arrival times among the requests in a burst are no longer non-deterministic, but fixed to 0. The traces that are not explored could not cause worse response times for CUA's superblocks than the ones with the bursty arrivals. This is because, if some of the b interfering access requests arrived later (non-zero inter-arrival time), the next access of CUA would suffer equal or less delay compared to the bursty case. Specifically, if the "delayed" interfering requests arrived still before the next request of CUA (FCFS arbitration) or before the turn of CUA (RR arbitration), the delay for serving the latter's request would be the same as if the interfering requests had arrived in a burst. Otherwise, the "delayed" interference requests would not be served before CUA's request, thus reducing its response time. Therefore, the omission of the non-bursty traces has no effect on the correctness or tightness of the WCRT estimates. The same holds also for the dynamic segment of the FlexRay arbitration cycle.

6. The *Upper Bound* TA requires to reset its clock once *Access Request Generator* has emitted the maximal number of resource access requests at a single point in time, i.e., it has produced a burst of access requests. It is safe to omit this clock reset, since the *Access Request Generator* could simply emit more events than bounded by the original curve α^{st}. For instance, the TA *Access Request Generator* of Figure 4.9 could release `Bmax` events just before clock `x` expires (through successive traversals of edge `access[1]?`). Without

resetting clock `x` upon the `Bmax`-th traversal, one could actually release more access requests once clock `x` expires, i.e., `x == Delta` holds. This yields emissions of more requests than bounded by the original α^{st}, which in turn could introduce pessimism into the analysis. On the other hand, it can help with lowering the state space explosion, as it reduces the number of clock resets. In the case of our model, we cannot notice any additional pessimism w.r.t. the WCRT of the superblocks. That is because the maximum number of pending requests is not derived from `Bmax`, but from the number of cores in the system. Additional resource access requests can only be issued at the rate of service experienced at the shared resource.

4.8 Evaluation

This section presents two case studies, to which the proposed state-based WCRT analysis approach was applied in order to evaluate its *scalability* and the *accuracy* of the obtained results. We consider multi-core systems in which cores have private caches and shared access to a main memory. The systems are modelled either fully with TA (FTA, Section 4.5) or with a combination of TA and arrival curves (ATA, Section 4.6). Further modelling optimizations as discussed in Section 4.7 are applied whenever possible. In the first case study (Section 4.8.1), we model systems with two to six cores, on which a set of industrial benchmarks is executed. We evaluate the scalability of the FTA and ATA analysis methods and compare the respective WCRT estimates with results obtained from architectural simulations. In the second case study (Section 4.8.2), we model systems with two to 64 cores, on which an automotive application is executed. We evaluate the scalability limits of the ATA analysis method for different arbitration policies of the shared memory and we compare the WCRT estimates to those of state-of-the-art analytic approaches.

4.8.1 Case Study I: State-based Analysis vs Simulations

First, we consider systems with two to six cores, which access the main memory in a round-robin fashion when cache misses occur. We model such systems with the FTA and ATA formal approaches and compare the scalability and accuracy of the respective analyses. We also compare our derived WCRT bounds with results obtained from architectural simulations. While simulations by definition might fail to reveal the real worst-case scenario, they are nevertheless useful in validating the overall system settings and providing a lower bound on WCRT to compare with the safe upper bound provided by analysis.

We base our evaluation on a simulated multi-core platform using private LVL1 and LVL2 caches and shared main memory. The

Gem5 [BBB+11] architectural simulator is used to execute selected benchmarks and obtain superblock parameters for accesses to main memory. The simulator is configured as follows: in-order core based on x86 instruction set running at 1 GHz; split LVL1 cache of 32 kB for instruction and 64 kB for data; unified LVL2 cache of 2 MB; cache line size of 64 bytes; a constant time of 32 ns for each memory access. Section 4.8.1.1 provides more details on the evaluated benchmarks. Section 4.8.1.2 describes our multi-core simulation settings and presents comparative results for the simulation and the two suggested analysis methods. Finally, Section 4.8.1.3 evaluates the accuracy of our analysis by comparing WCRT estimates obtained with ATA to conservative WCRT bounds that can be derived (without model checking) for the RR arbitration policy.

4.8.1.1 Benchmarks and Determination of Superblock Parameters

To evaluate the proposed techniques, we considered six benchmarks from the AutoBench group of the EEMBC (Embedded Microprocessor Benchmark Consortium) [eem] and ported them to PREM[4] [PBB+11]. We examined benchmarks representing streaming applications that process batches of input data and produce a stream of corresponding output. Specifically, the six benchmarks we used from the AutoBench group were "a2times" (angle to time conversion), "canrdr" (response to remote CAN request), "tblook" (table lookup), "bitmnp" (bit manipulation), "cacheb" (cache buster) and "rspeed" (road speed calculation). Ideally more benchmarks would have been examined, however, making a benchmark PREM-compliant is a time-consuming operation. A similar approach to benchmarking was employed in past works using PREM, in particular [PBB+11, YPB+12, BYPC12].

Each benchmark in the EEMBC suite comes with a sample data file that represents typical input for the application. The benchmark comprises required initialization code, followed by a main processing loop. Each iteration of the benchmark algorithm processes a fixed amount of input data, and the number of iterations is selectable. We configured the number of iterations such that each benchmark processes its entire sample data. Then, we compiled the whole main loop into a superblock. This way every periodic execution of the resulting task consists of a single superblock. During the acquisition phase, the whole sample data of the benchmark, static data and code of the main loop are loaded into cache. In the execution phase, the main loop is repeated for the selected number of iterations. Finally, during the replication phase all output data are written back to main memory. Note that we did not include the initialization code of the benchmark in the superblock, since such code must be run only

[4]The application of the PREM framework for the derivation of the benchmark superblock parameters and the simulations with the Gem5 simulator consist joint work with Zheng Pei Wu and Rodolfo Pellizzoni.

Benchmark	Iterations	PREM				
		$\mu^{max,a}$	$\mu^{max,r}$	$e^{max,a}$	$e^{min,e}$	$e^{max,e}$
a2times	256	129	26	1561	215552	296448
canrdr	512	186	26	1821	110080	1047552
rspeed	256	90	23	1094	92160	163328
tblook	128	271	23	2453	103424	795648
bitmnp	64	170	47	1678	4669760	5173056
cacheb	32	100	33	1025	8704	11872

Table 4.2: Benchmark parameters.

once at the beginning of the benchmark and is not executed as part of a periodic task.

Table 4.2 provides the derived characterization for the six benchmarks run on our architectural simulator. To obtain valid measures for our superblock model, each benchmark was executed in isolation on one core, with no other computation in the system. We provide the number of iterations for each benchmark. We report the maximum number of accesses $\mu^{max,a}, \mu^{max,r}$ for the acquisition and replication phases, the maximum execution time $e^{max,a}$ (ns) for the acquisition phase, as well as minimum and maximum execution times $e^{min,e}, e^{max,e}$ (ns) for the execution phase. Note that in our simulations, the replication phase was implemented by flushing the cache content before the next superblock starts, hence the cache was empty at the beginning of each acquisition phase. Since furthermore the amount of processed and modified data is constant for a given number of benchmark iterations, we have $\mu^{min,a} = \mu^{max,a}$ and $\mu^{min,r} = \mu^{max,r}$, i.e., the number of accesses in the acquisition and replication phases is constant. Also note that the largest working set in Table 4.2 (see "tblook") is 271 cachelines $\times$ 64 bytes-per-cacheline = 17,344 bytes, which can fit in LVL2 cache for commonly used processors. The number of accesses during the execution phase is zero. The execution time during the acquisition phase is also constant and dependent on the number of instructions required to prefetch $\mu^{max,a}$ cache lines. We do not report the execution time of the replication phase since a single instruction can be used to flush the cache independently of its content. Finally, the minimum and maximum lengths of the execution phase depend on the input data, and were thus computed as follows: we first measured the minimum and maximum execution time for a single iteration of the benchmark. Then, we obtained $e^{min,e}, e^{max,e}$ by multiplying the number of benchmark iterations by the measured minimum and maximum per-iteration time, respectively. Since the provided sample data are designed to test all code paths in the algorithms, we believe that this way, we sufficiently approximated the minimum and maximum execution time bounds that would be computed by a static analysis tool. For an in-depth comparison of PREM versus normal (non-PREM) execution, we refer the

Benchmark	Period (ns)	Benchmark	Period (ns)
a2times	360000	tblook	900000
canrdr	1350000	bitmnp	5400000
rspeed	200000	cacheb	40000

Table 4.3: Benchmark periods for simulation.

interested reader to [PBB+11, YPB+12]. In general, the number of cache line fetches under PREM is slightly higher than the number of fetches under non-PREM execution, but this overhead is relatively low for most benchmarks.

4.8.1.2 Evaluation: FTA vs ATA vs Simulation

We simulated the parallel execution of two to six tasks on an equal number of cores, where each task is composed of a single superblock, based on one of the described benchmarks. We used simulations to provide a meaningful lower bound to the WCRT of each task considering contention for access to the shared resource and to validate our model. Our resource simulator uses traces of memory requests generated by running each benchmark in isolation on Gem5, as described earlier in Section 4.8.1.1. The simulator keeps track of simultaneous requests by multiple cores and arbitrates accesses to the shared resource based on round-robin arbitration. Since we assume an in-order model where the core is stalled while waiting for main memory, the simulator accounts for the delay suffered by each memory access by delaying by an equivalent amount all successive memory requests performed by the same core.

For each scenario, we simulated a synchronous system, i.e., all tasks were activated for the first time simultaneously. Each task was then executed periodically with a given period, shown in Table 4.3, on its dedicated core. The task periods were selected so that each task can complete execution within them. Namely, a superblock's period is at least equal to its conservative WCRT estimate:

$$WCRT_{cons} = (\mu^{max,a} + \mu^{max,r}) \cdot m \cdot T_{acc} + e^{max,a} + e^{max,e} \ , \qquad (4.12)$$

which assumes that every access experiences the worst possible delay, i.e, $m \cdot T_{acc}$ under RR resource arbitration (e.g., for $m = 6$ and $T_{acc} = 32$ns, we have $WCRT_{cons} = 327769$ ns for benchmark "a2times"). In both the simulations and the analytical model, we allowed the execution time of the execution phase to vary between $e^{\min,e}$ and $e^{\max,e}$. To enable this variability, we decided to simulate each scenario until the task with the largest period has executed 2000 times, and we recorded the maximum response time for each task during the simulation; the length of the execution phase of each job was randomly selected based on a uniform distribution in $[e^{\min,e}, e^{\max,e}]$. For each scenario, we also applied

the proposed WCRT analysis methods, first the one where the system is *fully* modelled with TA (FTA) and then, the one where a part of the system (interfering cores) is *abstractly* modelled with arrival curves (ATA). The additional abstractions of Section 4.7 were also applied whenever possible. Table 4.4 presents the WCRT of each task as observed during simulation as well as the difference among this value and the corresponding WCRT with the two analysis methods. The difference is defined for each task as:

$$Difference = 100 \cdot \frac{WCRT_{analytic} - WCRT_{simulation}}{WCRT_{simulation}} . \tag{4.13}$$

Table 4.4 presents also the time required to verify one WCRT query with the Uppaal timed model checker in each case. Note that the total verification time will be a multiple of the presented time because of the binary search performed to specify a tight response time bound. All verifications were performed with Uppaal v.4.1.7 on a system with an Intel Xeon CPU @2.90 GHz and 128 GB RAM. Note that the experimental results should be reproducible also on machines with a lower RAM capacity (e.g., 8 GB). In particular, the model checker required up to 4.3 GB RAM for the FTA analysis, for systems with 2-3 cores. For the WCRT analysis of benchmarks "a2times" and "rspeed" in the 4-core scenario, the RAM utilization surpassed 25 GB. This is the only case where model checking for the FTA method could fail to complete on a machine with restricted RAM capacity. Respectively, model checking for the ATA analysis required several tenths of MB in most cases. The peak RAM consumption was observed when verifying the WCRT of benchmark "bitmnp" in the 6-core scenario and was equal to 1.2 GB.

The FTA analysis method could be applied to systems with up to 4 cores. For tasks "canrdr" and "tblook" in the 4-core scenario as well as for the 5-core and 6-core scenarios, verification of a WCRT query with Uppaal required more than 120 hours and was, thus, aborted. Nevertheless, for the scenarios where the FTA method could be applied, the obtained WCRT results are very close to the ones observed during simulation. The maximum deviation between the corresponding values is **0.85%**, confirming that our formal modelling is a good reflection of the dedicated superblock (PREM) execution. This can be explained as follows: PREM yields phase structured executables, respectively compiled code, and we used this code with the simulator. It can actually be expected that higher timing determinism allows the upper WCRT bounding to be close to the measured results of the simulation. This has also as a consequence that the comparison of ATA and simulation is almost as decisive as the direct comparison between ATA and FTA. This is particularly important for cases where the FTA WCRT estimates cannot be obtained, e.g., because the analysis runs out of memory.

Note that the scalability of FTA is already improved compared to

Cores	Benchmark Set	Simulation	FTA		ATA	
		WCRT (ns)	Diff.(%)	Verif. time	Diff.(%)	Verif. time (sec)
2	a2times	305540	0.28	2.4 sec	0.78	1.5
	canrdr	1058020	0.09	2.5 sec	0.29	1.9
3	a2times	308431	0.51	16 min	1.44	2.7
	canrdr	1060294	0.15	16.25 min	0.43	3.3
	rspeed	172712	0.45	15.25 min	1.48	1.8
4	a2times	312839	0.64	79.9 hrs	1.60	3.6
	canrdr	1066062	-	-	0.78	5.2
	rspeed	175588	0.85	12.2 hrs	1.88	2.2
	tblook	819105	-	-	0.44	7.0
5	a2times	315704	-	-	2.25	4.6
	canrdr	1068112	-	-	1.42	6.6
	rspeed	178424	-	-	2.29	2.8
	tblook	822330	-	-	1.13	10.8
	cacheb	28666	-	-	19.22	4.0
6	a2times	319802	-	-	2.49	5.0
	canrdr	1074540	-	-	1.45	7.2
	rspeed	181249	-	-	2.69	3.4
	tblook	827793	-	-	1.43	14.4
	cacheb	32251	-	-	19.17	4.4
	bitmnp	5202608	-	-	0.26	11.1

Table 4.4: WCRT results of EEMBC benchmarks: FTA vs. ATA vs. simulation for RR arbitration.

earlier model checking-based analysis methods, e.g., [GELP10, LYGY10], in which analysis did not scale efficiently beyond two cores for event-driven resource arbitration schemes. This improvement can be attributed to our first proposed abstraction, namely the dedicated superblock structure of the tasks.

On the other hand, the ATA analysis method scales efficiently, with the verification of each WCRT query being completed in few seconds in all cases, almost independently of the number of cores in the system. One can observe that the obtained WCRT results are slightly more pessimistic than the ones derived with FTA, as the differences to the simulation observations are now larger. However, the pessimism of ATA compared to FTA is limited, reaching at maximum **1.03%** for task "rspeed" in the 3-core and the 4-core scenarios.

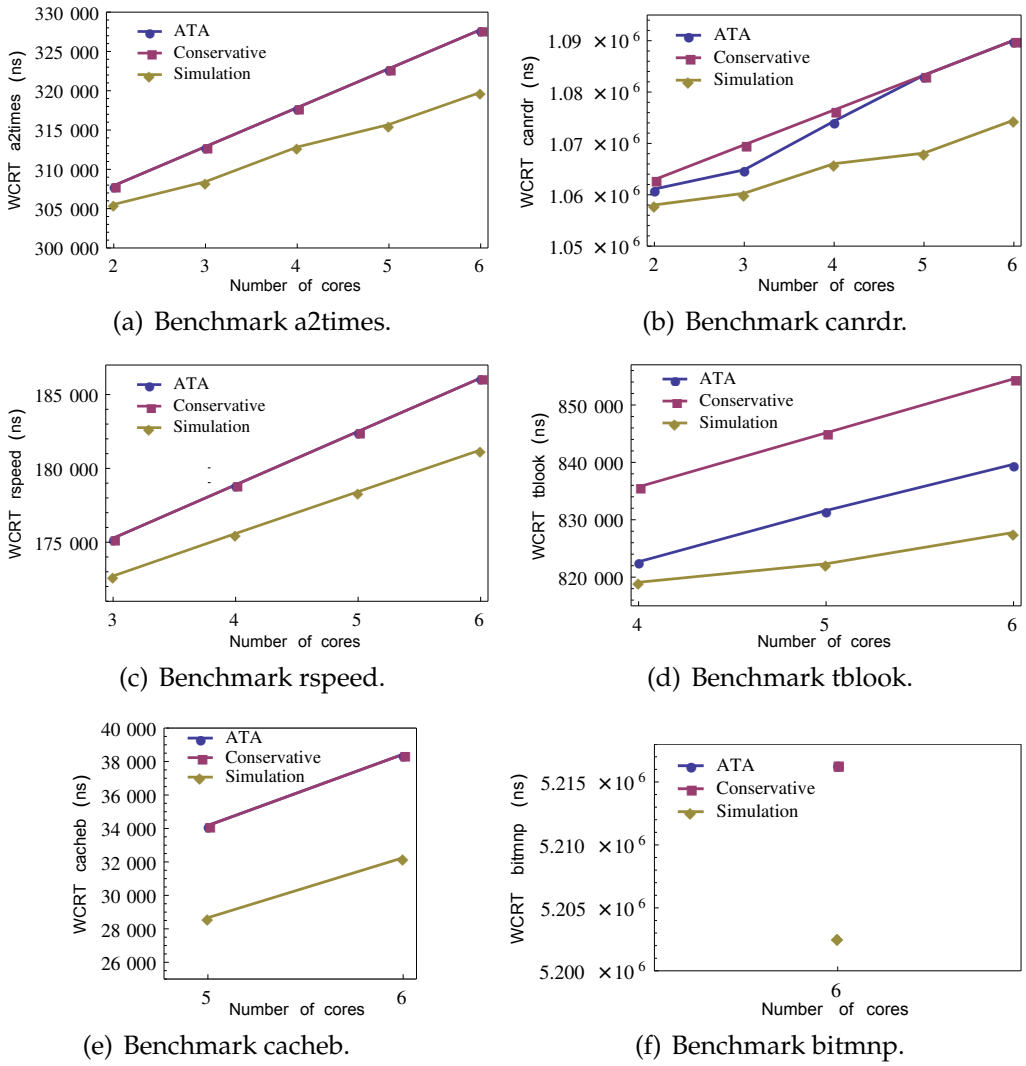
The small deviation between the FTA and ATA estimates can be explained as follows: the two methods produce the same WCRT for a superblock if the system under analysis operates close to the conservative case. For instance, for RR arbitration of a shared memory, this happens when each core uses its assigned slot for accessing the resource while an access of the core under analysis is pending. Likewise for FCFS-based resource arbitration, this happens when the FIFO buffer is always filled with $m - 1$ requests upon the arrival of a request from the core under analysis; assuming that there are m cores in the system. It appears that

with the considered benchmarks most of the time the system shows such a behavior and this is why both methods produce comparable results. In the general case, if the real-time system exhibits a smaller overlap of the access requests from different cores, the ATA method can become more pessimistic. This is due to (i) the construction of the interference curve α (sum of individual curves, Section 4.6.2), which provides a conservative upper bound on the interfering access requests, (ii) its over-approximation by the staircase function α^{st}, and (iii) the behavior of the access request generator (Section 4.6.3), which emits interfering requests non-deterministically over time, thus enabling the exploration of certain request streams that are bounded by α^{st} but may never be encountered at runtime.

The maximum difference between the ATA WCRT and the corresponding simulation-based WCRT is observed for task "cacheb" in the 5-core scenario and is equal to **19.22%**. What differentiates "cacheb" from other benchmarks and contributes to this high difference is the fact that the access-to-computation time ratio is considerably high for this benchmark. If we look at the maximum bounds from Table 4.2, resource accessing accounts for 25% of the total execution time when "cacheb" is executed in isolation. This is why the effects of a pessimistic analysis method affect the WCRT of "cacheb" more than other less memory-intensive benchmarks. As pointed out above, a similar picture could be expected also when comparing the FTA and ATA method, as FTA and simulation produce similar WCRT due to the code structuring. The results of Table 4.4 show, however, also that the difference (pessimism) is limited since in most cases, the ATA-derived WCRT are only up to **2.5%** greater than the corresponding simulation results. This allows us to conclude that the gain in scalability, obtained by the abstraction of a part of the system with arrival curves, does not compromise the accuracy of the WCRT. The topic of accuracy is discussed further in the next section.

4.8.1.3 Evaluation: ATA vs Conservative WCRT Estimation

Figures 4.10(a)-4.10(f) illustrate the absolute WCRT estimates of each EEMBC benchmark for each scenario (different number of concurrently executed benchmarks), as obtained by (i) the ATA analysis methodology, (ii) the simulation environment presented in Section 4.8.1.2, and (iii) a conservative WCRT estimation under RR resource arbitration, given by Eq. (4.12). The results of the first two methods are the same as in Table 4.4. The last estimation assumes that every single access of a core under analysis is delayed by *all* interfering cores in the system. This conservative, yet safe assumption, which was also the basis of the WCRT analysis in Chapter 3, enables modelling RR through a TDMA scheme, where each core has one slot of length equal to T_{acc} (access latency) in every arbitration cycle. Independently of whether the interfering cores



(a) Benchmark a2times.

(b) Benchmark canrdr.

(c) Benchmark rspeed.

(d) Benchmark tblook.

(e) Benchmark cacheb.

(f) Benchmark bitmnp.

Figure 4.10: WCRT of EEMBC benchmarks: ATA vs. Conservative Bound vs. Simulation for RR arbitration.

emit accesses within a cycle, the corresponding slots cannot be used by the core under analysis, which has to wait for a whole cycle interval between any two successive accesses.

Based on the depicted results, the difference between the ATA estimates and the observed through simulation WCRT varies between 0.26% ("bitmnp", 6 cores) and 19.22% ("cacheb", 5 cores). Respectively, the difference between the ATA and the conservative WCRT estimates varies between 0% (benchmarks "a2times", "rspeed", "cacheb", "bitmnp") and 1.8% ("tblook", 6 cores). The equality of the ATA and the conservative estimates for some benchmarks is a special case stemming from the nature of the considered benchmarks. Particularly, the six benchmarks (i) exhibit similar structure with a burst of hundreds of access requests at the beginning of their execution and (ii) start synchronously (at time 0) among

all cores. This results in massive interference on the shared memory at the beginning of each hyper-period, where the memory accessing (acquisition) phases of the benchmarks overlap. Consequently, for the considered case study the exhibited interference at runtime is closely described by the conservative assumption mentioned above. A larger deviation between the two estimates would be expected if the benchmarks started their execution after given phases, so that their acquisition phases would only partially (or not at all) overlap.

In general, for more complex arbitration policies than RR, like FlexRay, the derivation of a conservative bound is not trivial and can be only based on overly pessimistic assumptions (e.g., no access provided during the dynamic FlexRay segment) due to the complexity of modelling the state of the arbiter. Under such arbitration scenarios, we expect the ATA WCRT to be more accurate than any conservative estimates. The same applies also to cases, where the initial phases of the benchmarks (here, 0) are synchronized such that the interference on the memory path is reduced. If the interference arrival curves are computed by considering the initial phases, the ATA WCRT will be refined as opposed to the conservative bounds which do not reflect this information.

4.8.2 Case Study II: ATA vs Analytic Approaches

In the second case study, we explore the scalability limits of our WCRT analysis approach when resource accesses of some of the processing cores are abstracted with an arrival curve, i.e., for the ATA approach[5]. We also evaluate the accuracy of ATA for different arbitration schemes w.r.t. state-of-the-art analytic methods.

For this purpose, we used a real-world application provided by an automotive supplier, which consists of 4 independent tasks. Each task is defined as a sequence of 2-8 general superblocks (resource accesses can occur also during execution phases), of which the access requests and computation parameters were derived with static analysis techniques. The superblock parameters cannot be disclosed due to confidentiality restrictions.

We consider systems with 2, 4, 8, 16, 24, 32 or 64 cores and a shared memory that can be arbitrated according to any of the policies addressed in this chapter. Each of the application tasks is assumed to be executed periodically on one core (for systems with more than 4 cores, tasks are replicated), and depending on the arbitration policy of the memory controller, we make the following additional assumptions:

- When the shared memory arbitration is FCFS or RR, the processing cycles of different cores are considered non-synchronized, i.e.,

[5]For the derivation of the access request arrival curve of each core, we used here the method presented in [GLST12], which is more conservative compared to the method of Section 4.6.1.

Cores	FCFS		RR		TDMA	FlexRay	
	FTA	ATA	FTA	ATA	FTA/ATA	FTA	ATA
2	1.2 sec	1 sec	1.2 sec	0.5 sec	0.1 sec	0.7 sec	27.1 sec
4	-	1.7 sec	-	1.1 sec	0.2 sec	0.9 sec	28.5 sec
8	-	31.3 sec	-	1.2 sec	0.5 sec	2.8 sec	17.8 sec
16	-	4.9 min	-	1.3 sec	0.9 sec	-	10.4 sec
24	-	18.8 min	-	1.6 sec	1.1 sec	-	26.8 sec
32	-	-	-	1.9 sec	1.2 sec	-	57.4 sec
64	-	-	-	2.7 sec	1.7 sec	-	7 min

Table 4.5: Verification time for safety property regarding a superblock's WCRT.

execution of superblock $s_{1,j}$ can start at any time within $[0, T_j)$. On the other hand, for TDMA and FlexRay, the processing cycles are considered synchronized to each other and also to the arbitration cycle, i.e., all tasks and the arbitration cycle start at time 0.

- The DYN segment of FlexRay corresponds to 20% of the total arbitration cycle and enables cycle multiplexing. Namely, there are two minislot assignments which alternate with each other in consecutive arbitration cycles.

The system model exploits the interference abstraction ATA. Therefore, we analyzed the WCRT of a selected task running on a particular core (CUA), while all other cores were modelled by an access request generator that emits request streams bounded by their aggregate interference curve (Section 4.6). Additionally, abstractions from Section 4.7 were applied whenever possible. Verification of the WCRT queries was performed with Uppaal v.4.1.7 on a system with a dual-Core AMD Opteron CPU @2.7GHz and 8 GB RAM.

Table 4.5 shows the verification time required for each WCRT estimate for the different system configurations and arbitration policies. For comparison, the respective verification times for FTA are also included, when available. For the FCFS and RR cases, the analysis scalability for FTA gets severely challenged due to the complexity of the considered industrial application and the assumption of non-synchronized processing cycles among the cores. Particularly, non-determinism w.r.t. the starting time of each task's first execution causes the analysis not to scale beyond 2 cores. The application of the interference abstraction (ATA), however, enabled us to overcome this obstacle and obtain safe upper bounds on the WCRT of the considered tasks in a few minutes for systems with up to 24 cores for FCFS and 64 cores for RR and FlexRay. This is a major step forward compared to any of the existing approaches, opening the way for efficient interference analysis even in many-core systems. In the case of TDMA arbitration, analysis scales very efficiently for any number of cores due to abstraction 4 of Section 4.7, i.e.,

Cores	FCFS (%)	RR (%)	TDMA (%)	FlexRay (%)
2	0	0	0	0
4	11.1	11.1	0	0
8	13.8	13.8	0	**1.8**
16	17.2	17.2	0	-
24	**17.3**	17.3	0	-
32	-	16.8	0	-
64	-	**25.4**	0	-

Table 4.6: Accuracy of ATA compared to state-of-the-art methods. The results define the relative difference of ATA-derived WCRT bounds compared to FTA for FCFS/RR (2 cores) and FlexRay, to [PSC+10] for FCFS/RR (more than 2 cores) and to [SCT10] for TDMA.

the non-representation of interference in the system model. Because of this, the verification time in Table 4.5 is the same for both systems with (ATA) and without (FTA) the interference abstraction.

Furthermore, Table 4.6 shows the accuracy of ATA, measured as the relative difference to the best known WCRT estimate for each scenario. For FCFS and RR arbitration schemes in systems with up to 2 cores, accuracy was compared against the results given by FTA. For systems with more than 2 cores, however, accuracy was compared against the methods presented in [PSC+10]. A value of '0', in this case, means that the results of both ATA and the analytic method exhibit the same degree of pessimism. For TDMA systems, accuracy was evaluated against results obtained with the method in [SCT10]. Since for FlexRay arbitration no analytic methodology is known, the accuracy of the obtained WCRTs was compared to the results of FTA when the latter were available.

For FCFS and RR, comparison shows that the results of ATA can be more pessimistic (up to 25.4%) than the analytic approach [PSC+10], with higher pessimism for higher number of cores. As main source of this pessimism, we identified the behavior of the *Access Request Generator* TA in the abstract system specification (Section 4.6.3), which emits interfering requests non-deterministically over time, thus enabling the exploration of several request streams that are bounded by α^{st}, but may never be encountered in the real-time system. Unlike the results of case study I (Section 4.8.1.3, it seems that the structure and the parameters of the superblocks in this case study (general superblocks with relatively many access requests and very high access latency T_{acc} compared to superblocks' execution time) lead to high pessimism for the ATA analysis.

For TDMA, the ATA results are identical to those of the analytic method [SCT10]. Finally, for systems with a FlexRay arbiter, the WCRT estimates are identical for the FTA and ATA approaches for systems with 2 and 4 cores, and only slightly more pessimistic (1.8%) for ATA for systems with 8 cores. Note that beyond 8 cores, pessimism could not be evaluated

as no other approach can compute tight WCRT bounds for tasks with synchronous access requests to a FlexRay-arbitrated resource.

In summary, the results show that ATA analysis scales efficiently to a large number of cores, unlike previous state-based analysis approaches [GELP10, LYGY10]. Analysis scalability is achieved without compromising the accuracy of the obtained results in the cases of TDMA and FlexRay resource arbitration. For purely event-driven arbitration policies, like FCFS and RR, the accuracy of ATA analysis as compared to FTA and state-of-the-art analytic approaches depends on the characteristics of the considered applications. For instance, not being able to enforce the dedicated superblock execution model has a significant contribution to analysis pessimism.

4.9 Summary

This chapter presented a framework for state-based worst-case response time (WCRT) analysis of periodic tasks that are executed in parallel on a multi-core platform and perform synchronous (blocking) accesses to a shared resource under e.g., FCFS, RR, TDMA or FlexRay arbitration schemes. For achieving deterministic execution and hence, analysis scalability, tasks are organized in dedicated superblocks, namely sequences of resource access and computation phases. We showed how such systems can be precisely modelled with timed automata and analyzed using exhaustive model-checking techniques (FTA analysis). Empirical evaluations with benchmark applications showed that the proposed approach delivers safe WCRT bounds, which due the precise system model lie very close to simulation-driven results from the Gem5 architectural simulator. On a next step, we abstractly represented access requests of cores which compete for the shared resource against a core under analysis by a real-time calculus arrival curve and integrated this curve into the timed automata system specification (ATA analysis). A case study based on a real-world automotive application showed that this method scales much better than previous state-based approaches without compromising the accuracy of the WCRT estimates.

The presented methods are the first to analyze the worst-case response time of tasks with synchronous accesses to a shared resource arbitrated by FlexRay and can be easily extended to model any well-defined arbiter of commercial-off-the-shelf multicores. This opens the way for safe and tight worst-case timing analysis even for industrial-size applications on many-core systems.

5

Deployment of Mixed-Criticality Scheduling on a Multi-Core Architecture

The deployment of mixed-criticality applications on resource-sharing multicores is challenging mainly due to two reasons. The first reason is the need to bound temporal interference among applications with different safety criticality levels for certification purposes. This challenge has been addressed in Chapters 2 and 3, by proposing scheduling policies that avoid inter-criticality interference by construction. The second reason is the implementation of mixed-criticality schedulers, which is itself subject to certification. Mixed-criticality scheduling policies typically employ runtime mechanisms to monitor task execution, detect exceptional events like task overruns, and react by switching scheduling mode. Implementing such mechanisms efficiently, but also with bounded overhead, is crucial for any scheduler to detect runtime events and react in a timely manner without compromising the system's safety. Although several mixed-criticality multi-core scheduling approaches have been proposed in recent years, currently there are very few implementations on hardware that demonstrate the ability to bound interference on shared resources and the overheads of scheduling mechanisms.

To address this necessity, we develop in this chapter a mixed-criticality runtime environment on the Kalray MPPA-256 many-core processor. The runtime environment implements the FTTS scheduling policy of Chapter 3. We evaluate different mechanisms to implement the scheduling primitives and we propose a worst-case response time analysis which accounts for scheduling overheads and for the inter-task

interference on the shared cluster memory. Using realistic benchmarks from avionics and signal processing, we validate the adherence to the analytic response time bounds and demonstrate a maximum achievable utilization of 78.9% on the 16 cores of an MPPA-256 cluster. This result promotes FTTS and similar policies as a viable solution for the efficient and predictable deployment of mixed-criticality multi-core systems.

5.1 Introduction

The prevalence of multi-core architectures in the electronic market has led to an ongoing shift from single-core to multi-core designs even in safety-critical domains, such as avionics and automotive [NP12]. As has been previously discussed, this shift is challenging due to the need to bound interferences on shared platform resources. Such interference can cause lower-criticality applications to delay higher-criticality applications, e.g., upon accessing a shared memory, thus hindering certification. Scheduling of mixed-criticality applications requires typically complex mechanisms to enable sufficient isolation among different criticality levels. Consider for instance the flexible time-triggered and synchronization-based (FTTS) scheduling policy of Chapter 3, which is based on frequent inter-core synchronization through a barrier mechanism, dynamic scheduling decisions at runtime, etc. Given that the scheduler itself is subject to certification, such mechanisms need to be implemented with bounded execution times.

Although theoretical aspects of mixed-criticality multi-core scheduling have been studied [BD16], currently there are very few implementations on hardware that demonstrate efficient resource utilization (through efficient scheduling mechanisms) or the ability to bound interference on shared resources. In this chapter, we show that the deployment of mixed-criticality applications on multicores can be achieved with a high degree of *predictability* and *efficiency*, such that real-time guarantees are provided for all applications and a high utilization of the platform resources is possible. We accomplish this based on four key design factors:

1) Timing-predictable hardware. Providing real-time guarantees on multicores requires that execution times and resource access times can be (tightly) bounded. For this, we employ the Kalray MPPA-256 processor [dDvAPL14], a cluster-based architecture consisting of clusters with a 16-core shared-memory architecture. To the best of our knowledge, it is the only many-core platform where the cores are designed for timing compositionality [WGR+09]. This makes it appropriate for bounded worst-case response time analysis (see also Chapter 3).

2) Adaptive temporal partitioning. Our multi-core scheduling approach reduces the complexity of interference analysis by applying Isolation Scheduling (IS, Chapter 2), i.e., by temporally partitioning the

platform and allowing only applications of the same criticality to be executed at any time. For efficient resource utilization, the schedule of the temporal partitions can be dynamically adapted at runtime to react to occasionally higher resource demand (from high-criticality applications). To further reduce the complexity of bounding intra-partition interference, we look into a subset of IS-compliant scheduling policies which do not allow (dynamic) task preemptions or migrations. In the following, we refer to this scheduling class as adaptive temporal partitioning. Note that the FTTS scheduling policy of Chapter 3 belongs to this class.

3) Efficient implementation of scheduling primitives. The successful deployment of mixed-criticality applications on multi-core architectures depends on the ability to implement scheduling primitives, such as inter-core synchronization and dynamic scheduling adaptations, with bounded and low overhead. With the majority of commercial multicores and operating systems being optimized for average-case rather than worst-case performance, this is not trivial. In previous work, we have shown that the overhead of mixed-criticality mechanisms can have a prohibitive effect on schedulability [SGH+15]. To avoid this, we employ the simplest possible primitives for the implementation of adaptive temporal partitioning on the MPPA-256 and optimize them with respect to runtime overhead.

4) Bounded interference on shared resources. Bounding the tasks' mutual delays due to interference on shared resources is highly complex, since all possible overlapping access patterns need to be considered and all resource arbitration mechanisms need to be precisely modelled. With an increasing number of applications and shared resources, random access patterns and proprietary architectures, the problem of interference bounding becomes soon intractable [DAN+13]. We show how implementing adaptive temporal partitioning on the MPPA-256 can lead to a tight bounding of such delays.

This chapter focuses mainly on the last two design goals. Note that although existing mixed-criticality policies such as FTTS and the cyclic-executive approach of [BFB15] implement adaptive temporal partitioning, so far there has been insufficient empirical evidence on whether these two goals can be achieved on available multicores. By presenting the first deployment of adaptive temporal partitioning on a timing-predictable many-core platform, we show that this approach is indeed a viable solution to mixed-criticality scheduling.

Contributions. The main contributions of the chapter can be summarized as follows:

- We develop a runtime environment for adaptive temporal partitioning on the MPPA-256 Andey processor. We propose alternative implementations for the scheduling primitives and compare them w.r.t. overhead.

- We use an accurate model of the MPPA-256 shared memory and benchmarks to bound the worst-case delay that a task can suffer due to contention on the memory path. The results improve upon the analysis of Section 3.5 by considering an additional source of interference between neighbouring processing cores, which was not modelled before.

- We present a worst-case response time analysis which accounts for the scheduler overheads and the worst-case interference on the memory path. Based on this, we can provide real-time guarantees for the scheduled task sets.

- With a set of industrial-representative benchmarks, we validate runtime adherence to the analytically derived response time bounds and show that complex applications with utilization up to 78.9% can be scheduled on 16 cores without deadline misses. This utilization, which is significantly higher than previously reported results [SGH+15, PMN+16, BDN+16], confirms the applicability of adaptive temporal scheduling for efficient *and* predictable deployment of mixed-criticality applications on multicores.

Outline. The chapter is organised as follows. Section 5.2 provides an overview of existing implementations of mixed-criticality systems. Sections 5.3 and 5.4 present the application and scheduling models and the relevant features of the MPPA-256 architecture for timing analysis. Section 5.5 details the implementation of our runtime environment and Section 5.6 presents a method for bounding the worst-case length of the temporal partitions. Section 5.7 presents the empirical evaluation on the MPPA-256 platform and Section 5.8 summarizes the main results of the chapter.

5.2 Related Work

Previous chapters presented alternative approaches that have been proposed in research literature for bounding temporal interference among applications with different criticality levels that are co-hosted in a multi-core platform and access shared resources, see Section 2.2 and 3.2. From these approaches, we adopt Isolation Scheduling and particularly, a subset of IS-compliant approaches which, like FTTS (Chapter 3), do not permit task preemptions or migrations so as to reduce the complexity of intra-criticality interference analysis. In the following, we highlight existing works that consider implementation aspects of mixed-criticality scheduling and explicitly account for the runtime overhead of scheduling mechanisms on commercial-off-the-shelf platforms.

First, Herman et al. [HKM+12] considered the implementation and runtime overhead of multi-core mixed-criticality scheduling by implementing the scheduling method of [ABB09, MEA+10] in the real-time operating system LITMUS [CLB+06]. The implemented framework operates at kernel level and is customized for the specific scheduling scheme. Unlike our work, the scheduling scheme in [HKM+12] does not explicitly support temporal isolation among criticality levels in the presence of shared non-computational resources. Huang et al. [HGL] implemented several mixed-criticality policies on top of a standard Linux kernel and evaluated their runtime overheads on an Intel Core i7 processor. The developed framework, however, addresses single-core and mainly priority-driven scheduling policies. Sigrist et al. [SGH+15] presented a user-space implementation of the FTTS scheduling policy (Chapter 3) and partitioned EDF-VD [BCLS14] on top of the Linux operating system. The authors opted for a user-space implementation for quick prototyping of different scheduling policies and comparison of their performance on different platforms. Their evaluation on Intel Xeon Phi and Intel Core i5 showed that the overhead of certain runtime mechanisms, such as the barrier synchronization and the sub-frame initialization of FTTS, can be extremely high, resulting in significant schedulability loss. This can be justified since neither the operating system nor the target platforms were designed for timing predictability. We show how to mitigate such effects through a careful, light-weight implementation of the scheduling primitives on the Kalray MPPA-256.

To the best of our knowledge, this chapter presents the first implementation of adaptive temporal partitioning on a many-core platform designed for timing predictability. From an implementation perspective, close to our work lie the scheduling frameworks of [PMN+16, BDN+16]. Both aim at providing contention-free execution of safety-critical applications on the Kalray MPPA-256. Unlike our work, the authors eliminate all possible sources of interference on shared resources by adopting spatial partitioning of cores and memory banks, time-triggered access to shared resources, hypervisors for enforcement of resource budgets [PMN+16] and/or dedicated execution models with computation and resource access phases (similar to the superblock model of Chapter 4) and fine-grain scheduling of these phases [BDN+16]. Although the scope of these works is more general, as they can utilize several compute clusters of the MPPA-256 (our framework currently does not support inter-cluster communication), the methods for elimination of contention on shared resources can have a detrimental effect on schedulability. The empirical evaluation of the frameworks validates this hypothesis, since in [BDN+16] the maximum achievable utilization of a cluster with 14 cores is 14.3% for task sets with a 10% accessing-to-execution time ratio. We show that a much higher utilization can be achieved by allowing contention among same-criticality applications.

5.3 System and Scheduling Model

We consider the same application model as in Section 3.3 and mixed-criticality scheduling policies that implement adaptive temporal partitioning such as FTTS of Section 3.4. For convenience, we recapitulate here the basic principles of the application and scheduling model for a simplified dual-criticality system. For a more detailed presentation, please refer to Chapter 3.

5.3.1 Mixed-Criticality Task Model

We consider periodic mixed-criticality task sets $\tau = \{\tau_1, \ldots, \tau_n\}$ executed on a shared-memory multi-core architecture with m processing cores. For simplicity, we focus on dual-criticality systems, in which the criticality levels are denoted as high (HI) and low (LO). Each task $\tau_i \in \tau$ is characterized by a tuple $\tau_i = (T_i, D_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where $T_i, D_i \in \mathbb{N}^+$ denote the period and relative deadline, $\chi_i \in \{\text{LO}, \text{HI}\}$ the criticality level, and $C_i = (e_i, \mu_i)$ represents an upper bound on the task's execution time (e_i) and number of shared memory accesses (μ_i). The execution time e_i is specified when τ_i runs in isolation, i.e., *without* considering the delay it may experience due to contention on the shared memory[1]. Each task has two execution profiles, at different assurance levels [Ves07]. For high-criticality tasks, the HI-level profile $C_i(\text{HI})$ is more conservative since at a higher assurance level more stringent safety guarantees need to be provided. Depending on the actual profile of high-criticality tasks at runtime, a dual-criticality system can execute in two modes. In LO (default) mode, all tasks are scheduled according to their LO-level parameters $C_i(\text{LO})$. If a high-criticality task runs according to its HI-level profile $C_i(\text{HI})$, from then on the system switches (temporarily or permanently) to HI mode. In HI mode, high-criticality tasks require more resources, i.e., if χ_i =HI: $e_i(\text{HI}) \geq e_i(\text{LO})$ and $\mu_i(\text{HI}) \geq \mu_i(\text{LO})$. Low-criticality tasks may need to execute in degraded mode, i.e., with reduced functionality, to preserve the system schedulability. Execution in degraded mode is specified by $C_i(\text{HI})$, i.e., if χ_i =LO: $e_i(\text{HI}) \leq e_i(\text{LO})$ and $\mu_i(\text{HI}) \leq \mu_i(\text{LO})$. For simplicity, we assume that the first job of all tasks is released at time 0 and that the relative deadline of τ_i is equal to its period, i.e., $D_i = T_i$. Precedence constraints may exist among tasks with equal periods as long as the resulting dependencies are acyclic. Finally, we define the total utilization of a periodic dual-criticality task set τ as

[1] In Section 3.3, e^i was defined as the maximum computation time without considering at all the memory accessing time. Here, e^i is the maximum execution time including memory accessing when the task runs in isolation (no memory contention). This differentiation does not affect WCRT analysis, but makes the extraction of parameters e^i, μ^i easier in the experimental evaluation of our runtime environment.

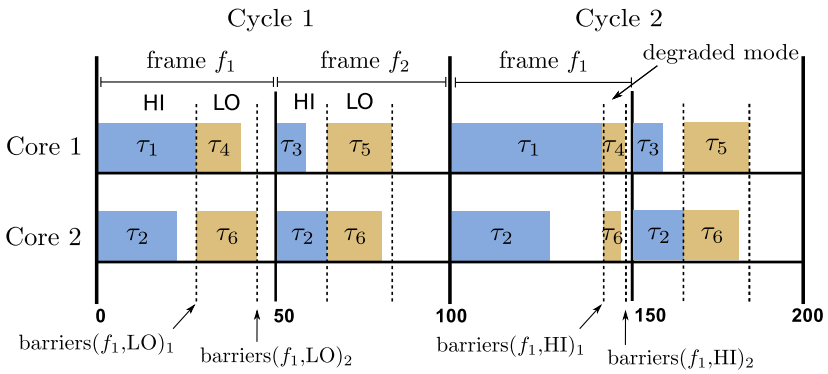


Figure 5.1: Two consecutive FTTS scheduling cycles ($H = 100$), with 2 frames ($L_{f_1} = L_{f_2} = 50$) divided into flexible-length HI and LO sub-frames. Jobs in frame f_1 run in LO mode in the first cycle and in HI mode in the second cycle.

the maximum utilization across LO and HI execution mode:

$$U_\tau := \max\left\{U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau), U_{\text{LO}}^{\text{HI}}(\tau) + U_{\text{HI}}^{\text{HI}}(\tau)\right\} , \tag{5.1}$$

where $U_{\text{x}}^{\text{y}}(\tau)$ represents the total utilization of the tasks with criticality level x for their $y-$level execution time (in isolation), i.e., $U_{\text{x}}^{\text{y}}(\tau) = \sum_{\chi_i=\text{x}} e_i(y)/T_i$.

5.3.2 Adaptive Temporal Partitioning

As mentioned earlier, adaptive temporal partitioning is a special case of Isolation Scheduling (Chapter 2). Isolation Scheduling policies partition a platform temporally and allow only jobs of a single task class (here, criticality level) to utilize the platform resources at any time. To implement this principle, all processing cores switch synchronously between task classes. The switch can be triggered dynamically or based on a static time-triggered pattern. In contrast to the more general Isolation Scheduling model, adaptive temporal partitioning does not allow task preemptions or migrations. This makes it applicable to the MPPA-256, where cores do not support multitasking, and reduces the complexity of interference bounding within each partition. In the following, we recapitulate the basic principles of FTTS scheduling (Section 3.4), which is a representative policy for adaptive temporal partitioning and is implemented in our runtime environment.

FTTS combines time and event-triggered task activation. An FTTS schedule repeats over a *scheduling cycle* H equal to the hyper-period of the tasks in τ. The cycle consists of fixed-length *frames* (set $\mathcal{F}$). Each frame is divided further into two flexible-length *sub-frames*, the first containing high-criticality tasks (HI sub-frame) and the second containing low-criticality tasks (LO sub-frame). The beginning of frames and sub-frames is synchronized among all cores. Frames start at predefined time

points. Within a frame, the HI sub-frame begins immediately. The LO sub-frame begins once all tasks of the HI sub-frame complete execution across all cores. Synchronization for switching from the HI to the LO sub-frame is achieved dynamically via a barrier mechanism, for efficient resource utilization. Namely, if the HI sub-frame completes earlier than statically expected, the platform can be used by the following task class, without wasting resources. Within the sub-frames, tasks are scheduled sequentially on each core following a predefined order. The mapping of tasks to cores is fixed.

At runtime, the length of a sub-frame varies depending on: (i) the exhibited execution profile of high-criticality tasks, (ii) the memory interference patterns of the co-running tasks, (iii) the scheduling overheads, e.g., the cost of barrier synchronization on the target platform. We use $barriers(f, \text{LO})_k$ (resp. $barriers(f, \text{HI})_k$) to denote the worst-case length for the k-th sub-frame of frame $f \in \mathcal{F}$, when the tasks in it exhibit their LO (resp. HI)-level execution profile. At runtime, the FTTS scheduler monitors the actual length of the sub-frames. If the length of the HI sub-frame does not exceed $barriers(f, \text{LO})_1$, it triggers normally the execution of tasks in the LO sub-frame (LO mode). However, if the length of the HI sub-frame exceeds $barriers(f, \text{LO})_1$, the tasks of the LO sub-frame are triggered in degraded mode (HI mode). A LO to HI mode switch depends, therefore, on the monitored length of the HI sub-frame and it can affect the execution of LO-criticality tasks only in the current frame. The same procedure is repeated independently for the following frames.

We show how to compute the worst-case sub-frame lengths, namely function *barriers*, in Section 5.6. Once this function is computed for a given FTTS schedule, we consider the schedule *admissible*, i.e., all tasks are guaranteed to meet their deadlines, if in every frame $f \in \mathcal{F}$ (with fixed length L_f), the last sub-frame completes by the end of the frame in either LO or HI mode, i.e., if $\forall f \in \mathcal{F}$:

$$\max \left\{ \begin{array}{l} barriers(f, \text{LO})_1 + barrier(f, \text{LO})_2, \\ barriers(f, \text{HI})_1 + barrier(f, \text{HI})_2 \end{array} \right\} \leq L_f \tag{5.2}$$

Finding an admissible FTTS schedule for a given task set τ can be performed by the optimization framework of Section 3.6 once a method for computing *barriers* is specified.

Note that the deployment of FTTS on a multi-core platform requires support for: global time synchronization for the time-triggered frame activation, inter-core barrier synchronization, inter-core communication for the implementation of dynamic scheduling decisions, static per-core schedule tables. These mechanisms are supported in most commercial platforms. The results of this chapter are relevant not just to FTTS, but to any scheduling policy that uses the above primitives.

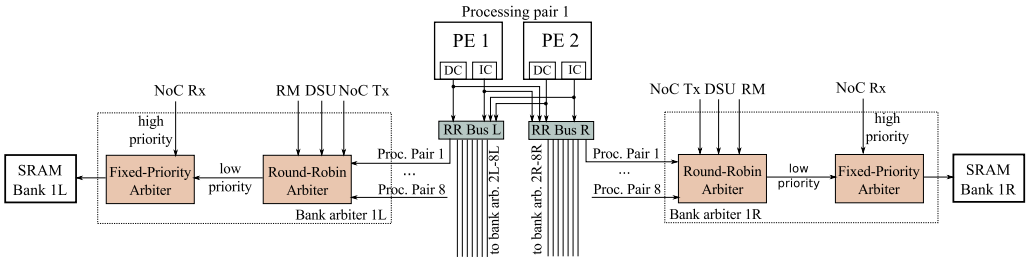


Figure 5.2: Memory path from one processing pair to one memory bank on the left or right side of an MPPA-256 Andey compute cluster.

5.4 Kalray MPPA-256

We present a brief overview of the Kalray MPPA-256 with emphasis on the memory system, which is important for the timing analysis in Section 5.6. Note that the memory model includes a source of temporal interference within pairs of neighbouring cores, which was not addressed in the respective model of Chapter 3. More details on the MPPA-256 architecture and runtime environment can be found in [dDvAPL14].

Architecture. The Kalray MPPA-256 Andey processor integrates 256 processing cores (PE) and 32 resource management (RM) cores, which are distributed across 16 compute clusters and four I/O sub-systems. Application code is executed on compute clusters, whereas the I/O sub-systems are dedicated to the management of external memories and I/O devices. Each compute cluster and I/O sub-system owns a private address space and inter-cluster communication is supported by network-on-chip.

A compute cluster includes 16 processing cores and one resource management core, each with private instruction and data caches. All cores implement the same VLIW architecture and execute at 400 MHz. Each cluster has a local 2 MB SRAM memory, which is organized in 16 independent banks with a dedicated access arbiter for each bank. The memory banks are arranged in two sides (left, L and right, R) of 8 banks. Each memory bank arbiter is connected to 12 masters: the NoC Rx interface, the NoC Tx DMA engine, a debug support unit (DSU), the resource management core and 8 processing pairs. A processing pair consists of two neighbouring processing cores. As shown in Figure 5.2 (for simplicity, only for processing pair 1), the access path from a processing core to a memory bank passes through three request arbiters. The path starts with a bus shared by the cores of the processing pair. Access to this bus is arbitrated round-robin among the two data caches (DC) and the two instruction caches (IC) of the cores. Each processing pair has two buses, one for each side. This means that if the two cores of the processing pair need to access simultaneously two memory banks on different sides, there is no interference on the bus level. The buses of the processing pairs, along with all other masters are connected to the bank arbiters,

which implement a non-preemptive round-robin arbitration scheme with higher priority for NoC Rx (illustrated by the two-stage arbitration on Figure 5.2).

Runtime Environment. The runtime environment used in this work (Accesscore 1.4) supports two programming modes for the cores of a compute cluster: one using the services of a light-weight operating system and precompiled POSIX libraries for thread management and synchronization (we refer to it as OS) and a bare-metal mode (we refer to it as BM). Through Makefiles, the programmer can specify at compile time which binaries will be executed on each compute cluster. In OS mode, the code is automatically launched on processing core PE1 of every enabled cluster. This core can then spawn POSIX threads on the remaining 15 processing cores, PE2 to PE16 (maximum one thread per core). The resource manager of each cluster hosts the operating system and is involved in the management of interrupts, NoC access requests and system calls. In BM mode, the code is automatically launched on the resource management core of the cluster, so all 16 processing cores PE1 to PE16 can be used for application execution. The programmer has to handle explicitly the thread creation, management and synchronization, interrupt handling, NoC accesses, cache coherence using low-level operations. The BM mode offers complete control of the platform to the programmer, yet at the cost of increased programming effort. For comparison purposes, we implement our runtime environment in both modes. Concerning the shared cluster memory, the programmer can configure the memory address mapping as sequential (each bank spans 128 KB consecutive addresses) or interleaved (data are distributed over all banks). The programmer can also enable or disable the data cache of each individual core and is responsible for cache coherence, which is entirely managed through software.

5.5 Implementation of Scheduling Primitives

Adaptive temporal partitioning, such as FTTS, is based on three main primitives: (i) the enforcement of a predefined time pattern within a scheduling cycle, (ii) the dynamic synchronization among all cores, e.g., upon completion of their tasks in a partition, (iii) the communication of dynamic decisions of the scheduler to all cores, e.g., when the next partition needs to be triggered in degraded mode. The first goal for a successful deployment is to implement these primitives with *bounded* and *low* overhead. In the following, we discuss alternative implementations on the MPPA-256 and compare their overhead. We start by introducing the basics of our FTTS runtime environment on MPPA-256.

FTTS Runtime Environment. Our runtime environment features a

scheduler thread, mapped on processing core 1 (PE1) of a compute cluster in OS programming mode (resp. on the resource management core in BM mode), and 1 to 15 (resp. 1 to 16) *worker threads*, mapped on processing cores PE2 to PE16 is OS mode (resp. PE1 to PE16 in BM mode). The scheduler thread is responsible for enforcing the time pattern of an FTTS schedule (timekeeping), synchronizing with the worker threads at the beginning on each FTTS frame, performing dynamic decisions at runtime and communicating them to the worker threads. The worker threads are responsible for executing sequentially the functions that implement the task functionality in each FTTS sub-frame and synchronizing with the scheduler thread when all functions are executed. Each worker thread has access to its schedule table, hence activation of individual task functions is managed without involving the scheduler. Execution of every task function is followed by a data cache flush to ensure cache coherence. The memory address mapping is interleaved, since all threads share libraries and scheduling information and shared-memory inter-task communication is supported, thus making the benefits of bank privatization under sequential address mapping unattainable. The scheduler thread collects profiling data regarding the beginning/end of each frame and sub-frame. Respectively, the worker threads collect profiling data regarding the beginning/end of each task. The overhead measurements later in this section are based on post-processing of these profiling data.

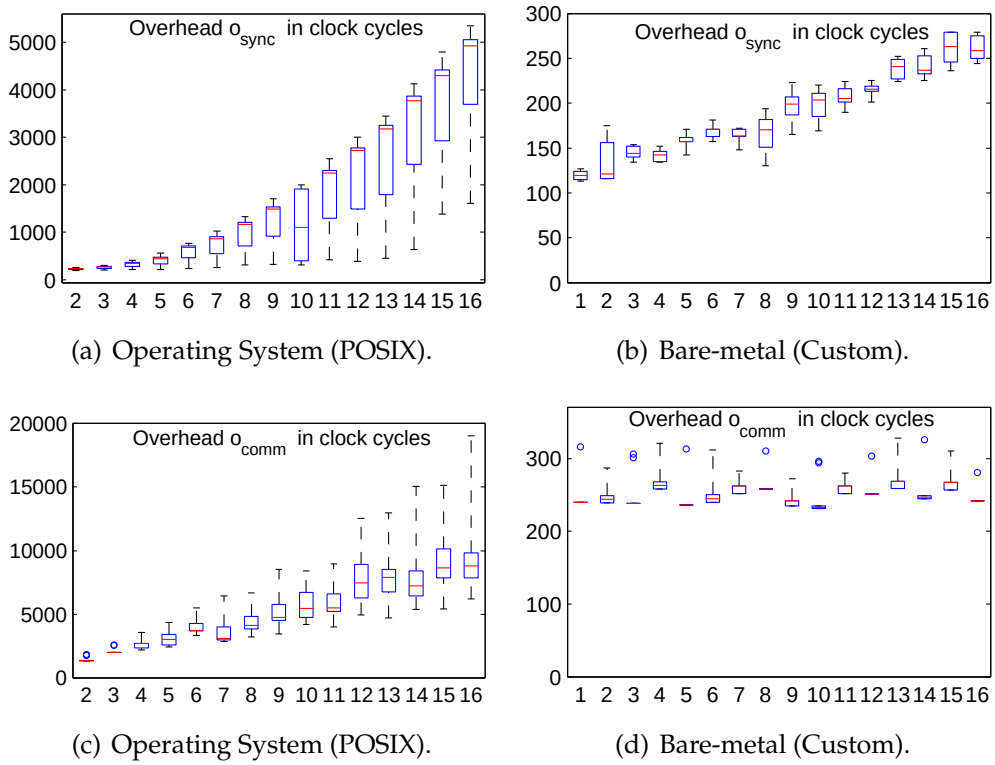
Time-triggered Activation of FTTS Frames. Timekeeping for synchronous time-triggered frame activation is a responsibility of the scheduler thread. Our runtime can be configured to use the following methods: (i) the POSIX function `nanosleep`, which is called after the completion of the last sub-frame with the remaining time until the end of the frame as input argument, (ii) the POSIX function `cond_timed_wait` which is configured at the beginning of the frame with an absolute value indicating its exact completion time, and (iii) a custom busy-wait function (default option) which is called upon completion of the last sub-frame and performs `nop` operations for a given amount of cycles (dynamically computed by the scheduler). The first two methods are applicable only in OS mode. The overhead of their call often leads to the scheduler "waking up" hundreds of cycles later than expected. In contrast, the busy-waiting approach is highly accurate, resulting in negligible offsets (few cycles) from the expected frame completion time.

Barrier Synchronization. This primitive is of utmost importance for a correct and efficient FTTS deployment, given that barrier synchronization among all active cores of a compute cluster is performed at the beginning of each frame and upon completion of each sub-frame. Our runtime environment supports two alternative implementations: (i) the POSIX function `pthread_barrier_wait`, which is supported only in OS mode,

and (ii) a custom function using event signals, which is supported only in BM mode. In the second case, we are able to use hardware event lines which connect every processing core directly to the resource management core of a cluster. This way, barrier synchronization is achieved by letting all worker threads notify the scheduler thread on the resource management core, e.g., once the tasks of the current sub-frame are completed, and letting the resource management core broadcast the end of the barrier once it has received a notification from all processing cores.

Figures 5.3(a), 5.3(b) depict the statistical distribution (box-whisker-plot) of the overhead of the two implementations as a function of active cores, based on measurements. For the measurements we used an FTTS schedule, consisting of two frames, with two sub-frames each. In each sub-frame, one or two instances of a single task run on every worker thread. The task performs busy waiting for 2 μs. We selected this setup after observing that the cost of barrier synchronization typically increases when all worker threads reach the synchronization point almost simultaneously (which happens if one instance of the same task runs on every worker thread) and when the time distance between successive barrier synchronizations is short (which is achieved by the low busy waiting interval). The FTTS schedule was executed on the MPPA-256 for 100,000 scheduling cycles (in total 200,000 frames). For each FTTS frame, we measured the interval between the completion of the last task of a sub-frame and the completion of barrier synchronization as seen by the scheduler thread (o_{sync} in Figure 5.4). Repetitions of the experiment with other schedules (with different busy waiting interval or different tasks from Section 5.7) produced similar results. Therefore, we consider it a realistic upper bound for the barrier synchronization overhead in our experiments. Note that the overhead of the custom implementation is an order of magnitude lower than that of the POSIX function call.

Communication of Scheduling Decision. This operation is performed upon completion of the HI sub-frame in a frame. After barrier synchronization, the scheduler checks the elapsed time since the beginning of the HI sub-frame. If the sub-frame duration surpasses a statically determined value (function *barriers*), the scheduler notifies all worker threads to run the tasks of the following LO sub-frame in degraded mode. Note that this is the only scheduling decision that must be communicated from the scheduler to the worker threads, since otherwise the worker threads manage the execution of their schedule tables independently. Our runtime provides two alternative implementations: (i) the POSIX function `pthread_cond_wait`, which is supported only in OS mode and is based on conditional variables and mutex locks for blocking the worker threads until a decision is made by the scheduler and communicating the decision through a shared protected variable in memory, and (ii) a custom function, which is supported only



(a) Operating System (POSIX). (b) Bare-metal (Custom).

(c) Operating System (POSIX). (d) Bare-metal (Custom).

Figure 5.3: Runtime overheads of scheduling primitives on MPPA-256 (200,000 measurements). On each box, the central mark, bottom and top edges indicate the median, 25th and 75th percentiles, respectively. The whiskers extend to 10 times the interquartile range. Outliers are denoted with circles.

in BM mode and uses the hardware event lines. In the second case, the same mechanism is used as for barrier synchronization, differing in that the scheduler does not notify the worker threads immediately after it receives their notifications of completion, but it first makes the scheduling decision, it writes the decision in a global variable in memory and broadcasts a signal to all processing cores afterwards.

The statistical distribution (box-whisker-plot) of the overhead of the two implementations for the same experiment as before (over 100,000 scheduling cycles of the FTTS schedule), is depicted in Figures 5.3(c), 5.3(d). The overhead is defined as the measured interval between the completion time of barrier synchronization at the scheduler thread and the latest starting time of a task in the LO sub-frame (o_{comm} in Figure 5.4). The difference is significant, with the OS implementation having up to 68 times (two orders of magnitude) higher maximal overhead than the BM implementation. A full explanation of this difference is impossible since the implementation of the POSIX libraries on the MPPA-256 is not open-source. One possible factor for the higher overhead is related to the data cache flush, which automatically follows the call of POSIX

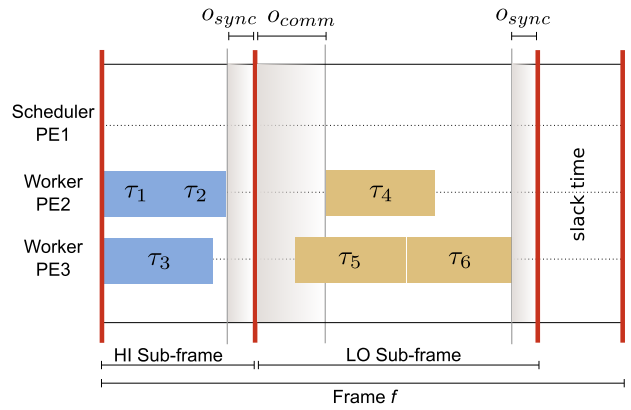


Figure 5.4: Timing diagram for an FTTS schedule frame.

synchronization functions due to the lack of hardware cache coherence protocols. In BM mode, such a flush operation is unnecessary since the scheduler writes explicitly its decision in shared memory (bypassing the cache) and similarly, the worker threads read it directly from the memory. Note also the variance of the measured OS overhead, ranging e.g., from 6,210 to 19,038 clock cycles for 16 active cores. This variance represents a certain source of pessimism in our timing analysis, since the overhead of communicating the scheduler's decision cannot be tightly bounded in OS mode.

5.6 Worst-Case Response Time Analysis

The previous section focused on the first goal for successful adaptive temporal partitioning, i.e., the ability to (experimentally) bound the overhead of scheduling primitives. Here, we investigate also the second goal, i.e., the ability to bound the worst-case delays that tasks experience due to memory contention. Generally, this is an arbitrarily complex problem, since all possible accessing patterns of co-running tasks need to be considered and precise models of the resource arbiters needs to be available. By employing adaptive temporal partitioning on MPPA-256, we reduce the complexity of the problem by restricting the tasks that can interfere at any time, by avoiding preemptions and by accurately modelling all interference sources in the memory architecture (Section 5.4).

In the following, we propose a method for the computation of function *barriers*, i.e., for the offline estimation of the worst-case sub-frame lengths of a given FTTS schedule. The timing analysis accounts for the runtime overheads of the scheduling primitives and for the task execution delays on the shared memory path of an MPPA-256 cluster. This analysis enables the classification of an FTTS schedule as admissible or inadmissible, according to Condition (5.2).

5.6.1 Impact of Runtime Scheduling Overheads

Figure 5.4 presents a timing diagram of an FTTS schedule for a single frame with two sub-frames. The thick lines indicate the completion of barrier synchronization as seen by the scheduler thread at the beginning of the frame, at the end of each sub-frame and at the end of the frame, respectively. We identify the following runtime overheads that have an impact on the actual length of the FTTS sub-frames:

- **Barrier synchronization.** This overhead, denoted as o_{sync} must be considered in every sub-frame, since barrier synchronization is used always to detect the sub-frame completion.

- **Communication of scheduling decision.** This overhead, denoted as o_{comm} must be considered in every LO sub-frame, since the scheduler decides about the normal or degraded execution of subsequent tasks only after completion of the HI sub-frame. For HI sub-frames, $o_{comm} = 0$.

Let $WCRT_i(f, \ell)$ denote the worst-case response time of task τ_i in frame f at level $\ell \in$ {LO,HI}, when it executes in parallel with other tasks (considering all possible interference delays), m the number of active cores in an FTTS schedule, and $S(c, f, k) \subseteq \tau$ the set of tasks executing on core c in the k-th sub-frame of frame f. The worst-case length of sub-frame $k \in \{1,2\}$ in f can be expressed as:

$$barriers(f, \ell)_k = o_{comm} + \max_{1 \leq c \leq m} \left\{ \sum_{\tau_i \in S(c,f,k)} WCRT_i(f, \ell) \right\} + o_{sync} \ . \tag{5.3}$$

We show how to compute $WCRT_i(f, \ell)$, $\forall \tau_i \in \tau$ in the following.

5.6.2 Impact of Interference on Shared Memory

The problem of bounding the worst-case response time of tasks under FTTS scheduling and memory contention on the MPPA-256 was previously investigated in Section 3.5. However, the shared memory model used then differs from the model of Section 5.4 by not accounting for timing interference on the shared bus between the cores of a processing pair. Here, we extend the timing analysis of Section 3.5 to account for this additional source of interference and guarantee safe response time bounds[2]. Recently, Skalistis et al. [SS16] used a similar memory

[2]Note that despite not modelling interference on the shared bus of each processing pair, the experimental results of Sections 3.7 and 3.8 remain correct. This is because the considered systems have up to $m = 8$ processing cores. By selecting one core from each processing pair in an MPPA-256 cluster and assuming no instruction cache misses during task execution, interference on the shared bus cannot exist.

model as in Section 5.4 for WCRT analysis, yet assuming dataflow-based scheduling, where certain tasks cannot interfere due to precedence constraints.

For our analysis, we assume enabled data caches and interleaved memory address mapping, which is the default configuration in our runtime environment. The latter leads to conservative WCRT bounds because any two tasks with the same criticality level are potentially interfering (accessing the same memory bank). The WCRT bounds can, however, be refined if sequential address mapping is applied with a known allocation of data to memory banks (see Section 3.5.1.1). Additionally, we assume that no memory access requests are generated by the NoC Rx, NoC Tx, the debug support unit or the resource management core of a compute cluster. This can be achieved by not allowing any data transfer over the NoC, not using debug support and not performing operations on the resource management core that may result in data/instruction cache misses during the FTTS sub-frames.

Under these assumptions and given the memory model of the MPPA-256 Andey architecture, every memory access request of a task can be delayed at two points: (i) on the shared bus of the processing pair where the task is executed, by pending requests from the other three caches (in the worst-case, all requests target banks on the same side), (ii) on the round-robin arbiter of the target bank, by pending requests from the other seven processing pairs (see Figure 5.2). This yields a worst-case delay of 31 memory accesses until the access request can be served, in case all four caches of the processing pair are active and at least one core is active in each of the seven interfering processing pairs.

Recall that the level-ℓ execution profile $C_i(\ell)$ of a task τ_i consists of its worst-case execution time $e_i(\ell)$ when τ_i runs in isolation and the maximum number of memory access requests $\mu_i(\ell)$ (due to data cache misses) that it can generate. Following the previous discussion, the $WCRT_i(f,\ell)$ of task τ_i which runs in processing pair $1 \leq p \leq 8$ can be upper-bounded by:

$$WCRT_i(f,\ell) = e_i(\ell) + \mu_i(\ell) \cdot \left(N_{caches}(f,p) \cdot \sum_{j=1}^{8} active(f,j) - 1 \right) \cdot T_{acc} \ , \quad (5.4)$$

where the second term specifies the worst-case delay due to interference on the memory path. $N_{caches}(f,p)$ denotes the number of active caches in the processing pair p to which τ_i is mapped. If there is at least one task running on the neighbouring core in the same sub-frame of frame f as τ_i, then $N_{caches}(f,p) = 4$, otherwise $N_{caches}(f,p) = 2$. Function $active(f,p)$ returns 1 if there is at least one task running in processing pair p in the same sub-frame of frame f as τ_i, or 0 otherwise. Finally, T_{acc} bounds the latency of a single memory access (under no contention). Using an assembler-based benchmark, we have estimated a bound of T_{acc} =14

cycles on the MPPA-256 (details below). By substituting $WCRT_i(f, \ell)$ in Eq. (5.3), we can bound the worst-case sub-frame lengths in LO and HI execution mode and validate the admissibility of any given schedule according to Condition (5.2).

Empirical Estimation of Single Memory Access Latency T_{acc}. We describe, here, how we derived the memory access latency bound T_{acc} on the Kalray-MPPA 256 Andey processor[3]. For this purpose, different assembler code snippets were executed in bare-metal mode with interleaved memory address mapping. The execution time of the code was measured using a hardware cycle counter. All experiments were run twice in a row, with the first run to load all code into the instruction cache and the second run for the actual measurement.

For a ground truth comparison, a varying number of no-operation (`nop`) instructions were executed as the body of a hardware-controlled loop. The number of loop iterations was continuously incremented from 1 to 128. As expected, each time the number of loop iterations was incremented by one, the total execution time for all iterations of the loops increased exactly by the number of `nop` words in the loop body. In other words, we were able to measure the execution time of one `nop` instruction as exactly one cycle.

Afterwards, an additional instruction was added to the loop. This triggered a double-word memory access (64-bit load or 64-bit store) and an increment of the memory address by at least 64 bytes to ensure cache misses as well as accesses to different memory banks. The total execution time was now increased by 10 additional cycles per iteration. If the number of `nop` instructions in the loop was smaller than 4, the execution time of one loop iteration was constantly equal to 14 cycles independently of the number of `nop`'s. This execution time was always the same for load and store accesses. We conclude from this that a memory access has a latency of 10 cycles, however loading an entire cache line requires 14 cycles. If the accesses are executed too quickly in a row (less than 4 cycles between two accesses), the processor stalls until the last cache line refill is finished, i.e., for 14 cycles. In this way, we derived the memory access latency T_{acc} = 14 cycles, which is used in the experimental evaluation of the next section.

5.7 Evaluation

This section presents the experiments that were performed on the MPPA-256 to empirically validate the WCRT analysis of Section 5.6 and to evaluate the schedulability loss due to the temporal partitioning (FTTS) constraints, the scheduling overheads and the mutual delays

[3]The development of this benchmark consists joint work with Andreas Tretter.

of co-running tasks on the shared memory. We start by presenting our experimental setup (Section 5.7.1) and the employed benchmarks (Section 5.7.2). The experiments that follow (Section 5.7.3) show that for admissible FTTS schedules, the analytically derived function *barriers* (Eq. (5.3)) bounds the actual sub-frame lengths at runtime, and that an effective utilization of 78.9% on 16 processing cores with realistic workload is possible without deadline misses.

5.7.1 Experimental Framework

The experimental framework enables: 1) specification of a mixed-criticality application, 2) automatic generation of an admissible FTTS schedule (if one exists), 3) deployment of the FTTS schedule on an MPPA-256 cluster for a given number of scheduling cycles, 4) post-processing and statistical analysis of collected profiling data. The required inputs by the programmer are the specification of tasks to be scheduled (period, criticality, execution profiles), the number of available cores, the desired execution mode (OS or BM), and the C/C++ source code which describes the tasks' functionality (initialization and execution).

For the first two steps, we used the FTTS design optimization framework of Section 3.6. This aims at finding an admissible FTTS schedule for a specified application and number of cores, with maximal aggregate slack time at the end of frames. This objective implies a balanced workload among the cores and allows for incremental design, e.g., if more tasks of lower criticality need to be integrated later into the system. For the design space exploration, the framework employs a simulated annealing approach. We extended the framework so that the worst-case timing analysis, which is performed for every visited solution, becomes cognizant of the scheduler overheads and all potential sources of interference on the memory path (Section 5.6). For the overhead of barrier synchronization o_{sync} and communication of scheduling decision o_{comm}, we considered the upper bounds shown in Figure 5.3 for each number of active cores. The output of the optimization framework is an XML file and in case of a BM implementation, also auto-generated source code for the deployment of the schedule on the platform.

For the next two steps, the framework that executes the FTTS schedule repeatedly on the MPPA-256 and collects and post-processes profiling data was built upon an existing C++ runtime environment for static dataflow applications [TPM+15]. The runtime environment was modified to implement the FTTS scheduling policy instead of the previous data-driven approach, to collect/analyze different types of profiling data and to run in bare-metal mode. After the schedule deployment, the collected profiling data (start/finish time and data cache misses of each task execution, start/finish time of each FTTS frame and sub-frame) are sent from the MPPA-256 to a host computer for post-processing.

App.	Task τ_i	Crit. χ_i	Period T_i (ms)	Execution Cycles e_i	Data Cache misses μ_i
FMS	sens_c1	HI	5	14752	84
	loc_c1	HI	5	8545	105
	loc_c2	HI	40	2245	41
	loc_c3	HI	40	11162	103
	loc_c4	HI	40	2189	51
ROSACE	engine	HI	5	1214	8
	elevator	HI	5	1249	10
	aircraft_dynamics	HI	5	9159	31
	h_filter	HI	10	1302	12
	az_filter	HI	10	1301	12
	Vz_filter	HI	10	1299	12
	q_filter	HI	10	1252	12
	Va_filter	HI	10	1296	12
	altitude_hold	HI	20	1220	8
	Vz_control	HI	20	1224	11
	Va_control	HI	20	1256	11
StreamIt	matmult	LO	5	56483	156
	fft	LO	5	19655	102
	bitonic_sort	LO	5	202041	102
	insertion_sort	LO	5	80755	134
	dct_2D_coarse	LO	5	59833	137
	idct_2D_coarse	LO	5	57125	342
	fm	LO	5	36736	905
	filter_bank	LO	5	1538871	253
	autocorrelation	LO	5	2859	12
Synth.	busy_wait_HI	HI	5	80043	6
	busy_wait_LO	LO	5	1600025	6

Table 5.1: Specification of benchmark tasks.

5.7.2 Benchmarks

For the evaluation, we used two avionic benchmarks, nine streaming benchmarks from the StreamIt suite [TKA02] and two synthetic benchmarks which perform busy waiting. The benchmarks are characterized by diverse computational/memory requirements and memory accessing patterns. A brief presentation of the benchmarks follows.

- The flight management system (FMS) [DFG+14] is a safety-critical application, responsible for aircraft localization, flightplan computation, detection of the nearest airport, etc. We used an industrial implementation of an FMS sub-set, consisting of one task (sens_c1) which periodically reads (hard-coded) sensor data and four tasks (loc_cx) which compute the current aircraft location based on the data. The tasks communicate through shared memory,

but their read/write operations are non-blocking. Namely, if no new data exist in a shared buffer, a reader task uses "stale" data, read previously. This avoids execution delays due to blocking.

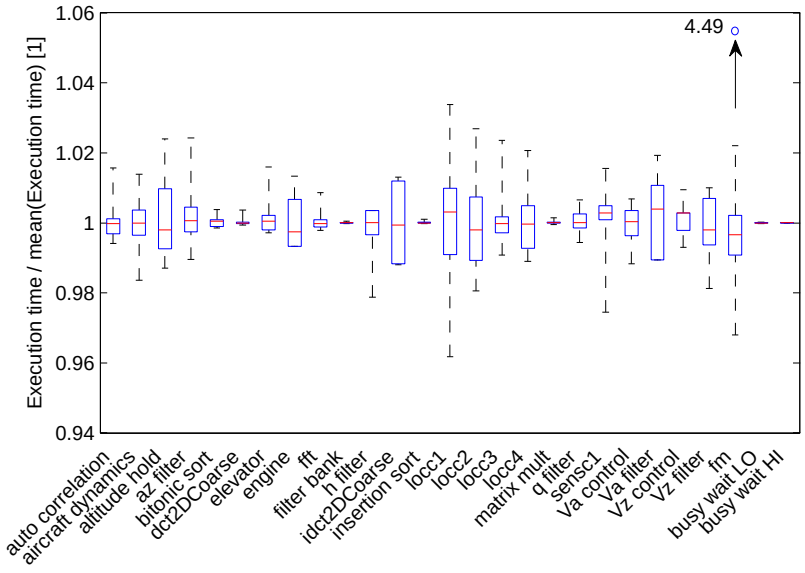
- The ROSACE application [PSG+14, PMN+16] is an open-source avionic benchmark, implementing a longitudinal flight controller. It consists of 11 tasks: three perform environment simulation and eight implement the controller logic. For the environment simulation, which requires inputs simulating different pilot instructions to change the flight level, we used hard-coded inputs. Reading/writing to shared data is again non-blocking.

- For the class of low-criticality applications, we used nine benchmarks from the StreamIt suite [TKA02], since most of them represent computation and memory-intensive applications. The selected benchmarks include: a matrix multiplication of 20x20 matrices, a fast-Fourier transform (fft) of signals with length 128, a bitonic sorting network for lists of 512 integers, an insertion sort algorithm for lists of 512 integers, IEEE compliant implementations of discrete cosine transforms (dct) and inverse discrete cosine transforms (idct) used in the MPEG/JPEG standards, a FM radio with multi-band equalizer, a filter bank for multi-rate signal processing, and a filter generating the autocorrelation series of an input with 2048 elements. For the benchmark implementation, we used the reference C implementations for single cores from the benchmark suite repository. We avoided parallel implementations, since for these applications blocking reads/writes are necessary, which would complicate timing analysis.

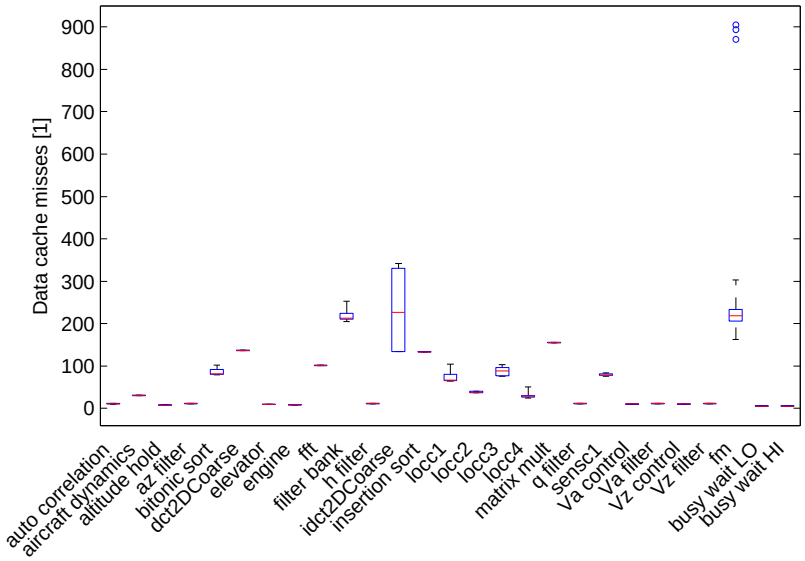
- Finally, we implemented two synthetic tasks which simply perform busy waiting for 200 μs and 4 ms, respectively.

All benchmarks were implemented so that their code and data fit into the memory of a compute cluster, without the need to access the address space of other compute clusters or I/O subsystems. Their inputs are set during an initialization phase prior to their first execution. The specification of all tasks, which is given as input to our FTTS optimization framework, is presented in Table 5.1. The task periods were assigned according to their specification for high-criticality tasks (the FMS periods being down-scaled) or randomly for low-criticality tasks. The execution profiles were obtained through measurements on the MPPA-256, since we do not have access to static analysis tools for this platform.

Specifically, we used FTTS schedules in which either a single task or a sequence of tasks under analysis ran in every sub-frame on a single core. Running the tasks in sequence is important for the FMS and ROSACE applications, where some tasks exhibit their worst-case execution time



(a) Execution time vs mean execution time ratio. For ease of presentation, 10 outliers of "fm" at 4.49 have been omitted.



(b) Data cache misses.

Figure 5.5: Profiling of benchmarks: Statistical distribution of measured values over 10,000 executions in isolation. On each box, the central mark, bottom and top edges indicate the median, 25th and 75th percentiles, respectively. The whiskers extend to 10 times the interquartile range. Outliers are denoted with circles.

depending on the output of previously executed tasks (implicit data dependencies). The FTTS schedules were executed for at least 10,000 scheduling cycles, and the maximum observed execution time and data cache misses in Table 5.1 were extracted from the collected profiling data.

Figure 5.5 presents the statistical distribution of the execution time and data cache miss measurements over 10,000 executions in the form of a box-whisker-plot. Note that the measured execution times include the cost of data cache flush, which takes place at the end of each task execution for data coherence. This operation costs approximately 1,100 cycles. Execution time variance is relatively low for all benchmarks except "fm", which executes for 4.49 times its mean execution time on every 1000th execution. Cache miss variance is also low or zero for most benchmarks, with the exception of "idct2DCoarse" and "fm".

The measured worst-case parameters specify the LO-level execution profile $C_i(\text{LO})$ of all tasks. For high-criticality tasks, we assume $C_i(\text{HI})=C_i(\text{LO})$ and for low-criticality tasks $C_i(\text{HI})=(0,0)$. At design time, an admissible schedule must ensure that *all* tasks meet their deadlines when running according to the execution profiles of Table 5.1 (LO mode). High-criticality tasks are not expected to run longer than $WCRT_i(\text{LO})$. If, nonetheless, the length of a HI-subframe exceeds its statically computed value (e.g., due to unusually high scheduling overhead), the next LO-subframe will not be executed (HI mode).

5.7.3 Results

In the following, we experiment with several combinations of the benchmarks with a two-fold objective: (i) to validate whether the analysis of Section 5.6 bounds the FTTS sub-frame lengths at runtime, and (ii) to investigate the practical limits of our scheduling approach in terms of maximum achievable utilization. With these objectives in mind, we evaluate every deployed FTTS schedule based on two criteria:

- **Number of frame violations**. It is defined as the portion of FTTS frames in which the LO sub-frame was not completed by the end of the frame. If an FTTS schedule is statically deemed admissible based on condition (5.2), then a frame violation should *never* happen at runtime. If it happens, this implies that either our worst-case timing analysis is incorrect or the considered upper bounds for scheduling overheads, single memory access latency, task execution and accessing parameters are not safe. Note that the second case cannot be excluded in our experiments, since *all* mentioned parameters were acquired through measurements. For the deployment of a safety-critical system, more rigorous methods would need to be applied.

- **Availability**. It expresses how many computational resources are available, if needed for incremental design, and it is defined as:

$$A := (16 - m_a) + m_a \cdot \frac{\sum_{\forall f \in \mathcal{F}} \left(L_f - \sum_{k=1}^{2} barriers(f, \text{LO})_k \right)}{H}, \qquad (5.5)$$

where m_a is the number of active cores (implementing the schedule) out of the 16 processing cores of an MPPA-256 cluster and H is the period of the FTTS scheduling cycle. Availability is a combined expression of the number of currently inactive cores and the aggregate portion of time when all active cores are idle, which happens in each frame between the completion of the last sub-frame and the frame end. In an ideal 16-core platform, without scheduling or memory interference overheads and without the constraint of temporal partitioning, $A = 16 - U$ should hold for any application with utilization U. By comparing the availability of our deployed FTTS schedules to the "ideal" availability, we get a measure of the schedulability loss due to the temporal partitioning constraint and the overheads of implementation on a real platform.

We consider 15 experimental configurations, with different combinations and number of instances (replications) of the benchmarks. **Configuration 1** (C1) contains only the high-criticality benchmarks, FMS and ROSACE. **Configurations 2** and **3** contains the FMS, ROSACE and StreamIt benchmarks, with one and two instances of each benchmark, respectively. **Configurations 4–10** contain the FMS, ROSACE and StreamIt benchmarks except "fm", with three to nine instances of each benchmark. "fm" has been excluded due to its large memory footprint, since having more than two instances of this task along with all other benchmarks was exceeding the memory capacity of 2 MB. Similarly, hosting more than 10 instances of each benchmark in a single cluster led to memory allocation problems. For this reason, **Configurations 11–15** contain nine instances of the FMS, ROSACE, StreamIt benchmarks and respectively $\{1, 2, 3, 4, 5\}$ instances of the synthetic benchmarks, which have a very low memory footprint. In configurations with multiple instances of the same benchmarks, each instance is regarded as an individual task, with its own code, acting on its own private data, to avoid race conditions and erroneous behavior.

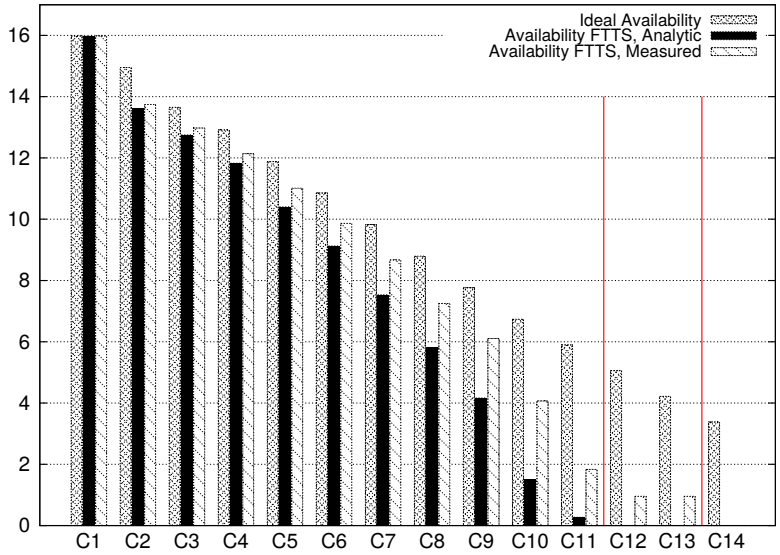
The number of tasks and the utilization of each configuration are shown in Table 5.2. Columns 4 and 5 of the table show the number of active cores in the obtained schedules from the FTTS optimization framework. The optimizer was able to find admissible FTTS schedules for configurations C1 to C11 for OS deployment and C1 to C13 for BM deployment. For the remaining configurations, it returned the best found solutions even if admissibility under the timing analysis of Section 5.6 could not be guaranteed. All FTTS schedules have a cycle period $H = 40$ ms, consisting of 8 frames with a fixed length of 5 ms each. Note that for configurations with low utilization, the OS schedule employs one more active core than the respective BM schedule. This is because in OS mode processing core PE1 is used exclusively by the scheduler thread, whereas in BM mode this core can be used for task scheduling. This explains also the higher achievable utilization in BM mode.

Conf.	Tasks	U	Cores OS	Cores BM	Frame Viol. OS	Frame Viol. BM
C1	16	0.02	2	1	0%	0%
C2	25	1.05	3	2	0%	0%
C3	50	2.10	4	3	0%	0%
C4	72	3.09	5	4	0%	0%
C5	96	4.12	6	5	0%	0%
C6	120	5.15	7	6	0%	0%
C7	144	6.18	9	8	0%	0%
C8	168	7.20	11	9	0%	0%
C9	192	8.23	12	11	0%	0%
C10	216	9.26	15	13	0%	0%
C11	218	10.10	16	15	0%	0%
C12	220	10.94	16 (*)	16	0%	0%
C13	222	11.78	16 (*)	16	0%	0%
C14	224	12.62	16 (*)	16 (*)	99.9%	0%
C15	226	13.46	16 (*)	16 (*)	-	99.9%

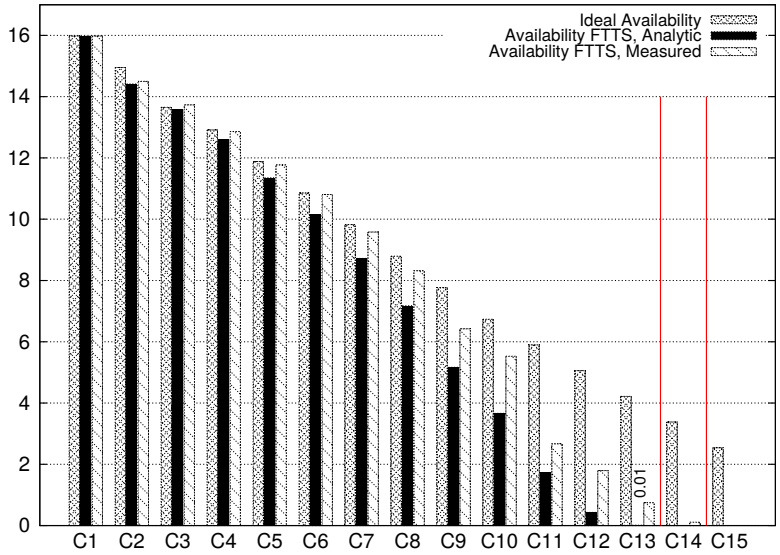
Table 5.2: Benchmark configurations and deployment on the MPPA-256. The FTTS schedules with (*) were deemed *inadmissible* at design time.

Using the MPPA-256 runtime environment, we deployed the best found FTTS schedule (even if it was deemed inadmissible by the optimizer) for each configuration and each execution mode (OS or BM) on one compute cluster. Each schedule was executed for 10,000 scheduling cycles, for a total duration of approximately 7 minutes. The last two columns of Table 5.2 show the ratio of frame violations detected during deployment. Figure 5.6 illustrates the availability metric. For comparison purposes, three computations of availability are depicted: (i) the "ideal" availability, considering no overheads, interferences or partitioning constraints, which is equal to $16 - U$, (ii) the availability of an FTTS schedule, computed by the FTTS optimization framework (Eq. (5.5)) based on our WCRT analysis, and (iii) the minimum availability of a deployed FTTS schedule based on the profiling data of 10,000 scheduling cycles. In case (iii), L_f and $\sum_{k=1}^{2} \mathit{barriers}(f, \mathrm{LO})_k$ are substituted in Eq. (5.5) by the actual frame lengths and the completion time of the last sub-frames in each scheduling cycle.

Frame Violations. For the FTTS schedules that were deemed admissible offline, i.e., up to C11 (OS) and up to C13 (BM), there was *no* frame violation at runtime. Frame violations occurred only for inadmissible schedules at high utilizations (C14, C15 in Table 5.2). For admissible BM schedules, we detected in total 9 HI sub-frame overruns over 130,000 scheduling cycles (1,040,000 frames), leading to 9 skipped LO sub-frames (0.009‰). Since sub-frame overruns occurred only in BM mode, we consider an unusually high overhead of the scheduling primitives in BM mode as the most probable cause. Note that the runtime reaction to the



(a) Operating System Implementation.



(b) Bare-metal Implementation.

Figure 5.6: Availability of different configurations on the MPPA-256.

HI sub-frame overruns prevented frame violations, resp. a practically inadmissible schedule, which supports the claim for adaptive temporal partitioning.

By comparing the analytic versus measured availability in Figure 5.6, we observe that the pessimism of timing analysis increases with increasing utilization, especially for OS schedules. Note that no admissible FTTS schedules could be found for configurations C12 and C13 (OS), although the best found solutions were deployed successfully without any frame violations. Namely, the maximum achievable

utilization with guaranteed schedulability is 10.10 (63.1%, C11), while a utilization of 11.78 (73.6%, C13) is practically possible without frame violations. In our timing analysis the pessimism can stem from (i) over-estimated task execution parameters, (ii) over-estimated memory access latency, (iii) over-estimated scheduling overheads, (iv) the assumption that every memory access can be delayed by simultaneous accesses from all other cores to the same bank (if this rarely happens in practice). The very low variance in the statistical distribution of measured task execution parameters (Figure 5.5) and the measured memory access latency (benchmark of Section 5.6.2) provide evidence against the first two sources. In contrast, the very large variance in the statistical distribution of the scheduling overheads, especially o_{comm} (Figure 5.3(c)), shows their important effect on the analysis pessimism (in the analysis we consider only the upper bound, which was encountered extremely rarely in the measurements). The role of overhead o_{comm} explains partly why pessimism increases with an increasing number of active cores (so does the variance of o_{comm}) and why BM schedules suffer less pessimism compared to OS schedules. Besides scheduling overheads, the assumption of maximal interference on the same memory bank leads inevitably to increasing pessimism as the number of cores (bank masters) increases. This is however a necessary assumption for safe WCRT analysis for a shared memory with interleaved address mapping.

Maximum Achievable Schedulability. For evaluating the schedulability loss that is caused by temporal partitioning and the runtime overheads and interferences, we compare the "ideal" to the empirical availability in Figure 5.6. As expected, the difference between the two metrics is higher for OS than for BM schedules. This is because in BM mode, there is one more processing core available for running worker threads and the worst-case runtime scheduling overheads are significantly lower. These two factors lead to a higher availability of the platform, closer to the ideal bound. The maximum achievable utilization of 11.78 (73.6%, C13) with guaranteed schedulability or 12.62 (78.9%, C14) for a practically admissible solution is a significant result of this work. The comparison to previous works, e.g.,[BDN+16], where the maximum achievable utilization is 14.3% on 14 cores (without considering runtime overheads), shows that adaptive temporal partitioning is a viable solution for the efficient, yet predictable implementation of safety-critical systems on existing many-core architectures.

5.8 Summary

This chapter presented a runtime environment for adaptive temporal partitioning on many-core platforms, with focus on the Kalray MPPA-

256 Andey processor. Applicability was demonstrated based on the FTTS scheduling policy for periodic mixed-criticality applications (Chapter 3). We proposed alternative implementations of the FTTS scheduling primitives and evaluated them w.r.t. runtime overhead. Additionally, we proposed a worst-case response time analysis methodology for evaluating the admissibility of FTTS schedules, by accounting for scheduling overheads and the interference of co-running tasks on the shared resources of a compute cluster. Using industrial-representative benchmarks, we were able to validate runtime adherence to the analytic worst-case response time bounds. We also showed that by optimizing the implementation of the scheduling primitives, an effective utilization of 73.6% (analytically guaranteed) or 78.9% (empirically admissible) can be achieved on the 16 cores of an MPPA-256 cluster. This suggests that adaptive temporal partitioning is a promising solution for efficient and predictable safety-critical application deployment on many-cores, since it enables sufficient isolation among applications with different criticality, yet at a low schedulability cost.

6

Conclusion and Outlook

This chapter summarizes the contributions of this thesis and outlines possible future research directions.

6.1 Contributions

Driven by the need to reduce the cost, size and power dissipation of embedded electronics, systems in safety-critical domains become increasingly mixed-critical, i.e., integrating applications with different safety criticality levels. With a constantly increasing computational demand (consider, for instance, the required functionalities for autonomous driving in next-generation cars), deployment of mixed-criticality systems on multi-core platforms is the next milestone for industrial safety-critical applications. The design and certification of mixed-criticality multi-core systems comes with certain challenges though. These are mainly related to the utilization of shared platform resources and the difficulty in characterizing the temporal interference of concurrently executed applications due to resource contention.

The aim of this thesis is to show that the deployment of mixed-criticality applications on timing-compositional multi-core processors is indeed feasible, in a way that enables efficient resource utilization *and* real-time guarantees. To this end, the main contributions of the thesis are:

- We proposed the Isolation Scheduling (IS) model which enforces interference-free scheduling of applications with different safety criticality, by partitioning the platform temporally and allowing only tasks with the same criticality to execute at any time. The IS model is applicable not only to mixed-criticality systems, but to any system with task classes among which resource

interference should be avoided. By introducing the IS-Server policy and exploring approaches for optimal task to processing core partitioning (applicable to any partitioned IS policy), we showed experimentally that Isolation Scheduling does not incur significant schedulability losses compared to existing mixed-criticality multi-core policies, even if those were designed for efficiency rather than isolation among criticality levels.

- For reducing the complexity of intra-criticality interference analysis, we proposed the IS-compliant FTTS policy, based on flexible time-triggered, partitioned and non-preemptive scheduling. Under FTTS, we were able to provide the first unified WCRT analysis for cluster-based manycores, such as the Kalray MPPA-256, in which accesses to a shared (cluster) memory can be delayed by accesses from concurrently executed tasks in the cluster or by incoming traffic from a network-on-chip (NoC). Our approach accounts for interference on the shared memory and on the NoC routers/links. Additionally, we proposed design optimization approaches for partitioning tasks to processing cores and task data/instructions to memory banks with the purpose of minimizing intra-criticality interference and achieving an overall efficient resource utilization.
- For increasing the accuracy of intra-criticality interference analysis and enabling scalable WCRT analysis under different resource arbitration policies, we introduced a state-based analysis approach. The system is modelled with timed automata and model checking is applied to derive safe and tight WCRT bounds. The novelty of our approach compared to existing model checking approaches for multicores lies in the adoption of a dedicated execution model (superblock) and in an abstraction of the execution and resource accessing behavior of several interfering cores in the form of an arrival curve. These two choices enabled unprecedented scalability. A comparison of the WCRT estimates against results from hardware simulations and analytic approaches confirmed further that the gain in scalability does not come at the cost of analysis accuracy.
- Finally, to demonstrate the applicability of Isolation Scheduling on commodity platforms, we implemented the FTTS policy on the Kalray MPPA-256 processor. Our previous WCRT analysis was extended to capture besides the temporal effects of interference on the shared cluster memory, also the (measured) runtime overheads of the FTTS scheduling primitives. An empirical evaluation with real-world benchmarks validated runtime adherence to the analytic WCRT bounds and a maximal guaranteed utilization of 73.6% on 16 cores. This result promotes Isolation Scheduling as a worth-considering solution for the efficient and predictable implementation of industrial-size mixed-criticality systems.

6.2 Possible Future Directions

The contributions of the thesis represent an important step towards efficient and timing-predictable implementation of mixed-criticality systems on resource-sharing multicores. Nevertheless, there exists potential for further improvements and extensions. The following list details possible directions for future research.

Extension of Isolation Scheduling to Multiple Clusters
The proposed IS-compliant policies, e.g., FTTS in Chapter 3, target cluster-based many-core architectures, in which processing cores are grouped into shared-memory clusters and inter-cluster communication is supported by a network-on-chip. Our WCRT analysis and design optimization methods were developed with focus on a single cluster although global knowledge on the NoC data flows was assumed. A straightforward extension of FTTS and similar policies to multiple clusters would require that the executed tasks in any cluster at any time are characterized by the same criticality level, in order to avoid inter-criticality interference e.g., on the NoC or the external memory. This constraint, however, may be too restrictive and inter-cluster synchronization can be too costly, thus resulting in severe system under-utilization. The relative benefits and drawbacks of enforcing either global temporal isolation across all clusters or local temporal isolation within each cluster combined with a deterministic, e.g., time-triggered access schedule on the NoC and external memory [BDN+16, PMN+16], need therefore to be investigated.

With regards to design optimization and WCRT analysis, an extension to multiple clusters would introduce several open questions. For instance, how can we schedule the NoC (DMA) transfers and choose the flow regulation parameters and routing such that the requested data are available on time (thus, minimizing blocking delays in tasks' execution), interference on the routers is minimized and no deadlocks are possible on the NoC? How can WCRT analysis (in a single cluster) be extended to account for the inter-core interference not only on the cluster memory, but also on the NoC Tx (transmit) interface? How can we efficiently map applications/tasks to compute clusters considering their criticality level, computation and memory requirements, existing data dependencies? Potential solutions to individual problems, e.g., selection of NoC flow regulation parameters for the avoidance of deadlocks [dDYvAG14], CPU-DMA co-scheduling on multicores aiming at minimizing task response times [YPB+12, WP14, AP14, AWP15], NoC aware real-time scheduling [CDPB+14, SRM14, AJF+16] have been recently proposed. However, the integration of such solutions and the development of a holistic design optimization and schedulability analysis framework for cluster-based manycores seems challenging and requires further research.

Spatial Isolation among Criticality Levels
Besides temporal isolation, industrial standards typically require mechanisms for spatial isolation [ARI03], such that applications with different criticality levels do not "pollute" the address space of each other. Spatial isolation has been outside the scope of this work. However, it is interesting to notice that adding this requirement could lead to contradictory objectives during design optimization. Suppose, for instance, that memory bank partitioning is applied as a means to achieve spatial isolation. For temporal isolation under the IS model, the presented design optimization approach in Chapter 3 aims to maximally distribute the memory blocks of tasks with the same criticality level to disjoint memory banks, such that interference during their execution is minimized. This likely results in a memory configuration, in which the allocated blocks in every bank belong to tasks with different criticality levels (since these tasks cannot be executed concurrently and hence interfere). For spatial isolation, however, it is preferable to avoid bank sharing among tasks with different criticality level. Similarly, consider the exploitation of a cluster-based architecture for spatial isolation. By partitioning applications with different criticality levels to disjoint compute clusters, we could define an isolated address space (shared cluster memory) per application. However, such a design does not necessarily preserve temporal isolation, since applications with different criticality can delay each other e.g., on the NoC or the external memory. The trade-off between temporal and spatial isolation needs to be further studied, and design optimization approaches that account for the potentially contradictory objectives need to be proposed.

Benchmarking of Multicores w.r.t. Worst-Case Timing Behavior
For providing safe WCRT bounds on resource-sharing multicores, accurate models of the architecture and all resource arbitration policies are necessary. However, such models are often not disclosed by the vendors or only limited information is available. This can lead researchers to the adoption of hardware models which do not reflect realistic systems, as discussed in [KdNA+14]. Although there are several benchmarks suites for shared-memory systems, e.g., PARSEC [Bie11], aiming to evaluate the average-case performance of new platforms and operating systems, to the best of our knowledge there exist no (platform-independent) benchmark suites for evaluating the *worst-case* timing behavior on resource-sharing multicores. Such benchmarks could be configured, for instance, to generate maximal interference on the shared memory (by exhibiting no locality in their accesses, so that cache misses are continuously triggered) or to maximally stress the NoC (by using different flow configurations depending on the NoC topology, routing policy, flow control). Our experience with implementing low-level hardware benchmarks for the Kalray MPPA-256 has shown that

this is a challenging and error-prone procedure, which currently needs to be repeated for every new target platform (if the new platform features different shared resources). The existence of "worst-case" benchmarks or commonly accepted guidelines on how to design them would ease the characterization of a platform w.r.t. timing predictability, would enable the derivation of realistic resource models (e.g., inference of arbitration policy, resource access latency under zero/maximal interference) and hence, would enable safe and accurate WCRT analysis for commercial-off-the-shelf platforms. For the inference of the worst-case timing parameters for resource accessing, even learning procedures could be considered. We believe that benchmarking of resource-sharing multicores for revealing (and bounding) their worst-case timing behavior is an often neglected practical problem, which requires however a deeper understanding.

Security
Besides safety considerations, industrial certification standards often pose requirements regarding security and the integration of applications with different security levels. System design concepts, such as temporal and spatial isolation, which are commonly used in mixed-criticality systems for fulfilling safety requirements can be applied (but do not suffice) to achieve a secure design. Security requires controlled information flow among applications [Rus81] and a multi-layer design approach, such that an attacker has to circumvent multiple security barriers before acquiring access to the protected assets of a system [PDK+15]. Implementing mixed-criticality applications on multicores introduces new challenges in terms of security. The impact of shared memory controllers and I/O devices on security has been studied in [NP12, MSTP14, PDK+15]. Additionally, recent works have demonstrated how covert channels based on shared last-level cache can be established on commodity multicores [YYG14, MNHF15, YST16]. Integrated solutions for safe and secure implementation of mixed-criticality multi-core systems will be required for addressing such challenges. To this end, extensions of the Isolation Scheduling model for security could be considered and evaluated in future work.

Bibliography

[ABB09] J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[AD90] R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In M. Paterson, editor, *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[ADLD14] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis. Evaluation of cache partitioning for hard real-time systems. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 15–26, 2014.

[AEPR08] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *21st International Conference on VLSI Design (VLSID)*, pages 103–110, 2008.

[AJF+16] L. Abdallah, M. Jan, C. Fraboul, et al. Reducing the contention experienced by real-time core-to-i/o flows over a tilera-like network on chip. In *28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 86–96, 2016.

[AM10] K. Altisen and M. Moy. ac2lus: Bringing SMT-solving and abstract interpretation techniques to real-time calculus through the synchronous language Lustre. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[AP14] A. Alhammad and R. Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *14th International Conference on Embedded Software (EMSOFT)*, pages 20:1–20:10, 2014.

[ARI03] ARINC. ARINC 653-1 avionics application software standard interface. Technical report, 2003.

[ARM] ARM. Cortex-a17 processor. `http://www.arm.com/products/processors/cortex-a/cortex-a17-processor.php`.

[aut] AutoSAR. Release 4.0. `http://www.autosar.org`.

[AWP15] A. Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 285–296, April 2015.

[BB14] S. Baruah and A. Burns. Achieving temporal isolation in multiprocessor mixed-criticality systems. In *Proc. 2nd Workshop on Mixed Criticality Systems (WMC)*, page 21, 2014.

[BBB09] E. Bini, M. Bertogna, and S. Baruah. Virtual multiprocessor platforms: Specification and use. In *30th Real-Time Systems Symposium (RTSS)*, pages 437–446, 2009.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[BBD+12] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 145–154, 2012.

[BCGM99] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[BCLS14] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BD16] A. Burns and R. Davis. Mixed criticality systems - A review. 2016.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT)*, number 3185 in LNCS, pages 200–236. Springer–Verlag, 2004.

[BDN+16] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, 2016.

[BEG16] S. Baruah, A. Easwaran, and Z. Guo. Mixed-criticality scheduling to minimize makespan. 2016.

[BF11] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *32nd Real-Time Systems Symposium (RTSS)*, pages 3–12, 2011.

[BFB15] A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2015.

[Bie11] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[BLS10] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–22, 2010.

[BY04] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.

[BYPC12] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 300–309, 2012.

[CB10] D. B. Chokshi and P. Bhaduri. Performance analysis of flexray-based systems using real-time calculus, revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 351–356, 2010.

[CDPB+14] T. Carle, M. Djemal, D. Potop-Butucaru, R. De Simone, and Z. Zhang. Static mapping of real-time applications onto massively parallel processor arrays. In *14th International Conference on Application of Concurrency to System Design (ACSD)*, pages 112–121, 2014.

[CEN03] CENELEC. EN 50129. railway applications - communication, signalling and processing systems - safety related electronic systems for signaling, 2003.

[cer] European Commission's 7th framework programme: Certification of real-time applications designed for mixed criticality (CERTAINTY). `www.certainty-project.eu`.

[Cer14a] Certainty. D8.3 - validation results. Technical report, 2014.

[Cer14b] Certification Authorities Software Team. CAST-32 position paper. Federal Aviation Administration / European Aviation Safety Agency, 2014.

[CFP14] B. Cilku, B. Fromel, and P. Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 147–151, 2014.

[Cha00] C.-S. Chang. *Performance Guarantees in Communication Networks*. Springer, 2000.

[CJDR00] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference (DAC)*, 2000.

[CLB+06] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. Litmus^{rt} : A testbed for empirically comparing real-time multiprocessor schedulers. In *27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 111–126, 2006.

[Com10] I. E. Commission. IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[Cru91] R. L. Cruz. A calculus for network delay. i. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[DAN+11] D. Dasari, B. Anderssom, V. Nelis, S. Petters, A. Easwaran, and J. Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1068 –1075, 2011.

[DAN+13] D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 39–48, 2013.

[dDAB+13] B. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. de Massas, F. Jacquet, S. Jones, N. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC*, pages 1–6, 2013.

[dDvAPL14] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 97:1–97:6, 2014.

[dDYvAG14] B. de Dinechin, D. Y., D. van Amstel, and A. Ghiti. Guaranteed services of the noc of a manycore processor. In *International Workshop on Network on Chip Architectures (NoCArc)*, 2014.

[DE10] J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *4th ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 155–162, 2010.

[DFG+14] G. Durrieu, M. Faugere, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS)*, 2014.

[EAS11] EASA. Certification memorandum-development assurance of airborne electronic hardware (chapter 9). CM EASA CM-SWCEH - 001 Issue 01, 2011.

[EBSA+11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, 2011.

[eem] EEMBC 1.1 Embedded Benchmark Suite. `http://www.eembc.org/benchmark/automotive_sl.php`.

[ETSE14] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2014.

[EY13] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems*, 50(1):48–86, 2013.

[FDNG+09] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, and P. Gai. Time and memory tradeoffs in the implementation of AUTOSAR components. In *Design, Automation, Test in Europe Conference (DATE)*, pages 864 –869, 2009.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[Fis13] S. Fisher. Certifying applications in a multi-core environment: The world's first multi-core certification to sil 4. *SYSGO white paper*, 2013.

[fle] FlexRay Communications System Protocol Specification, Version 2.1, Revision A. `http://www.flexray.com/`.

[FLY14] J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 151–159, 2014.

[GAG13] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 525–530, 2013.

[GAGC16] S. Goossens, B. Akesson, K. Goossens, and K. Chandrasekar. Memory controllers for mixed-time-criticality systems. Springer, 2016.

[GELP10] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In

10th International Workshop on Worst-Case Execution Time Analysis (WCET), pages 101–112, 2010.

[GGDY14] C. Gu, N. Guan, Q. Deng, and W. Yi. Partitioned mixed-criticality scheduling on multiprocessor platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2014.

[GLST12] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *10th ACM International Conference on Embedded Software (EMSOFT)*, pages 63–72, 2012.

[GSHT13] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *11th ACM International Conference on Embedded Software (EMSOFT)*, pages 1–15, 2013.

[HBHR11] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. Cama: A predictable cache-aware memory allocator. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–32, 2011.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 108–109. IEEE, 2010.

[HGA+15] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele. An isolation scheduling model for multicores. In *Real-Time Systems Symposium (RTSS)*, pages 141–152, 2015.

[HGL] H.-M. Huang, C. Gill, and C. Lu. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. *ACM Trans. Embedded Computing Systems*, 13(4s):126:1–126:25.

[HGST14] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. In *ASP-DAC*, pages 125 – 130, 2014.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis-The SymTA/S Approach. *IEEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[HHK01] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, volume 2211 of *LNCS*, pages 166–184. 2001.

[HKM+12] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *18th Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 197–208, 2012.

[HP16] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[HRK12] A. Hattendorf, A. Raabe, and A. Knoll. Shared memory protection for spatial separation in multicore architectures. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 299–302, 2012.

[HRW13] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis-definition and challenges. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2013.

[Inf] A. A. Informatik. aiT software suite. `http://www.absint.com/`.

[int] Intel-core i7 processor for lga2011 socket: Datasheet, vol. 1. `http://www.intel.com/content/www/us/en/processors/core/4th-gen-core-i7-lga2011-datasheet-vol-1.html`.

[iso11] ISO 26262, Road Vehicles - Functional Safety, 2011.

[JQA+14] J. Jalle, E. Quiñones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *Real-Time Systems Symposium (RTSS)*, pages 207–217, 2014.

[KAZ11] O. R. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1051–1059, 2011.

[KBL+15] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 317–326, 2015.

[KdNA+14] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014.

[KFM+11] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3 –12, 2011.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kir89] D. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Real Time Systems Symposium (RTSS)*, pages 229–237, 1989.

[KKR13] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 80–89, July 2013.

[KLSP10] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek. Operation and data mapping for cgras with multi-bank memory. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 17–26, 2010.

[KNP+14] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2014.

[KWC+16] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, et al. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[LB12] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 166–175, 2012.

[LBT01] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer, 2001.

[LCX+12] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 367–376, 2012.

[Lee08] E. A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[LFS+10] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, 2010.

[LMJ+09] Z. Lu, M. Millberg, A. Jantsch, A. Bruce, P. van der Wolf, and H. T. Flow regulation for on-chip communication. In *Design,*

Automation & Test in Europe Conference & Exhibition (DATE), pages 578–581, 2009.

[LPG+14] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee. Mc-fluid: Fluid model-based mixed-criticality scheduling on multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 41–52, 2014.

[LPT10] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, 14(3):193–227, 2010.

[LPT12] K. Lampka, S. Perathoner, and L. Thiele. Component-based system design: analytic real-time interfaces for state-based component implementations. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2012.

[LWYP99] K. G. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–198, 1999.

[LYGY10] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 339–349. IEEE Computer Society, 2010.

[MBF+12] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications. In *49th Annual Design Automation Conference (DAC)*, pages 1137–1142, 2012.

[MDB+13] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, April 2013.

[MEA+10] M. Mollison, J. Erickson, J. Anderson, S. Baruah, J. Scoredos, et al. Mixed-criticality real-time scheduling for multicore systems. In *10th International Conference on Computer and Information Technology (CIT)*, pages 1864–1871, 2010.

[MFC01] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Real-Time Technology and Applications Symposium (RTAS)*, pages 75–84, 2001.

[MFXJ10] W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware cooperative dram bank partitioning for chip multiprocessors. In *Network and Parallel Computing*, volume 6289 of *LNCS*, pages 329–343. 2010.

[MNHF15] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection*

of Intrusions and Malware, and Vulnerability Assessment, pages 46–64. Springer, 2015.

[MSTP14] K. Müller, G. Sigl, B. Triquet, and M. Paulitsch. On mils i/o sharing targeting avionic systems. In *10th European Dependable Computing Conference (EDCC)*, pages 182–193, 2014.

[NALS06] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 208–222, 2006.

[NP12] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *9th European Dependable Computing Conference (EDCC)*, pages 132–143, 2012.

[NSE09] M. Negrean, S. Schliecker, and R. Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Design, Automation, Test in Europe Conference (DATE)*, pages 524–529, 2009.

[p40] Freescale p4080 processor documentation. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080`.

[Pat12] R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 309–320, 2012.

[PBA+14] D. Potts, R. Bourquin, L. Andresen, J. Andronick, G. Klein, and G. Heiser. Mathematically verified software kernels: Raising the bar for high assurance implementations. 2014.

[PBB+11] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, 2011.

[PBCS08] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Real-Time Systems Symposium (RTSS)*, pages 221–231, 2008.

[PD14] M. Paulitsch and K. Driscoll. *Industrial communication technology handbook. Chapter 48 SAFEbus*. CRC Press, 2014.

[PDK+15] M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench, and J. Nowotsch. Mixed-criticality embedded systems – a balance ensuring partitioning and performance. In *Euromicro Conference on Digital System Design (DSD)*, pages 453–461, 2015.

[PLT11] S. Perathoner, K. Lampka, and L. Thiele. Composing heterogeneous components for system-wide performance analysis. In *Design, Automation and Test in Europe (DATE)*, pages 842–847, 2011.

[PMN+16] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. Temporal isolation of hard real-time applications on many-core processors. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[PPE+08] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[PQnC+09] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. volume 37, pages 57–68, 2009.

[PSC+10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 741–746, Dresden, Germany, 2010.

[PSG+14] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE case study: From Simulink specification to multi/many-core execution. In *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318, 2014.

[QLD] Y. Qian, Z. Lu, and W. Dou. Analysis of communication delay bounds for network on chips.

[QLD10] Y. Qian, Z. Lu, and W. Dou. Analysis of worst-case delay bounds for on-chip packet-switching networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(5):802–815, 2010.

[Qua] Qualcomm. Snapdragon 820 specification. `https://www.qualcomm.com/products/snapdragon/processors/820`.

[RAEP07] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 49–60, 2007.

[RDK+00] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

[RLP+11] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *7th IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, 2011.

[RLSS10] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *47th Design Automation Conference*, pages 731–736. ACM, 2010.

[RTC12] RTCA. DO-178C/ED-12C software considerations in airborne systems and equipment certification, 2012.

[Rus81] J. M. Rushby. *Design and verification of secure systems*, volume 15. ACM, 1981.

[Rus99] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical Report NASA Contractor Report CR-1999-209347, NASA Langley Research Center, 1999.

[Sam] Samsung. Exynos 8 octa (8890) specification. `http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mod_ap/8890/`.

[SCM+14] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, et al. Single core equivalent virtual machines for hard real-time computing on multicore processors. Technical report, University of Illinois at Urbana-Champaign, 2014.

[SCT10] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[SEL08] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 181–190, 2008.

[SGH+15] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele. Mixed-criticality runtime mechanisms and evaluation on multicores. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 194 – 206, 2015.

[SGTG12] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 155–165, 2012.

[SL03] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

[SL04] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Real-Time Systems Symposium (RTSS)*, pages 57–67, 2004.

[SM08] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *45th Annual Design Automation Conference (DAC)*, pages 300–303, 2008.

[SNE09] S. Schliecker, M. Negrean, and R. Ernst. Response time analysis on multicore ecus with shared resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, 2009.

[SNE10] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation, Test in Europe Conference (DATE)*, pages 759–764, 2010.

[SPC+10] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 332–337, 2010.

[SPC+11] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–222, 2011.

[SRL+11] A. Simalatsar, Y. Ramadian, K. Lampka, S. Perathoner, R. Passerone, and L. Thiele. Enabling parametric feasibility analysis in real-time calculus driven performance evaluation. In *14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 155–164. ACM, 2011.

[SRM14] M. Shekhar, H. Ramaprasad, and F. Mueller. Network-on-chip aware scheduling of hard-real-time tasks. In *9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 141–150, 2014.

[SS16] S. Skalistis and A. Simalatsar. Worst-case execution time analysis for many-core architectures with NoC. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 211–227, 2016.

[Sta08] J. A. Stankovic. When sensor and actuator networks cover the world. *ETRI journal*, 30(5):627–633, 2008.

[SW07] R. Sedgewick and K. Wayne. Algorithms and data structures. *Princeton University, COS*, 226, 2007.

[TAED13] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer. IDAMC: A NoC for mixed criticality systems. In *19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 149–156, 2013.

[Tak12] H. Takada. Introduction to automotive embedded systems. `http://estc.dsr-company.com/images/b/b5/Automotive-embedded-systems.pdf`, 2012.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. Intl. Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196, 2002.

[TPM+15] P. Tendulkar, P. Poplavko, J. Maselbas, I. Galanommatis, and O. Maler. A runtime environment for real-time streaming applications on clustered multi-cores. Technical Report TR-2015-6, Verimag Research Report, 2015.

[TS09] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *7th ACM international conference on Embedded software (EMSOFT)*, pages 127–136, 2009.

[TS12] T. Tian and C.-P. Shih. Software techniques for shared-cache multi-core systems. *Intel Software Network*, 2012.

[TSP11] D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Real-Time Systems Symposium (RTSS)*, pages 24–33, 2011.

[TSPS12] D. Tamas-Selicean, P. Pop, and W. Steiner. Synthesis of communication schedules for ttethernet-based mixed-criticality systems. In *8th IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 473–482, 2012.

[Ves07] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

[WGR+09] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, 2009.

[WHKA13] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 157–167, 2013.

[WKP13] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *Real-Time Systems Symposium (RTSS)*, pages 372–383, 2013.

[WMT06] E. Wandeler, A. Maxiaguine, and L. Thiele. Performance analysis of greedy shapers in real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 444–449, Munich, Germany, 2006.

[WP14] S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86, 2014.

[WT05] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *5th ACM International Conference on Embedded Software (EMSOFT)*, pages 80–89, 2005.

[WT06a] E. Wandeler and L. Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *Asia and South Pacific Conference on Design Automation (ASP-DAC)*, 2006.

[WT06b] E. Wandeler and L. Thiele. Real-time calculus (rtc) toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[WTVL06] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis - a case study. *International Journal on Software Tools for Technology Transfer*, 8(6):649 – 667, 2006.

[YMWP14] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

[Yov98] S. Yovine. Model checking timed automata. In *Lectures on Embedded Systems*, pages 114–152. Springer, 1998.

[YPB+12] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems Journal*, 48(6):681–715, 2012.

[YST16] M. Yan, Y. Shalabi, and J. Torrellas. Replayconfusion: Detecting cache-based covert channel attacks using record and replay. 2016.

[YYG14] P. Yang, Z. Yang, and S. Ge. Establishing covert channel on shared cache architecture. In *2014 10th International Conference on Natural Computation (ICNC)*, pages 594–599, 2014.

[YYP+12] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 299–308, 2012.

[YYP+13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[ZSO+13] J. Zhan, N. Stoimenov, J. Ouyang, L. Thiele, V. Narayanan, and Y. Xie. Designing energy-efficient NoC for real-time embedded systems through slack optimization. In *50th Annual Design Automation Conference (DAC)*, pages 1–6, 2013.

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini and L. Thiele. **An Isolation Scheduling Model for Multicores.** In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS).* San Antonio, Texas, December 2015. (Chapter 2)

G. Giannopoulou, N. Stoimenov, P. Huang and L. Thiele. **Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems.** In *Proceedings of the 13th ACM International Conference on Embedded Software (EMSOFT).* Montreal, Canada, October 2013. (Chapter 3)

G. Giannopoulou, N. Stoimenov, P. Huang and L. Thiele. **Mapping Mixed-Criticality Applications on Multi-Core Architectures.** In *Proceedings of the 2014 Design, Automation and Test in Europe Conference and Exhibition (DATE).* Dresden, Germany, March 2014. (Chapter 3)

G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele and B. D. de Dinechin. **Mixed-Criticality Scheduling on Cluster-Based Manycores with Shared Communication and Storage Resources.** In *Real-Time Systems Journal, Volume 52, Issue 4.* Springer, 2016. (Chapter 3)

G. Giannopoulou, K. Lampka, N. Stoimenov and L. Thiele. **Timed Model Checking with Abstractions: Towards Worst-Case Response Time Analysis in Resource-Sharing Manycore Systems.** In *Proceedings of the 12th ACM International Conference on Embedded Software (EMSOFT).* Tampere, Finland, October 2012. (Chapter 4)

K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu and N. Stoimenov. **A Formal Approach to the WCRT Analysis of Multicore Systems with Memory Contention under Phase-structured Task Sets.** In *Real-Time Systems Journal, Volume 50, Issue 5-6.* Springer, 2014. (Chapter 4)

The following list includes publications that were written during the PhD studies, yet are not part of this thesis.

S. Narayana, P. Huang, G. Giannopoulou, L. Thiele and V. Prasad. **Exploring Energy Saving for Mixed-Criticality Systems on Multi-cores.** In *Proceedings of the 22nd IEEE Real-Time Embedded Technology and Applications Symposium (RTAS).* Vienna, Austria, April 2016.

F. Sutton, R. Da Forno, M. Zimmerling, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel and L. Thiele. **Bolt: A Stateful Processor Interconnect.** In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys).* Seoul, South Korea, November 2015.

F. Sutton, R. Da Forno, M. Zimmerling, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel and L. Thiele. **Demo: Building Reliable Wireless Embedded Platforms using the Bolt Processor Interconnect.** In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys).* Seoul, South Korea, November 2015.

L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez and L. Thiele. **Mixed-Criticality Runtime Mechanisms and Evaluation on Multicores.** In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* Seattle, Washington, April 2015.

P. Huang, P. Kumar, G. Giannopoulou and L. Thiele. **Run and Be Safe: Mixed-Criticality Scheduling with Temporary Processor Speedup.** In *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition (DATE).* Grenoble, France, March 2015.

P. Huang, P. Kumar, G. Giannopoulou and L. Thiele. **Energy Efficient DVFS Scheduling for Mixed-Criticality Systems.** In *Proceedings of the 14th ACM International Conference on Embedded Software (EMSOFT).* New Delhi, India, October 2014.

N. Dhruva, P. Kumar, G. Giannopoulou and L. Thiele. **Computing a Language-Based Guarantee for Timing Properties of Cyber-Physical Systems.** In *Proceedings of the 2014 Design, Automation and Test in Europe Conference and Exhibition (DATE).* Dresden, Germany, March 2014.

P. Huang, G. Giannopoulou, N. Stoimenov and L. Thiele. **Service Adaptions for Mixed-Criticality Systems.** In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC).* Singapore, January 2014.

Curriculum Vitæ

Personal Data

Name	Georgia Giannopoulou
Date of Birth	December 19, 1985
Citizenship	Greek

Education

2011–present	ETH Zurich, Switzerland Computer Engineering and Networks Laboratory Ph.D. under the supervision of Prof. Dr. Lothar Thiele
2009–2011	ETH Zurich, Switzerland M.Sc. ETH in Electrical Engineering and Information Technology
2003–2009	University of Patras, Greece Diploma in Computer Engineering and Informatics (equivalent to M.Sc.)

Professional Experience

2011–present	ETH Zurich, Switzerland Computer Engineering and Networks Laboratory Research and teaching assistant
2010	Siemens Building Technologies, Zug, Switzerland Research and Development intern
2007	University of Patras, Greece Teaching assistant

Awards

March 2014	GI/ITG biennial award for best Master thesis in Measurement, Modelling and Evaluation of Computing Systems (MMB)
2009–2011	Latsis Foundation scholarship for Master studies at ETH Zurich