Scratchpad Memory Management For Multicore Real-Time Embedded Systems

by

Saud Wasly

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

© Saud Wasly 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Nathan W. Fisher Associate Professor Department of Computer Science, Wayne State University
Supervisor:	Rodolfo Pelizzoni Associate Professor Department of Electrical and Computer Engineering, University of Waterloo
Internal Members:	Hiren Patel Associate Professor Department of Electrical and Computer Engineering, University of Waterloo
	Sebastian Fischmeister Associate Professor Department of Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Martin Karsten Associate Professor Department of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

A large part of the content in this dissertation has been previously disseminated through 8 papers [170, 171, 18, 159, 161, 173, 160, 172], either peer-reviewed or under submission. The use of the content from the listed publications in this dissertation has been approved by all co-authors. I am the first author, sole student author and main contributor in [170, 171, 173, 172]. I am a student co-author in [18, 159, 161, 160]; for these collaborative papers, I further highlight the extent of my contribution below.

The first main contribution in this dissertation is an execution and scheduling model for 3-phase real-time tasks to hide access latency to shared resources, such as main memory, through the use of a Direct Memory Access (DMA) engine. Section 3.1 discusses the main concept of the proposed execution model, while Section 3.2 presents a fixed-priority non-preemptive partitioned scheduling scheme based on the proposed model. The content of these sections is reproduced from [171]. The proposed scheduling scheme is extended in Section 3.3 to handle software-based scheduling of the DMA engine, and in Section 3.4 to recover from soft memory errors. Content in these sections is reprinted from co-authored papers [159] and [161], respectively. In both cases, my main contribution was to develop the response time analysis of the proposed execution model (Sections IV-B, V, and VII-E in [159] and Sections IV-B, V, and VII-D in [161], which are reproduced in this dissertation).

Chapter 4 presents the realization of the proposed scheme. Section 4.1 discusses the implementation on an FPGA platform. Some of the content of this section is reprinted from [170], specifically, Sections IV and V in [170]. Section 4.2 presents the implementation of the proposed system on a COTS platform, including a complete OS design. The content of this section is reprinted from co-authored papers [159, 161]. I was principally responsible for the schedulability evaluation, presented in Subsection 4.2.3; the rest of the section is reproduced for the sake of completeness.

In Chapter 5, the scheduling scheme is extended to global scheduling. The content of this chapter is reprinted from co-authored paper [18], in which my contribution was to extend the scheduling policy (Section 5.2), bounding the schedule holes (Section 5.3.2), and conducting the evaluation (Section 7.3).

In Chapter 6, an asynchronous inter-task communication model is presented. The content of this chapter is reprinted from co-authored paper [160], in which I was responsible for adapting the model in the proposed execution scheme, developing the worst-case latency analysis and conducting the evaluation, (Sections 6.1, 6.3, and 6.4).

Finally, Chapter 7 presents a new scheduling model for parallel tasks, bundled scheduling, and Chapter 8 presents a predictable inter-core NoC. The content of these chapters is reprinted from [172] and [173], respectively.

Abstract

Multicore systems will continue to spread in the domain of real-time embedded systems due to the increasing need for high-performance applications. This research discusses some of the challenges associated with employing multicore systems for safety critical real-time applications. Mainly, this work is concerned with providing: 1) efficient inter-core timing isolation for independent tasks, and 2) predictable task communication for communicating tasks. Principally, we introduce a new task execution model, based on the 3-phase execution model, that exploits the Direct Memory Access (DMA) controllers available in modern embedded platforms along with ScratchPad Memories (SPMs) to enforce strong timing isolation between tasks. The DMA and the SPMs are explicitly managed to preload tasks from main memory into the local (private) scratchpad memories. Tasks are then executed from the local SPMs without accessing main memory. This model allows CPU execution to be overlapped with DMA loading/unloading operations from and to main memory. We show that by co-scheduling task execution on CPUs and using DMA to access memory and I/O, we can efficiently hide access latency to physical resources. In turn, this leads to significant improvements in system schedulability, compared to both the case of unregulated contention for access to physical resources, and to previous cache and SPM management techniques for real-time systems.

The presented SPM-centric scheduling algorithms and analyses cover single-core, partitioned, and global real-time systems. The proposed scheme is also extended to support large tasks that do not fit entirely into the local SPM. Moreover, the schedulability analysis considers the case of recovering from transient soft errors (bit flips caused by a single event upset) in several levels of memories, that cannot be automatically corrected in hardware by the ECC unit. The proposed SPM-centric scheduling is integrated at the OS level; thus it is transparent to applications. The proposed scheme is implemented and evaluated on a FPGA platform and a Commercial-Off-The-Shelf (COTS) platform.

In regards to real-time task communication, two types of communication are considered. 1) Asynchronous inter-task communication, between either sequential tasks (single-threaded) or parallel tasks (multi-threaded). 2) Intra-task communication, where parallel threads of the same application exchange data. A new task scheduling model for parallel tasks (Bundled Scheduling) is proposed to facilitate intra-task communication and reduce synchronization overheads. We show that the proposed bundled scheduling model can be applied to several parallel programming models, such as fork-join and DAG-based applications, leading to improved system schedulability. Finally, intra-task communication is governed by a predictable inter-core communication platform. Specifically, we propose HopliteRT, a lean and predictable Network-on-Chip that connects the private SPMs.

Acknowledgements

First and foremost, I am grateful to Allah for empowering me to complete this thesis. Without his help, I would not have the ability to reach this stage in my life.

I would like to seize this opportunity to thank all the people who made this dissertation possible and my Ph.D. a spectacular experience. My deepest gratitude goes to my advisor, Prof. Rodolfo Pellizzoni. It is hard to describe the uncountable ways in which Rodolfo impacted my life and career. He has shared with me his expertise and always provided constructive guidance. He involved me in exciting projects and valued my suggestions. I truly admire his technical depth and scientific rigorousness. All in all, he has been an irreplaceable advisor and continues to be a precious friend.

I am extremely grateful to the members of my thesis committee: Prof. Nathan Fisher, Prof. Hiren Patel, Prof. Sebastian Fischmeister, and Prof. Martin Karsten. Each of them has supported my work in a peculiar way. I thank Prof. Fisher for taking the time and serving as the external examiner. His feedback was significantly valuable. My encounter with Prof. Fischmeister was the initial trigger to gain interest in the field of real-time systems during my master. I enjoyed his course and admired his regard to the practical aspects of the field. The lectures of Prof. Patel in computer architecture were inspiring. I have learned a lot from his research vision and accurate definitions to research problems. I also have learned a lot from the expertise of Prof. Karsten in operating systems. My encounter with Prof. Karsten inspired me with several research ideas that ended up in this dissertation.

I extend my thanks and appreciation to my co-authors, Ahmed Alhammad, Nachiket Kapre, Rohan Tabish, Renato Mancuso, and Marco Caccamo, with whom I have always enjoyed discussing the research-related problems and received valuable feedback. My thanks also extend to Michael Guo, Muhammad Refaat, Anirudh Kaushik, and Mohamed Hassan, the colleagues who I sadly did not have a chance to write papers with, but their help and feedback are highly appreciated.

My time at Waterloo was made enjoyable due to the many friends who became a part of my life. Special thanks go to my close friend Ali Albishi, with whom I really enjoyed discussing different aspects of life. His enthusiastic personality has been motivational for me to proceed during the tough times.

I am grateful to King Abdulaziz University for their financial support. Without their support, I probably would not have the opportunity to attend a great school like the University of Waterloo.

Foremost, I thank my father, Mohammad, for his endless love and optimism and for being with me at all times with his prayers. You always encourage me to do the best in my life. I am also thankful to my deceased mother, Zara'ah, for rising and teaching me to be a good person. My thanks extend to my brothers and sisters who believed in my cause and always remembered me in their prayers.

I am thankful for my son, Mohammad, and my daughter, Jumanah, for shining our home with their beautiful smiles. My deepest and greatest thanks and gratefulness go to my wife, Hind, for her love, patience, and trust that I was doing the right thing. I could not have finished my Ph.D. without you.

Dedication

Indeed, my prayer, my rites of sacrifice, my living and my dying are for Allah , Lord of the worlds. [Quran 6:162]

To the ones I love:
To my parents, Mohammad and Zara'ah
To my devoted and supportive wife, Hind
To my hero son, Mohammed
To my princess daughter, Jumanah

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Real-time systems require predictable temporal behavior. The system schedules shared physical resources, such as CPU time, to execute tasks of different applications. An application is composed of one or more tasks [98]. Typically, tasks are recurrent: each task produces a potentially infinite sequence of jobs, activated either periodically or sporadically. Applications in real-time systems are classified into safety-critical applications known as hard tasks and less-critical applications known as soft tasks. Each job of a hard task needs to finish its execution and produce its output before the expiration of a strict time (deadline); otherwise, the system might catastrophically fail. Medically-implanted pacemakers and airbag systems in a modern cars are examples of this class of applications. On the other hand, soft real-time tasks might suffer service degradation when a job exceeds the execution deadline. Examples of such soft-tasks include real-time communication in video conferencing. In this dissertation, we focus on hard real-time tasks.

In hard real-time systems, the collection of executed tasks (also known as a task set) requires timing validation. A task set is called feasible if it can be scheduled such that all jobs of all tasks meet their deadlines, under all permissible combinations of job-arrival sequences by the different tasks comprising the system. On a particular computing platform, the system of tasks is said to be A-schedulable with respect to a given scheduling algorithm A, if the algorithm A schedules the system such that all jobs of all tasks will meet all deadlines. A schedulability analysis for scheduling algorithm A accepts as input the specifications of a real-time system, and determines whether the system is A-schedulable or not. An A-schedulability analysis is said to be exact if it correctly identifies all A-schedulable systems, and sufficient if it may fail to identify some A-schedulable systems (it must guarantee, though, that all identified systems are indeed A-schedulable).

An important input to the schedulability analysis is the worst-case execution time (WCET) of each task. It is essential for the WCET estimate of a task to be safe; i.e., the upper bound of the execution time under all circumstances must be captured. While underestimating the WCET can lead to unsafe schedules (missed deadlines), overestimating the WCET can lead to a reduction in the system schedulability and leave the hardware underutilized. It is often desirable for the WCET estimate to be *tight*, i.e., close to the actual worst-case execution time. The tightness of WCET bounds can be highly affected by the underlying platform, including the hardware, the operating system (OS), and the application runtime environment adopted by the programming language. *Predictability* is an important and related term that indicates how tight the WCET estimate is. Specifically, [24] defines predictability as the ratio between the best-case and the worst-case behavior. That is, a component with constant-time behavior is 100% predictable, and this percentage decreases as the difference between the best-case and the worst-case behavior increases. Consequently, when any component in the system behaves in a less predictable manner, a safe but pessimistic performance assumption is needed to compensate for the less-predictable timing estimates. This pessimism can reduce the system utilization and prevents real-time engineers from exploiting the full performance of the hardware.

In the last decade, however, the popularity of multi-core chips has seen an uptrend in the domain of embedded computing. Modern emergent embedded systems, such as self-driving cars and Unmanned Aerial Vehicles (UAV), that heavily utilize parallel multi-threaded applications, such as machine learning, and computer vision, are driving an increasing performance demand in embedded computing in general and in real-time especially. This performance demand is answered by high-performance embedded multi-core chips. However, the introduction of multi-core chips has a disruptive impact on the design of real-time systems: most of the existing design practices in the field cannot be directly applied to multi-core systems. Hence, platform selection for the design of modern safety-critical systems often involves outdated technological solutions or highly inefficient exploitation of current modern hardware [5], such as disabling all but one core. In response, this is driving researchers' efforts to introduce new design methodologies and tools to overcome challenges with multi-core systems, in particular due to the presence of shared hardware resources that affect the predictability of the system.

1.1 Challenges with Multi-Core Systems

The introduction of multi-core systems in the real-time domain is relatively new compared to single core systems. Therefore, many challenges related to multi-core systems are

currently being addressed by the community. In this dissertation, we address two main challenges: 1) contention for access to shared physical resources, such as the memory subsystem of modern Multi-Processor Systems-on-Chip (MPSoC); and 2) the need for data sharing in real-time parallel applications.

1.1.1 Shared Resources and Contention

In industry, it is often the case that different components of the system are provided by several vendors and then integrated together. For more than three decades, a fundamental assumption of the schedulability analysis is that the WCET can be calculated on individual tasks in isolation, i.e., without considering the activity of other cores or hardware components. This simplifies the schedulability analysis of the complete system when tasks are running together. Unfortunately, this assumption is violated in multi-core systems. In a single core processor, tasks do not execute concurrently in a single core system, but rather sequentially, albeit they might be preempted. The sequential execution implies that a task has an exclusive access to all hardware resources such as caches, main memory, and I/Os; and no two tasks can access shared resources simultaneously. However, the situation is different in multi-core systems. Two tasks can run simultaneously on different cores and access shared resources at the same time, thus causing interference to each other. Consequently, the extensive sharing of hardware resources has made analyzing the temporal behavior of real-time applications running in parallel on multi-core systems one of the biggest challenges in the real-time domain today.

Figure 1.1 depicts the implications of contended access to shared resources, main memory in this case. The figure is based on results from slides presented during a keynote speech titled as "Mixed-Criticality Systems - a Journey Embedded in Time and Space" by *M. Paulitsch* at 27th Euromicro Conference on Real-Time Systems (ECRTS15), 2015[1]. The figure shows the performance degradation encountered by a set of concurrently running synthetic benchmarks, designed to maximize delay due to inter-core interference on the Freescale P4080 eight-core MPSoC platform. Specifically, the execution time of the task under analysis is observed first when it runs alone, and then with an increasing number of applications running in parallel on other cores. Figure 1.1(a) focus on showing the access time to main memory under four scenarios: when the task under analysis is preforming read operations and the interfering tasks are also performing read operations ("read-read"); and when the task under analysis is performing read operations and the interfering tasks are performing write operations ("read-write"); and so on for "write-read" and "write-write".

[1]See the presented slides at http://control.lth.se/ecrts2015/files/keynote.pdf

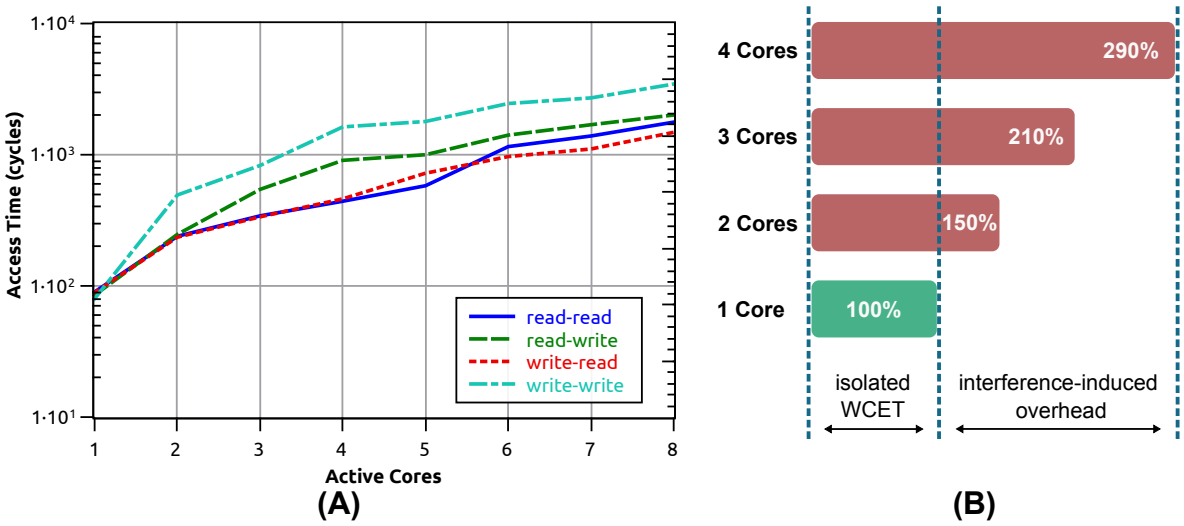


Figure 1.1: (**A**): access time to main memory is growing over proportionally with the number of contending cores. (**B**): the WCET is highly affected by the interference caused by the parallel cores.

It can be observed that the access time to the shared main memory in this platform grows over-proportional to the number of active cores. This performance behavior greatly affects the WCET, which is depicted in Figure 1.1(b). As it can be seen from the figure, with 4 contending cores, the execution time of the task under analysis is inflated by $\simeq$3X. While the slowdown of memory access time is about $\simeq$10X in the case of 4 contending cores, it only leads to $\simeq$3X slowdown in total execution time. This is because, during typical task execution, a portion of the time is spent for computation-only operations that can progress in parallel and often hide the extra time required for non-blocking memory operations. This degraded-performance behavior is not unique to real-time platforms. It has also been shown in [181] that similar effects are observed by running SPEC CPU2006 applications on an Intel Xeon X3565 quad-core processor. The results show that the execution time of one application can increase by 56% compared to the execution time of the same application when run in isolation. Resource contention has also been acknowledged by certification authorities, and it represents a source of concern for the use of multi-core processors in avionics systems [5].

Task isolation can be spacial or temporal. Spacial isolation restricts the ability of software components to access specified hardware components, or subsets of memory resources. Spacial isolation improves system safety and security by limiting each application

or application class to a set of accessible resources with no interaction between unrelated components; i.e., an isolated faulty application cannot affect other applications. On the other hand, temporal isolation protects the timing behavior of the isolated application from other applications running in parallel on other cores; i.e., the performance of the application is independent of the behavior of the other running applications. While contemporary OSs and COTS platforms are already equipped with advanced spacial isolation techniques, they do not provide extensive support to achieve strong temporal isolation among applications.

The current industry trend is to use Commercial-Off-The-Shelf (COTS) multi-core platforms to build hard real-time systems. Unfortunately, such platforms are not designed to support real-time applications, but rather to optimize average-case performance. We argue that to construct a tight WCET analysis for real-time tasks executing in such platforms, the designer should obtain: 1) a detailed model of the hardware components, so that their behavior can be accurately captured; 2) a detailed characterization of all tasks in terms of their accesses to shared hardware resources. Unfortunately, this approach is not practical for at least two main reasons. First, due to manufacturers' intellectual property, the detailed models of the COTS hardware components are often not available or incomplete. Second, assuming accurate knowledge of the entire system workload also faces practical limitations. Particularly, as the number of parallel cores increases, the problem of exhaustively testing all the possible platform/workload configurations combinatorially explodes.

We argue that if tasks are allowed to access shared resources in an unregulated way, the resulting WCET estimates are typically so large as to be practically unusable. Instead, the platform should be engineered to achieve strong temporal isolation between cores, either through mechanisms at the hardware level, the software, or both (hardware-software co-design). Temporal isolation allows the designer to determine system schedulability by composing the WCET of individual tasks. A strong temporal isolation between running tasks would therefore significantly improve the WCET of the tasks and simplify the schedulability analysis of the system.

1.1.2 Data Sharing in Parallel Tasks

With the increased demand for high-performance real-time applications, such as autonomous driving and computer vision [158], real-time designers are looking at parallel applications. Therefore, the real-time community has developed parallel task models, where each task comprises a number of subtasks, i.e., threads. The threads can be activated on different computing nodes simultaneously to run in parallel, and usually share data. In the real-time

literature, applications are broadly categorized into single-threaded applications known as *sequential tasks*, and multi-threaded applications known as *parallel tasks*.

With the increasing demand for more single-application performance, several parallel programming models have been adopted for real-time parallel applications, such as the fork-join and the Directed Acyclic Graph (DAG) models. However, the strong demand for temporal isolation between sequential hard-real-time applications has delayed the adoption of multi-core systems. As discussed in the previous section, task isolation permits to safely utilize multi-core systems for real-time applications, improves predictability, and simplifies analysis. Unfortunately, the temporal isolation mechanisms used for sequential tasks cannot be applied to parallel tasks. This is because we cannot isolate the cores that are communicating. As a matter of fact, existing scheduling schemes for parallel real-time tasks assume no synchronization cost or data sharing. We argue that parallel applications need new scheduling models that relax the aforementioned assumptions and allow for predictable inter-core communication.

1.2 Scope and Contributions of This Work

The goal of this work is to provide a set of solutions at the hardware, software and analysis level, to increase the predictability of both sequential and parallel safety-critical real-time applications on multi-core systems. In contrast to other approaches that target COTS systems only, this work focuses on platform engineering and hardware/software co-design to achieve better predictable performance for safety-critical applications. Our design philosophy it to employ the knowledge of system workload that exists at the scheduling level, and pass required information down to the hardware level to optimize task execution. This methodology significantly simplifies the hardware design and the WCET analysis, at the cost of more complex scheduling policies.

More in details, we provide solutions to 1) avoid access contention to shared resources such as main memory and IO devices, 2) provide predictable access to local memory, and 3) provide predictable inter-core communication. In particular, at the hardware level, we consider Scratchpad Memory (SPM), instead of conventional caches, to implement the local memory for predictability reasons [2]. Furthermore, we propose a lean and predictable Network-on-Chip (NoC) that governs inter-core communication. At the software level, we dynamically manage the local SPMs based on OS support, i.e., SPM management is transparent to the applications. This includes loading tasks into the local SPMs, unloading tasks

[2] As we discuss in Chapter 2, caches can have significant predictability issues.

from the SPMs, and moving data between the local SPMs as needed. The contributions of this work are broadly divided into two parts, 1) task isolation and 2) task communication.

1.2.1 PART(I): Efficient Tasks Isolation

To achieve strong inter-core timing isolation, we propose to adopt a new task execution model based on the existing PRedictable Execution Model (PREM) presented in [128]. The work in [128] is a software solution that relies on compiler techniques to instrument a task's code with extra instructions to divide the task into memory and execution phases. It does that by using the CPU to prefetch the required data into cache during the memory phase; while, in the execution phase, the task executes from cache and does not access the main memory. This execution model avoids contention by scheduling tasks to access to main memory while other tasks are in their execution phases. Unlike the original PREM, we propose to use a Direct Memory Access (DMA) controller to load tasks into the local SPM without stalling the CPU. Our model is based on hardware/software co-design and relies on dynamic management of the SPM as local memory.

Figure 1.2 illustrates the main ideas behind the proposed execution model. Part **(A)** shows a simplified illustration of the architectural requirements of the model. We assume that each core has access to a private SPM and a DMA controller. The SPM is dual ported to allow simultaneous access from the local CPU and the DMA: the CPU and the DMA can access different portions of the local memory concurrently without causing mutual interference. A task is loaded into the local SPM before execution, and the CPU does not need to access main memory while executing the task. Moreover, the SPM is divided into two partitions to allow the DMA to load a task into one partition while the CPU executes another task from the other partition simultaneously. Part **(B)** of the figure shows an arbitrary schedule of three tasks to illustrate the execution model.

Note that the described system can also be realized through caches. This is possible by applying cache partitioning techniques [155, 104] in systems that support cache locking and cache stashing [7]. However, it is still unclear if the proposed scheme will be deterministic when both the CPU and the DMA are active simultaneously. In addition, caches are intended to be self-managed, and require more effort to be managed in software. On the other hand, scratchpad memories, in fact, is simpler to manage and have been proven to provide better energy consumption [28] and temporal isolation when compared to traditional caches [136, 110].

A task is divided into three phases, **load, execution, and unload**. As shown in the figure, the DMA is used to unload any previously loaded task in the targeted partition

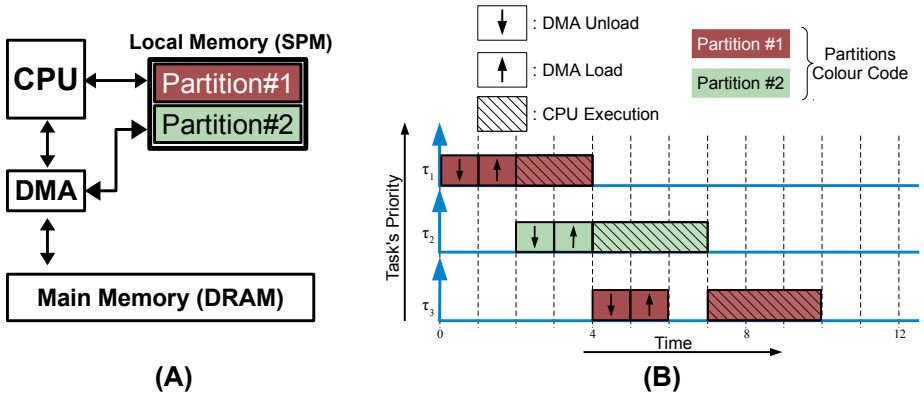


Figure 1.2: (A): Simplified hardware architecture. (B): Example schedule showing how tasks are executed on one core.

before loading a new task. After that, the task can be executed from the local memory with no interference. As depicted, due to the dual partitioning of the local memory, the CPU is kept busy most of the time and access latency to main memory is effectively hidden due to the overlap between executing a task from one partition while loading another task into the other partition. This desirable behavior of completely hiding the access latency to main memory can be observed as long as the execution time of a task is greater than or equal to the load time of the next scheduled task on the other partition in the SPM of the same core. When the execution time is smaller than the load time, the access latency is still partially hidden with an amount equal to the overlapped execution time. The proposed execution model can be applied to both sequential and parallel tasks. In the case of a parallel task, all code and data of the required threads of the task are loaded or unloaded to/from the SPM partitions together. Note that for communicating parallel tasks, we propose an orthogonal solution to handle inter-core communication as we discuss in Section 1.2.2.

The proposed execution model provides four main benefits: 1) it avoids task interference due to contention for access to main memory. 2) It efficiently hides the access latency to main memory. 3) It efficiently utilizes the memory transfer bandwidth as it moves bigger contiguous blocks of data; in particular, sequential transfers in Dynamic RAM (DRAM) are significantly more efficient than random accesses. 4) It improves the tightness of the WCET analysis due to the improved predictability of the platform as tasks are executed out of the local SPMs only.

In regards to I/O handling, we exploit the core specialization available in modern

embedded platforms, such as the Freescale MPC5777M used in our implementation. In short, a dedicated I/O core receives data from I/O devices and buffers them in its local SPM without affecting the main memory bus. Upon task loading on an application core, we then program the DMA to move the needed I/O data from the local SPM of the I/O core to the targeted partition on the application core. Similarly, upon task unloading, we program the DMA to move the produced I/O data from the task's partition on the application core to the corresponding buffer on the I/O core.

From a scheduling point of view, to avoid access contention to physical resources, we adopt a co-scheduling approach. We contribute several dynamic scheduling algorithms based on fixed-priority scheme for a set of sporadic real-time tasks, that efficiently co-schedule processor and DMA execution to hide memory access latency. The proposed algorithms target single-core, partitioned multi-core, and global multi-core scheduling schemes. We demonstrate that we improve processor utilization significantly compared to existing scratchpad and cache management systems in addition to contention-based systems.

The analysis is also extended to cover tasks with large memory-footprint that cannot entirely fit into the local SPM's partition. In essence, a task can be split into multiple segments where each segment can fit into the local memory. The first segment is released like a normal task and the subsequent segment is released after the previous segment finishes. Jobs of the same task are assumed to produce the same number of segments and each segment inherits the priority of the producing task. Note that while a segment of a task is executed by the CPU, the DMA cannot load the next segment of the same task, since a latter segment might depend on a previous segment's data.

Task-splitting is quite popular in practice and supported by several commercial tools. For instance, it is adopted in time-triggered architectures to mitigate the allocation problem of tasks to cycles [45]. The recent work in [132] shows how to break a program into non-preemptive segments through compiler analysis. Moreover, there has been a significant amount of work in literature that proposes techniques to manage the local memory of one task [53, 101, 25] for code and data, including stack and heap. Our focus in this work, therefore, is how to provide a timing guarantee for the scheduled jobs rather than showing how to divide a task into multiple segments.

In addition, a set of scheduling strategies are proposed to recover from detectable transient memory errors. Specifically, we describe how it is possible to recover from bit errors, in both main memory and scratchpad, that are detected but not corrected by the hardware logic via Error-Correcting Code (ECC). The strategy that we follow largely leverages the existing redundancy in the employed multi-phase task model. A minimum amount of additional redundancy is introduced to protect data that do not exist as multiple

copies in the original scheme. Schedulability analysis is extended to take into account the overhead introduced by the proposed recovery mechanisms. The proposed analysis considers different application recovery scenarios and isolates the critical path of the error-handling procedures, which often correspond to entire task re-execution. The result is a fault-tolerant scheduling framework for multi-phase real-time tasks.

The proposed approach has been realized in both an in-house FPGA-based architecture [170], and commercial embedded platform (Freescale MPC5777M) [159]. The execution model and the SPM management logic are integrated at the OS level. In particular, through collaboration with the co-authors of [159], ERIKA[3] [11] RTOS has been extended with a novel operating system design to exploit core specialization and low level resource management policies. To the best of our knowledge, this is the first OS that integrates a scratchpad-based task scheduling mechanism with a schedule-aware I/O subsystem.

The following list is a summary of the contributions in this part:

- SPM-centric system architecture with dynamic management at the OS level.
- Limiting task execution from local memory to enforce timing isolation between cores.
- Extending the PREM execution model to allow for hiding access latency to shared resource.
- A set of SPM-centric scheduling algorithms with sufficient schedulability analysis for partitioned and global task systems.
- Extending the schedulability analysis for tasks with large memory footprint that cannot fit entirely in the local memory.
- Extending the schedulability analysis to account for recovery from memory soft errors that cannot be directly corrected in hardware by the ECC unit.
- FPGA and COTS implementations of the proposed schemes.

1.2.2 PART(II): Predictable Tasks Communication

In regards to task communication, we considered two communication types, inter-task and intra-task communication. Inter-task communication addresses the communication

[3]Erika Enterprise is an open-source RTOS that features a small memory footprint and supports multi-core platforms.

between tasks, whether they are scheduled on the same core or on different cores. In this type of communication, we assume an asynchronous communication model. This means that the previously produced data, by a sender task, can be overwritten if it was not read by the receiver task before. In our proposed model, a sender task's communication data is written to main memory during its unload phase. Similarly, any communication data required by a receiver task is loaded into SPM during its load phase. Therefore, this model does not require inter-core communication in our proposed SPM-centric system: the communication is performed via main memory during the loads and unloads phases of the tasks. This model of communication can be applied to both sequential and parallel tasks, since the communication is scheduled and performed by the DMA used to load/unload the tasks. The OS-level management and scheduling policies are extended to account for inter-task communication. We also provide analytical bounds on the worst-case communication latency for a chain of sender-receiver tasks.

On the other hand, intra-task communication is unique to parallel tasks. This type of communication involves data exchange between the simultaneously running threads of the same parallel task. To facilitate this type of communication, we propose to gang schedule the communicating threads: all currently active threads of the same parallel task are executed concurrently. This policy is motivated by two objectives: 1) to facilitate intra-task communication, and 2) to reduce the synchronization overhead, thus obtaining a more predictable execution of parallel real-time tasks.

Indeed, gang scheduling of parallel tasks has been proved to have significant performance benefits in many cases [60, 77, 150]. Gang scheduling for parallel real-time tasks [81, 55, 64] considers a *rigid* task model, where the number of threads required by an application is assumed to remain constant over its entire execution time. While the rigid model has the benefit of simplicity, it can incur a significant loss of performance by overestimating the computational demand of an application: many parallel applications change their required number of threads during execution.

Hence, we introduce a novel task model, which we call the bundled model, that supports gang scheduling of parallel threads without incurring undue pessimism in modeling the application's demand. In this model, a real-time task is composed of a sequence of bundles, where each bundle is characterized by a known worst-case execution time (WCET) and number of required cores; successive bundles require different numbers of cores. All threads within a bundle are then gang scheduled. We show that the proposed bundled scheduling model can be applied to several existing parallel programming models, such as fork-join and DAG-based applications, and we derive a schedulability analysis for a set of sporadic bundled task based on fixed priorities.

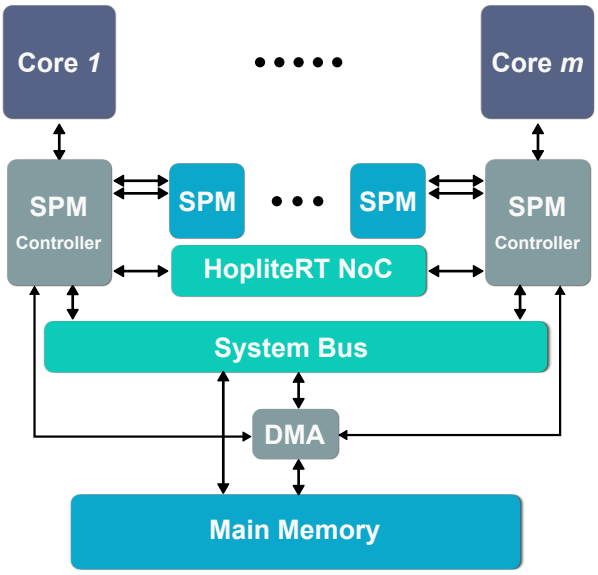


Figure 1.3: Illustration of the proposed system architecture with a predictable interconnect between private SPMs.

A remaining problem is how to bound the delay due to inter-core communication between threads of the same parallel task. In this dissertation, we propose to employ a dedicated real-time inter-SPM interconnect, that can provide tight bounds on worst-case access delay. In this regards, we have developed HopliteRT, a lean and predictable deflection-based Network on Chip (NoC). We then compute the worst-case response time of a parallel task based on the composition of the WCET on the cores and the worst-case communication time on the NoC. Figure 1.3 shows a simplified illustration of the considered system architecture. Note that we still rely on the main system bus for loading and unloading the tasks including inter-task communication as mentioned earlier; the inter-SPM NoC is only dedicated for intra-task communication. Note that, any interconnect with provable real-time bounds can be used instead of HopliteRT. However, aligning with our design philosophy, we are interested in reducing hardware cost, especially since HopliteRT is only targeting inter-core communications.

The following list is a summary of the contributions in this part:

- Asynchronous communication model for inter-task communication via main memory.
- Bundled scheduling: a communication-aware scheduling model for parallel tasks with

intra-task communication.

- Sufficient schedulability analysis for bundled scheduling.
- Extending bundled scheduling for the proposed 3-phase execution model mentioned in PART(I).
- Inter-core real-time NoC with provable latency bounds.

1.3 Structure of the Dissertation

The reminder of this dissertation is organized as follows. First, Chapter 2 reviews the important background concepts and highlights the related work.

Part(I) of the dissertation includes Chapters 3, 4, and 5. Chapter 3 discusses the fundamental concepts of the proposed execution model and scheduling scheme. Specifically, in this chapter we show how to hide access latency to shared resources in partitioned systems, under different ways to handle DMA operations. We also show how to handle tasks with large memory footprint. In particular, we discuss the case of scheduling multi-segment tasks where the size of a task cannot entirely fit into the local SPM. Furthermore, we present an error recovery mechanism with respect to the proposed execution model. In Chapter 4, we discuss the realization of the proposed approach on two different platforms: 1) a Xilinx FPGA platform using FreeRTOS [8]; and 2) a COTS platform using ERIKA RTOS [11]. Finally, in Chapter 5 we extend the proposed scheme to globally scheduled systems.

Part(II) of the dissertation includes Chapters 6, 7, and 8. Chapter 6 addresses the inter-task asynchronous communication model, while Chapter 7 presents our novel bundled scheduling model for real-time parallel tasks. In addition, in this chapter, we show how to integrate bundled scheduling with the proposed multi-phase execution model. Chapter 8 discusses inter-core communication and presents the proposed HopliteRT NoC. We also show how the communication latency of bundled parallel tasks can be bounded based on the proposed NoC design.

Finally, Chapter 9 concludes the work by discussing the limitations of the presented approaches, and providing an overview of possible extensions for this research.

Chapter 2

Background and Related Work

This chapter provides a background overview of important concepts required to better understand the work proposed in the following chapters. Following the same theme of the dissertation, we first overview important concepts and relevant recent works in regards to predictability and timing isolation of real-time tasks, for both cache and scratchpad-based systems. Afterwards, we discuss predictable task communication and related topics, in particular, scheduling of real-time parallel tasks and inter-core communication.

2.1 Predictability and Timing Isolation in Multicore Systems

Modern embedded system platforms often incorporate a multicore processor, which is a chip that has more than one processing core. This is to satisfy the increasing demand for higher computing performance required by modern real-time applications [13]. These chips are also known as Chip Multi-Processors (CMP). The generic architecture of a multicore processor is as shown in Figure 2.1. Throughout this work, we use the terms core, CPU, and processor interchangeably. Note that the private local memory can be realized as cache or scratchpad memory. In this dissertation, we focus on using scratchpad architecture.

In real-time systems, schedulability analysis is based on the assumption of a fixed, known WCET for each task. There are two main approaches to estimate the WCET; static analysis methods and measurement-based methods. A static analysis method does not require the task to be actually executed on the hardware. The analysis instead takes the task's code as input along with annotations from the user, finds the set of all possible

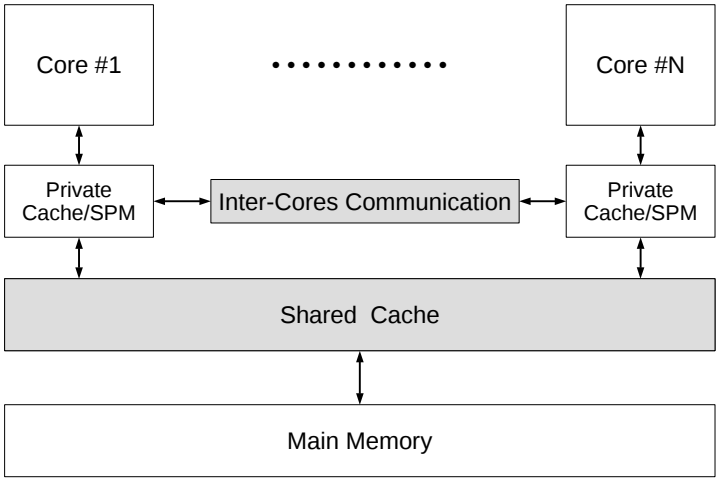


Figure 2.1: Generic multicore architecture, gray blocks might not be available in some systems

control-flow paths through the task, combines control flows with a model of the hardware, and estimates an upper bound to the WCET of the task. However, the complex nature of modern multicore systems makes this type of analysis very difficult or even impossible due to the lack of precise knowledge about the hardware; and thus hardware modeling becomes inaccurate [99, 44]. Usually, the results of static analysis are pessimistic due to the imprecise assumptions they need to consider to model the hardware safely.

On the other hand, measurement-based methods can provide more optimistic (but not safe) WCET estimates without the need for a precise hardware modeling. Generally, the task is executed on the hardware for a set of inputs. After that, the distribution of the observed execution times is determined. However, measurement-based methods are not guaranteed to cover all possible execution paths. Consequently, the estimated WCET can be unsafe, *i.e.*, less than the actual WCET. Modern tools often combine measurement and static analysis methods to improve the accuracy of the estimated WCET and provide safe bounds. For example, RapiTime [9] is a tool that uses a hybrid technique between static analysis and measurement. It basically uses measurement to time individual code blocks, while using analysis to determine the worst program execution path even if it was not executed during a specific run.

In contrast to analysis solutions, researchers also adopted another approach that tries to engineer the hardware and software platforms so that they become more predictable and easier to analyze. In such a case, WCET becomes easier to determine and tight bounds

can be achieved. Our research falls in this type of solutions: we focus on the design aspects of the embedded platforms, rather than WCET analysis. Among the issues that current multicore platforms impose on real-time systems are contention on shared resources such as main memory, system bus, and shared last-level cache. Even in the case of a private cache, tasks' execution predictability can be affected by inter-task and intra-task conflicts. In what follows, we briefly compare the SPM and cache architectures, and review related works on cache and scratchpad systems. Also, we review other platform engineering techniques aimed at achieving predictability by enforcing timing isolation between cores.

2.1.1 Comparing the Architecture of SPM with Cache

Scratchpad memory (SPM) has received considerable attention in the embedded and real-time communities as an alternative to processor caches. Contrary to cache memory, scratchpad memory supports consistent and predictable access times. The SPM is also better than the cache in terms of area and power efficiency [28]. Figures 2.2 and 2.3 compare the SPM and cache architectures. The SPM is a simpler digital circuit, and hence occupies smaller area and consumes less power. The on-chip memory (SPM and cache) is usually built using static RAM circuitry [76]. Static random-access memory (static RAM or SRAM) is a type of semiconductor memory that uses bistable latching circuitry (flip-flop) to store each bit.

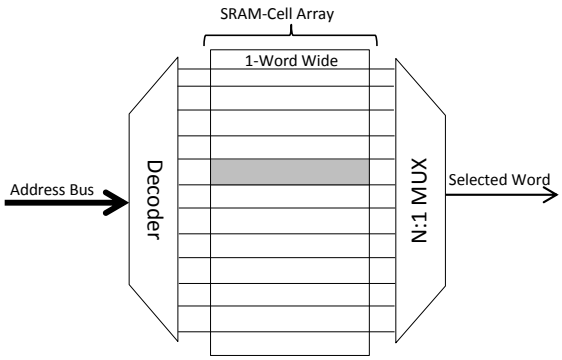


Figure 2.2: SPM Architecture

The SPM memory, as shown in Figure 2.2, is a simple memory circuit. From the architectural prospective, accessing a memory word only requires a decoder and a multiplexer. Accessing memory and selecting a specific word is called indexing. During the access, the

target SRAM cells are selected and then read or written. On the other hand, the cache architecture is more complicated. It consists of multiple data memories, all of which are indexed at the same time [28]. In addition, valid-bit and tag arrays are indexed for each access, which consumes more power. Moreover, the selection of the addressed word is implemented in two levels. First, one word from each data memory is selected the same way used in the SPM (N:1 MUX). This selects the whole cache line. Second, the block offset (part of the address) determines which word is selected from the cache line. This multiple level selection mechanism makes cache memory slower than the scratchpad memory; and the cache uses more memory to store tags and valid bits.

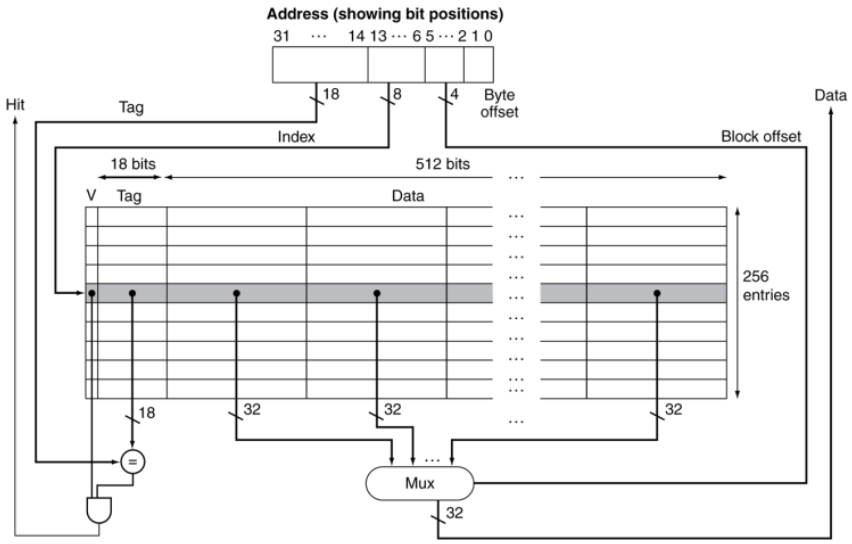


Figure 2.3: Direct-Mapped Cache Architecture (From [1])

In the direct mapped cache, which is the simplest cache architecture, there is one comparator that compares the requested tag and the stored tag, to determine if the access is hit or miss. Associative caches are more complicated, thus slower than direct mapped cache. In fully-associative caches, as shown in Figure 2.4, the entire address is used as a tag. Therefore, one comparator is required for each cache line. The increased number of comparators in the associative cache negativity impacts the maximum clocking frequency of the system. However, associative cache has better overall system performance than the direct mapped cache as it reduces the conflict-miss rate. The set-associative cache is somewhere in between the direct mapped cache and the full associative cache in order to balance the trade-of between the clocking frequency and the conflict-miss rate. A study

by Banakar et el. [28] in the embedded domain compared energy, area and performance of both SPM and cache. The study indicated that SPM achieves better than cache almost on all compared counts.

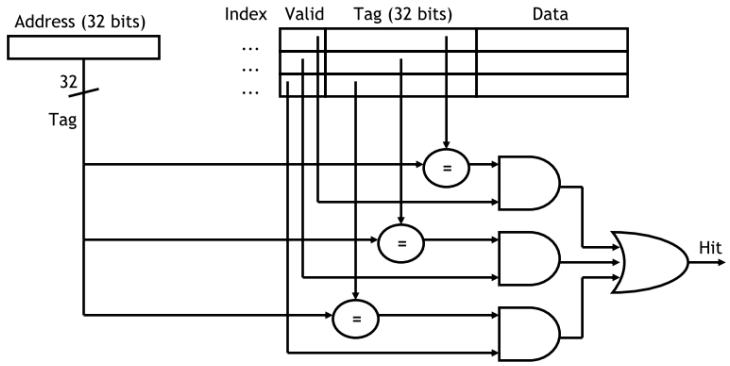


Figure 2.4: Full Associative Cache Architecture (From [1])

Faster access, smaller area, lower power, and predictable access time make scratchpad memories attractive for the embedded and real-time domains. However, SPMs are more difficult to manage. In many cases the application programmers need to do low-level programming in order to utilize the SPM. Thus, the programmers need to have some knowledge about the hardware platform they are working on. In multitasking systems, this becomes very difficult because programmers need to manage the SPM, and synchronize their work accordingly. Caches on the other hand are self managed at the hardware level and do not impact the software model. In other words, caches are transparent to programmers. Consequently, applications can be ported to new platforms more easily.

In the embedded system domain, there have been proposals for specialized memory structures that fall in between a cache and scratchpad. For example TickPad Memory (TPM) [86]. TPM has a dynamic hardware loading mechanism that is statically controlled. TickPAD stands for Tick Precise Allocation Device. Therefore, it is best suited for synchronous programming languages such as Esterel [37] and Pret-C [20] which sample inputs at discrete instants of time and provide outputs at instants known as tick time. Based on the Timed Concurrent Control Flow Graph (TCCFG) created for the synchronous program, a tickPad allocation analysis is performed to obtain a configuration file that determines static allocation decisions for the TPM. At runtime, TPM automatically loads the right context at the right time. TPM generally aims to be more predictable than cache and easier to manage than SPM. However, TPM relies on a specialized synchronous

programming language and is limited to a single application scenario.

2.1.2 Cache Memory

Several well-developed cache analysis techniques have been proposed for single-core processors. These techniques analyze the interference due to intra-task and intra-core cache conflicts. The latter is known as cache related preemption delay (CRPD). The CPRD focuses on cache reload overhead due to preemptions while the intra-task analysis focuses on the cache conflicts within the same task assuming non-preemptive execution.

In existing multicore processors, the last-level cache is typically shared by multiple cores. This design has several merits such as increasing the cache utilization, reducing the complexity of cache coherency and facilitating a fast communication medium between cores. However, it is extremely difficult to accurately determine the cache miss rate because the cache content depends on the size, organization and replacement strategy of the cache in addition to the order of accesses. Shared caches in multicore processors are similar to caches in single core processors in that they all have inter/intra-task interference. In addition, when multiple cores share a cache, they can evict each other's cache lines, resulting in a problem known as inter-core interference.

Unfortunately, single-core cache timing analysis techniques are not applicable to multicore systems with shared caches. Inter-core interference is caused by tasks that can run in parallel and this requires analyzing all systems tasks. The analysis of non-shared caches has been already considered as a complex process and extending it to shared caches is even harder. In fact, the researchers in the community of WCET analysis [155] seem to agree that "it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention between multiple cores in a shared cache".

A timing analysis technique for concurrently running software on multicores with shared caches was proposed by Liang et al. [95], which extend the work in [177]. This analysis targeted the inter-core cache evictions. In this work, the lifetime of all the tasks that concurrently run on multiple cores are determined and then the anticipated conflicts in LLC are computed. The analysis accounts for the cache accesses and use static analysis approaches to estimate tasks WCETs. Another work by Hardy et al. [70] aimed to tighten WCET estimate. This work is based on Hardy's previous work [71]. In this work, the authors proposed a compile-time method to reduce shared cache interference for instructions among cores. This work supports WCET estimates in multiple cache levels in the presence of inter-core interference. In addition, this work statically identifies code blocks

that are used only once during the execution enabling these blocks to bypass the cache. Consequently, a tighter WCET can be computed.

Probabilistic approach to analyze cache also has been introduced in the literature. Quinones et al. [139] explored the effect of using random cache replacement policy on hard real-time systems. They showed that by applying Probabilistic Timing Analysis (PTA), they were able to avoid the risk of the aforementioned cache-based unpredictable access time. Probabilistic Timing Analysis (PTA) has emerged as a solution to reduce the amount of information needed to provide tight WCET estimates. Nevertheless, it imposes new requirements on hardware design. For instance, before [84], only fully-associative random-replacement caches have been proven to fulfill the needs of PTA, but they are expensive in size and energy. As a solution Kosmidis et al. [84] have proposed a hardware design that implements random-replacement cache which allows set-associative and direct-mapped caches to be analyzed with PTA. There however exist other opinions in the real-time community regarding the use of PTA in real-time systems. For example, Reineke [140] showed that the probability of hits that are computed for PTA are not independent. Consequently, the convolution of Execution Time Profiles (ETP) is not possible with randomized caches and hence is not suggested to be used in real-time systems.

In contrast to timing analysis techniques where caches are used without restrictions, the approach of managed caches has the advantage to avoid complex analysis methods for estimating the cache behavior.

Cache Locking and Partitioning

Cache locking and partitioning are techniques to gain predictable access to the shared cache. By locking a cache-line, no replacement/eviction can occur on the content of that line until it is unlocked. Cache locking requires hardware support and it is only supported in some COTS platforms. Different platforms support different styles of locking. For example, lockdown by cacheline or by way. In addition, in multicore systems, there are some platforms that support lockdown by master (core) in which the locked cache-way by some-master cannot be altered by other masters. Among the platforms that support this feature are Nvidia Tegra-2 and Tegra-3[1], Xilinx Zynq-7000[2], and Samsung Exynos 4412[3].

Similarly, cache partitioning provides exclusive access to a portion/partition of the

[1] http://www.nvidia.ca/object/tegra-superchip.html

[2] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[3] https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-4-quad-4412/

cache. Partitioning can be core-based or task-based. Contrarily to locking, cache partitioning can be done in hardware or in software. The most common software-based cache partitioning technique is page coloring [97, 164, 67]. Page coloring explores the virtual to physical page address translations presented in virtual memory systems at OS-level. Cache partitioning can also be done by the compiler [117].

Suhendra et al. [155] explored the effect of locking and partitioning of last-level shared cache on predictability in multicore systems. They used different locking and partitioning schemes. The study combined static/dynamic locking with task-based/core-based partitioning. The results showed clear impacts of different configurations on predictability versus performance. They concluded that there is no one configuration suitable for all types of applications. The best cache configuration for predictability was static locking/task-based partitioning, as each task has its own cache partition that was not affected by preemption. However, a task gets a smaller partition as the number of tasks increases, which impacts the performance.

Shekhar et al. [147] is another work that utilizes cache locking to improve overall system utilization. This work targets many-core architecture where each core has lockable private cache, there is no L2 shared cache in the system. This work proposed a semi-partitioned scheduling where tasks, as many as possible, are statically assigned to cores. Those tasks that are not statically assigned to a core are allowed to migrate from one core to another. Tasks, in each core, are locally scheduled according to Earliest-Deadline-First (EDF). Tasks are allowed to lock some data in the private local cache. For the migrating tasks, locked lines belonging to that task are unlocked, migrated and re-locked on the target core.

A work at the kernel-level toward improving performance with guaranteed predictable timing is done by Mancuso et al. [104]. This work targeted shared last-level cache. In this work, real-time tasks are first profiled in order to depict the most accessed memory locations (hot pages) for each task. The profiling information are then used, at runtime, in a cache coloring and locking mechanism that helped tighten the WCET for real-time tasks. In this work, the cache space is partitioned among all tasks. Another work by Ward et al. [169] proposed the use of page coloring mechanism along with cache scheduling instead of statically partition the cache. However, this work exhibited some overheads due to the dynamic locking approach. None of the studies reviewed above consider parallel tasks and associated need for intra-task communication.

2.1.3 Scratchpad Memory

We split the discussion of SPM in two categories, general purpose embedded systems, and real-time embedded systems.

General-Purpose Oriented Approach

As mentioned before, the most important issues in the domain of general embedded systems, are power consumption and processing performance. Using scratch-pad memories in embedded platforms can result in significant gains to power efficiency and processing performance. Different approaches were taken to employ scratch-pad memories in embedded systems, but the common objective was to make it more usable and more programmer friendly in order to encourage porting software to the new platform. [63][57][127][116] used conventional MMU to manage the SPM at runtime. Managing the SPM at runtime allows different applications to utilize the SPM dynamically, resulting in better resource utilization.

Egger et al. [57] used a compile-time technique to manage the MMU, and loads tasks' code into the SPM dynamically. In this work, the compiled binary has to be processed by the post-optimizer, a tool they developed in order to make the final binary optimized for their memory architecture. Basically, the post-optimizer does a lot of work to disassemble and profile the code statically to determine the basic blocks and functions boundaries. Then the post-optimizer generates a profiling binary image by injecting profiling code. The profiling image in-turn runs in a simulator. By running the profiling image several times a new set of profiling information can be extracted, such as which parts of the code consume more power and which parts are executed more frequently, etc. Finally, the post-optimizer can now generate the SPM-optimized binary image by grouping the code sections into three main regions. The cached and uncached regions are mapped normally, using the MMU, to physical addresses. The pageable region is mapped using a runtime component called the ScratchPad Memory Manager (SPMM). When a process is first loaded, the SPMM disables all the mapping entries in the page table that correspond to the pageable region. Then when the code reaches a point that is not mapped to a physical address, the MMU causes an exception. The runtime in turn forewords the exception to the SPMM, which loads the code into the SPM and adds the mapping entries into the page table. One advantage of this technique is that neither the size of the SPM nor the size of the application's code is needed to be known at compile time as loading is done at the granularity of the page size. In this work, the CPU is used to copy the code from the main memory to the SPM based on interruption from the MMU, which can degrade the performance. This work did

not investigate the case of multitasking system, which is more realistic than a single task system.

Francesco et al. [63] adopted a hardware-software co-design approach to dynamically manage the SPM at runtime with minimum overhead on the CPU. This study proved the power efficiency of the SPM in dynamic applications. The study even showed an improvement in the performance. A DMA component was used to relieve the CPU of copying data to/from the SPM. Loading applications code into the SPM was not considered in this study as it only focused on data. The MMU helps map the addresses at runtime. This work exposes a high-level API to manage and allocate data into the SPM. However, it is still the programmer's responsibility to use the API in order to allocate space in the SPM and move the data back and forth using the DMA.

In [116], a simple technique at the OS level is used to allocate dynamic data (application-heap) to the SPM. A C++ framework enables the programmer to easily annotate the code. The annotation directs the OS to make the decision of where to put the data: on the system heap (main memory) or on the application heap (SPM). The C++ `new` and `delete` operators were overloaded to handle the programmer's preference of allocation. The OS handles the actual allocation depending on the available space; hence no guarantee to allocate data in the desired SPM heap. This work is evaluated on an FPGA platform and showed improvement in both power efficiency and performance. This technique only targets dynamically allocated data. In order to get more significant improvement, it is intended to be coupled with other compile-time techniques to allocate code and static data into the SPM.

In [162] the authors propose a scratchpad memory management technique for preemptive multi-tasking systems where they introduce three methods for SPM partitioning that are: (i) spatial, meaning that each task has its exclusive space in the SPM, (ii) temporal, meaning that a running task can use the entire SPM and the content of SPM is swapped using the native RTOS support and HW module ofthe context switch, and (iii) hybrid approaches where the higher priority task can temporarily use the space of the lower priority task. By employing these three methodologies on a real-time operating system the authors show that they were able to save 73% of energy when compared to the standard approach. The authors also conclude that hybrid approaches outperform the other two approaches. However, this work has not been applied to multi-core processors. Moreover, the focus of [162] is not predictability but energy efficiency.

Work presented in [127] is also an interesting effort that automatically loads the system stack into the SPM at runtime to achieve better system performance and power consumption. This work exploits the locality characteristics generally observed in the stack memory

access to achieve performance improvement. The basic idea is to continuously map the stack pointer into the SPM, using the MMU. Since the active part of the stack is always near to the stack pointer, mapping the stack pointer into the the SPM will improve the performance of the active part of the stack, which contains the local variables of the current function. If stack outgrows the SPM or if the stack grows out of the SPM, the MMU causes an exception. The exception handler will do a replacement for a segment of the SPM by saving that segment into the main memory and loading the new addresses into the SPM. The page table entries are modified accordingly. This mechanism is somewhat similar to the one used in [57], but this work is still interesting as it does not need post-compile processing or specialized hardware in addition to the MMU. It is also transparent to the application programmers and there is no API needed. Like in [57], there is no DMA component to load/unload the SPM, which results in reduced performance.

Real-time Oriented Approach

In the domain of real-time, SPM has first been utilized by statically allocating and partitioning the SPM to reduce the WCET. Suhendra et al. [156] constructed an Integer Linear Programming (ILP) to optimally allocate feasible paths into the SPM. Another study [163] also used ILP to statically partition and allocate tasks on the SPM with the focus on energy efficiency. The allocation algorithm used heuristics to avoid loading data from unfeasible paths and thereby reduces the WCET.

Other works such as [136] investigated the use of the SPM in embedded systems. [136] quantitatively compared memory-mapped SPM to locked cache in terms of WCET. They used a compile-time algorithm to dynamically load tasks into on-chip memory as needed. Similar to the work by Pellizzoni et al. [128], they injected code at the boundary of functions and basic blocks. Regardless of the inability, in some cases, of locking two basic blocks simultaneously in the cache due to their conflicting addresses, the results of testing benchmarks' WCET were closed to each other.

Other approaches attempted to manage SPM dynamically at runtime to achieve predictable execution time. Kim et al. [83] is a work that targeted Software Managed Multicore (SMM) architectures, such as IBM cell multi-processor [103]. SMM is particularly promising for real-time systems as it has advantageous characteristics such as scalability, power efficiency, and predictability. In SMM architectures, each core can only access its scratch-pad memory (SPM); any access to main memory is done explicitly by DMA. Consequently, dynamic management of the local SPM for both code and data has to be done by software. [83] introduced two WCET-oriented dynamic SPM code management techniques for SMM architectures. One is optimal and is based on ILP. The other is faster heuristic based

algorithm. This work focused on determining the optimal function to region mapping for SPM. This was done by constructing a control flow graph of the task and then performing an interference analysis of every function whenever the DMA is needed to load new code into the SPM. In case of mapping conflict, the needed functions need to be reloaded by DMA.

Whitham et al. [174] proposed simplified runtime load/store operations using custom instructions coupled with custom hardware. The Scratchpad Memory Management Unit (SMMU) is a custom hardware component that maps/unmaps data objects and variables to the SPM. In addition the SMMU performs DMA functionality by moving data from main memory to the SPM and visa-versa. The SMMU makes memory accesses transparent by allowing address translation from the CPU virtual address to the physical address in the main memory or in the SPM. The hardware structure of the SMMU is object-based and allows only up to N data objects to be mapped at the same time. As a result, the comparator array circuit of the SMMU can be a performance bottleneck for the system operating frequency, resulting in a scalability problem.

Whitham et al. [176, 175] is the first work to consider managing on-chip scratchpad dynamically in a multitasking system. It introduced a new memory model called Carousel. Carousel acts as a stack of fixed-size blocks. The top few (n) blocks are stored in the SPM, and the remaining blocks are stored in the main memory. A task is divided into several Carousel blocks depending on its size. Starting a task requires pushing all its blocks into the top of the Carousel stack, meaning that the task will be moved, using DMA, from the main memory to the SPM. In order to free space in the SPM, Carousel also swaps out a similar number of blocks to the main memory when they are not among the top n blocks of the stack. The process is reversed when ending a task. All task blocks are popped from the top of the stack, moving the task to the main memory. Carousel also brings the previously swapped-out blocks back to the SPM in a stack-wise fashion. The downside of this work is that the CPU remains idle while the DMA is doing the transfer.

In addition, there has been a significant amount of work in literature that proposes solutions to dynamically manage the SPM for one task [54, 101, 25, 157, 58] by reusing the available space over tasks execution. These works propose solutions for managing tasks code and data including stack and heap. Since SPM is not transparent with respect to address translation, management schemes have to impose constraints on analyzable code; in particular, memory aliases must be statically resolved, since otherwise the management scheme risks loading the same data into two different positions in the SPM.

[54] proposed a complier-level WCET-directed algorithm to dynamically allocate static data and stack data of a program to scratchpad memory. The granularity of placement of

memory transfers are at the function level, basic block boundaries.

2.1.4 Other Task Isolation Techniques

In this section we discuss works that are not strictly categorized as cache- or scratchpad-oriented solutions. However, they propose engineered hardware or software solutions that provide better task isolation thus improving predictability.

Hardware Solutions

Huangfu et al. [75] introduced Performance Enhancement Guaranteed Cache (PEG-C). It is a hardware addition to regular I-cache in the form of a benefit counter for the hit and miss rates. This hardware design addresses the unpredictability in the access to caches, and also enhances the average performance comparable to a regular cache. The benefit counter keeps track of the number of hits and misses at runtime and provides access to the cache only when the value of the benefit counter is positive; otherwise, the access is served from memory.

Allard et al. [19] have proposed a hardware component named hardware context switch (HwCS). HwCS replaces the standard Ll cache controller of a processor. It divides the cache into two interchangeable layers, similar to our approach. The CPU can execute from one layer that acts as a regular cache, while enabling to save or load the content of the other layer simultaneously. HwCS makes the preemption overheads smaller compared to the task WCET as the cache content is saved after preempting the task and restored before resuming the task. Since both layers can access main memory at the same time, memory bandwidth is divided between the tow layers in the worst case.

PRET machine [56, 96, 61, 135] is another line of solutions that target real-time systems from another angle. PRET stands for PREcision Timed machine. The philosophy of PRET is to provide a computing environment that is as timely and precise as the underlying synchronous digital system that implements the computer hardware. PRET demands big changes in processor design, memory subsystem, Instruction Set Architecture (ISA), programming language, and RTOS. The main reason for these changes is to deliver or propagate the notion of timing from the circuit-level to the software application level. Only then can the programmer express the temporal behavior as easy as the functional behavior of the application. As a first step, [96] proposed a multi-threaded single-core processor with extended version of SPARC ISA that delivers predictable timing. The processor pipeline is interleaved between six hardware threads. Each thread has

private instructions and data scratchpad memories. Access to the shared main memory is arbitrated by a mechanism called memory wheel that uses round-robin scheduling policy to guarantee an exclusive access of threads to main memory in their time window. In addition, a deadline instruction has been added to the ISA to help synchronizing threads precisely. [61] presented a plug-in for LabVIEW Embedded that maps the LabVIEW G graphical programming language and its timing specifications to PRET. [135] proposed a tool that statically allocates instructions from multiple threads to a shared SPM for the PRET architecture. Similarly, Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (MERASA) is a project that develops hardware specifically for Hard-real-time system [2, 4, 125]. Similar to PRET, MERASA [167] involves big modifications in processor design, cache memory, and bus and other interconnects for single- and multi- core systems. P-SOCRATES project [133] focuses on designing a predictable many-core systems. Specifically, the purpose of P-SOCRATES is to develop an entirely new design framework, from the conceptual design of the system functionality to its physical implementation, to facilitate the deployment of standardized parallel architectures in all kinds of systems.

Software Solutions

Pellizzoni et al. [131] highlighted the impact of the multicore architecture on WCET based on contention for access to main memory. This work shows a linear increase in the WCET with the number of cores. Pellizzoni et al. [128] also introduced a Predictable Execution Model (PREM). PREM provides isolation in a multi-tasking system by scheduling access to main memory. PREM is based on software only and does not require hardware arbiters. It is based on compiler and OS techniques to divide a task into a memory phase and an execution phase; a task can run predictably from cache with no access to main memory while it is in the execution phase. This allows other masters in the system, such as IO, to be scheduled to access main memory while a task is in the execution phase. The work has been extended to multicores in [178] adopting a TDMA arbitration scheme with partitioned system. [26] studied different scheduling policies for PREM in multicore systems and compared it with the previous TDMA approach, EDF, and contention-based. The best schedule was based on least-laxity-first policy. After that, a global schedulability study has been introduced in [15]. A parallel task model for PREM has also been introduced in [16]. [178] and its derivatives partition the cache space among all tasks.

Note that, our proposed solution differs from PREM in two ways; 1) while PREM uses the CPU to prefetch the task into the local cache affectively wasting the CPU time, in ours solution, we pipeline the CPU execution and the DMA transfer. 2) PREM partitions the

cache space among all tasks, while in our solution, we limit the local SPM to two partitions only.

MemGuard [180] is another work which provides memory performance isolation while still maximizing memory bandwidth utilization, based on resource reservation or reclaiming techniques. MemGuard dynamically reserves and regulates per-core memory bandwidth or accesses based on hardware performance counters. If a core exceeds the predefined maximum access usage, an interrupt will cause the core to jump back to the OS-level bandwidth regulator. As a result, the DRAM bandwidth is partitioned among cores guaranteeing a minimum bandwidth for each core. MemGuard dynamically adjusts the resource provision based on its actual usage. For example, when the task is highly demanding on the resource, it can try to reclaim some possible spare resources from other tasks; on the other hand, when it consumes less than the reserved amount, it can share the extra resource with others. Another way to achieve predictability can be DRAM bank-aware allocation (PALLOC) proposed by H. Yun [179]. However, on some platforms (such as NVIDIA TX1), controlling DRAM bank allocation is problematic due to address randomization aimed at improving average performance.

From higher level, Mancuso et al. [106] proposed OS-level techniques for COTS multicore architectures that partition the system into isolated single-core virtual machines. Usually, modular per-core certification cannot be performed in COTS multicore systems due to shared resource interference. However, this work allows per-core schedulability results to be calculated in isolation and to hold when multiple cores run in parallel. Thus, existing software and schedulability analysis developed for single-core can be used as is in a multicore environment by utilizing the proposed single-core-equivalent virtual machines.

2.2 Real-time Tasks Communication

As mentioned in Chapter 1 and detailed in Chapter 6, we consider two types of tasks communication, inter-task and, intra-task communication. For the inter-task communication we consider asynchronous communication model between different tasks. In this model, the communication is simplified as there is no precedence constraint between tasks as in synchronous communication. Tasks share data via main memory and a task reads the last updated data from the senders. Assuming our proposed SPM-centric system architecture, this type of communication does not not involve inter-core communication as communicating tasks do not exchange data while running in parallel. Therefore, most of synchronous real-time communication works is not directly related to this thesis. Note that, in cache-

based systems, this model might invoke inter-core coherence activity, *e.g.*, when tasks run on different cores, there might be invalidation/copy of data modified by the sender.

On the other hand, intra-task communication targets parallel tasks. This involves inter-core communication, as parallel threads need to exchange data while running in parallel. In the following subsections we overview related topics. First, in Section 2.2.1, we discuss parallel tasks in real-time domain and identify the shortcomings in modeling communicating threads. As discussed earlier in Section 2.1, we assume private SPM for each core. Accessing shared address in private SPMs breaks memory consistency and requires inter-SPM coherence. In section 2.2.2, we review memory consistency models and identify predictable real-time inter-core coherence protocols. Lastly, Section 2.2.3 reviews real-time NoCs.

2.2.1 Real-Time Scheduling of Parallel Tasks

Real-time scheduling for multicores usually has the assumption of a set of independent tasks. Lately, there has been increasing trend to schedule multi-threaded applications (parallel tasks). This growing interest is motivated by modern real-time applications that require more performance that can not be easily achieved without multi-threaded applications in which the threads can run in parallel. Some examples include unmanned aerial vehicles and self-driving cars that require processing data from different sensors including video cameras. Today, even many hobby toys and models, such as quad-copters, needs sophisticated real-time computation.

Researchers have devised schedulability analyses for a variety of different system models. To ease classification, we distinguish between two orthogonal concerns: the scheduling model and the task model. In the thread scheduling model, parallel threads of the same task are scheduled independently, whereas in gang scheduling, the threads must execute concurrently.

Parallel tasks can have multiple threads that can exceed the number of processors available in the system. It is common that parallel tasks are modeled using fork-join structure or a directed acyclic graph (DAG). In the fork-join structure, an application starts with a single thread and then forks multiple threads. After that, the task can keep alternating between a single sequential thread and multiple parallel threads. The application also ends, usually, with a single thread. In the case of a DAG, threads can be modeled as nodes in the graph, and the dependencies are modeled as directed edges. Edges represent precedence constraints: a predecessor node must be executed before a successor node.

Most related work considers thread scheduling. Work considering the fork-join task model include [88, 92, 141, 48, 120]. The DAG task model is analyzed in [32, 92, 93]. [109, 31] extended the work on classical DAG model by introducing conditional DAG model, where subtask execution depends on the conditional path of execution through the program.

As a first work in parallel real-time tasks scheduling, Lakshmanan et al. [88] proposed a scheduling algorithm for OpenMP fork-join structure. Nonetheless, the work in [88] restrictive as task has to fork to the same number of threads each time. Consequently, Saifullah et al. [141] relaxed the previous model in [88].

Some works have proposed scheduling schemes based on assigning intermediate deadlines to individual threads of the same parallel task. Nelissen et al. [120] have proposed techniques to determine the intermediate artificial deadlines while minimizing the number of processors needed to schedule the whole task set. In addition, threads are treated as if they were independent sequential sporadic tasks. Unlike the previous reviewed works, Baruah et al. [32] proposed a scheduling analysis for single parallel task expressed as a DAG. Whereas, Li et al. [92] considered the same model but for multiple tasks. Nonetheless, all reviewed works are only concerned with pure CPU scheduling, and none of them considered contention on shared resources, such as shared cache and main memory.

[93] introduced federated scheduling of classical DAG parallel-task model. It is a generalization of partitioned sequential tasks model by allocating dedicated set of cores for tasks with utilization higher than one. A fundamental assumption is that under these models, parallel threads of a task can be scheduled independently with no synchronization penalty.

On the other hand, Alhammad et al. [16] is the first work on real-time scheduling for parallel tasks that considered the interference from other threads caused by contended accesses to main memory. The technique used in this work, to provide thread isolation, is an improvement over [15] that avoids contention without the need to hardware arbitration. This work differs from [15] as it considered multi-threaded application instead of independent tasks. In addition, this work used profiling scheme similar to [104] in order to divide the application into segments. Each thread is segmented into three consecutive phases (prefetch, execution, write-back) in which synchronization happens at the boundaries of the memory phases (prefetch and write-back). Threads share data through main memory as the write-back phase flushes the cache content to main memory before the next thread starts the prefetch phase. It is also observed that the used technique is more scalable with the number of cores unlike the contention scheme.

In addition, Alhammad et al. [17], extended the work on federated scheduling in [93] by considering the cost of accessing memory. Basically, the authors introduce optimizing

algorithms to assign computation and memory budgets for each parallel task in the system to improve overall system schedulability.

Tessler et al. [165], introduced an application-level scheduling strategy to take advantage of cache memory to tighten the WCET in parallel task. In particular, the proposed method permits threads to execute across conflict free regions, and blocks those threads that would create an unnecessary cache conflict. The WCET bound is determined for the entire set of m threads, rather than treating each thread as a distinct task. The proposed method relies on the calculation of conflict free regions which are found by a static analysis of the task object.

The need to concurrently gang schedule related threads of the same parallel application has been first discussed in [123]. The performance benefits of gang scheduling has been studied in [60, 77, 150] and many others that looked into co-scheduling in general-purpose computing. In the real-time domain, the existing literature that consider gang scheduling distinguish among three task models. In the rigid model [81, 55, 64], the (constant) number of threads required by a task is fixed off-line. In the moldable [38] case, the number of threads assigned to each job is decided at run-time by the scheduler, but kept constant during the execution of the job. In the malleable [47] case, the scheduler can change the number of assigned threads during the job's execution.

Our proposed bundled scheduling for parallel tasks, as discussed in Chapter 7, considers systems where the number of parallel threads is dictated by the application's execution, so that global scheduling decisions do not influence the way the application is executed. Therefore, we only compare our approach against the rigid model. [81] introduced a schedulability analysis for rigid tasks with EDF scheduling policy; whereas, [55] provided a utilization-based schedulability test for EDF. In contrast, this paper targets fixed priorities. Furthermore, as noted in [55], [81] contains a mistake in the way carry-in interference is bounded, while [55] itself is limited to implicit deadlines, rather that constrained-deadlines as considered in our approach. Finally, [64] proposed an optimal, off-line slot-based scheduling algorithm for strictly periodic rigid tasks, but the framework does not naturally extend to sporadic tasks.

2.2.2 Memory Model

Parallel programming in multicore systems is challenging. Even in a shared-memory system, programmers can still get unexpected results due to misunderstanding of the underlying memory consistency model adopted by the platform. A memory model is the set of

specifications that describe how the different components of a system should behave in regards to memory operations. The choice of memory model is affected by many components in the system, such as CPUs, caches, buses or interconnects, memory controllers, and compilers. For example, whether the CPU issues memory operations in-order or out-of-order, the CPU has write buffer or not, the cache is write-though or write-back, and the compiler optimizes and reorders memory operations or not. By understanding the adopted memory model, programmers can code their programs accordingly to produce functionally correct applications. The memory model acts as an agreement between the computer platform and programs or programmers, in which both hardware and software have to adhere to the memory model; thus providing correct functionality. Memory model is very important and must be considered when porting applications from one platform to another.

Coming from a uniprocessor system in which all threads or tasks have consistent memory view, Sequential Consistency (SC) is a natural extension of the uniprocessor notion of correctness and the most commonly assumed notion of correctness for multiprocessors [14, 153]. In the SC all memory operations are serialized and viewed by the memory system in the original program order. When local caches are present, SC requires that write operations to the same memory address must be atomic, e.g, no other write to the same address is permitted until the current write is completed and propagated to all nodes. This model is considered as the natural successor to the uniprocessor memory model as programmers can confidently assume that any read will always return the most up-to-date value. The SC model makes migration of applications that share data from a uniprocessor system to a multicore system easy [14].

Although SC is good from a programming point of view, SC imposes some performance challenges such as preventing out-of-order memory operations and enforcing to wait for a previous write operation to complete [14]. Furthermore, some performance optimizations that were valid in a uniprocessor system, such as utilizing processor's write buffer, now can lead to problems. These restrictions are important because compilers and out-of-order CPUs generally guarantee the order of the dependent memory operations only, assuming that independent memory operations will not affect the same thread or task. In multicores however, this might affect threads or tasks running simultaneously on other cores. In addition, IO devices can be affected by the order of independent memory operations because the order of setting some devices' registers can be important.

Other less strict or more relaxed models have been proposed to allow for performance optimizations, such as Processor Consistency model (PC), and Total-Store-Order model (TSO) [153]. For example, TSO relaxed SC by allowing the use of the write buffer, hence allowing read after write re-ordering. Generally, different relaxed models allow different optimizations. To enforce program order, relaxed-modeled CPUs implement specific instruc-

tions to enforce order between memory operations, such as memory fences. Programmers and compilers can utilize these instructions to achieve the desired functionality. Although relaxed models offer better performance, they introduce more complex programming model as programs need to use low-level instructions for synchronization.

Another set of models are the Weak Ordering models (WO) [153]. In a weak ordering model, memory operations are classified as synchronization operations and data operations. The WO models are based on the intuition that reordering memory operations to data regions between synchronization operations does not typically affect the correctness of a program. Therefore, it offers more space for performance optimizations with less complex programming model than the relaxed memory models. To enforce program order between two operations, the programmer is required to identify at least one of the operations as a synchronization operation. In a WO model, all types of re-ordering are allowed between data memory operations. However, all memory operations issued before the synchronization memory operation have to complete before issuing any following memory operation.

Release Consistency model (RC) follows the philosophy of the WO. However, RC provides a clear and higher-level programming model. In RC synchronization operations are divided into acquire and release operations; synchronized memory operations are enveloped between acquire and release. Acquire and release are analogues to lock and unlock respectively. Acquire operations tries to lock (has exclusive access) to a flag that is used to sy nchronize accesses between parallel threads to what is called a Critical Section (CS). A CS is a piece of code that accesses memory addresses subjected to synchronization. On the other hand, release operation unlocks (allow access) the locked flag to allow other threads to lock it again and gain access to the critical section. All memory operations between acquire and release are not restricted to be in the program order. However, RC provides correctness similar to SC as it guarantees atomicity and sequential ordering between critical sections. For example, depending on the implementation, a release operation might be delayed until all memory operations enveloped between the acquire and release are completed and delivered to all relevant nodes. The implementation of this model can be done in different ways. For example, compilers and programming languages can implement the semantics of the model utilizing the low-level memory fences, and read-and-modify instructions. In addition, operating systems can provide a synchronization library that implement acquire and release semantics.

In this work, RC is chosen to allow more room for performance optimization while keeping the programming complexity minimal. This become relevant when we discuss the intra-task communication of parallel tasks in Chapter 8. In particular, our methodology is to keep the hardware design simple, which what the RC model allows.

Cache Coherency

As mentioned earlier, different components in the system, such as cache controller and interconnect, affect the consistency model. When private caches are not coherent, violation to the memory consistency can happen. In a shared memory system, a coherence problem arises if multiple cores have access to multiple copies of the same data, such as in private caches. Therefore, memory coherence is required in shared memory systems to keep only one version of data that is equally seen by all cores [115, 148].

A hardware platform implements a coherence protocol to adhere to a specific memory model. Based on the number of shared or distributed memories in the system, such as private/shared caches and main memory, and how they are connected, the implementation complexity of the protocol can vary. There are many cache coherence protocols that have been introduced in the literature. Examples are MSI, MESI, MOSI, MOESI, MERSI, MESIF, write-once, Synapse, Berkeley, Firefly, Dragon, and ARM AMBA 4 ACE [21, 85]

Generally, there are two broad classes of coherence protocols, snooping and directory-based. Snooping protocols tend to be faster, if a common bus with enough bandwidth is available, since every request is broadcasted to all nodes in the system. However, the snooping protocol is not scalable as the bus will not be able to provide enough bandwidth as the number of cores increases. Directory-based, on the other hand, is scalable and can be distributed but is however slower than snooping, as the directory transactions have to traverse through the interconnect. In addition, directory-based protocol requires dedicated memory to store the directory.

Independently of snooping or directory-based protocols, the other major design decision in a coherence protocol is to decide what to do when a core writes to a block. There are two options, invalidate and update. Invalidate protocol is when a core invalidates the copies in all other caches before writing to the block. If another core wishes to read the block after its copy has been invalidated, it has to initiate a new coherence transaction to obtain the block, and it will obtain a copy from the core that wrote it, thus preserving coherence. On the other hand, in update protocol, when a core wishes to write a block, it updates (pushes) the copies in all other caches to reflect the new value it wrote to the block. Update protocols reduce the latency for a core to read a newly written block. However, update protocols typically consume substantially more bandwidth than invalidate protocols because update messages are larger than invalidate messages. Most of currently implemented protocols are based on invalidation protocols.

Although there has been a lot of work in improving cache coherency in general-purpose architecture, real-time systems are still behind in analyzing and introducing predictable cache coherency protocols.

In the real-time community, Sarkar et al. [143] introduced a push mechanism that pushes some or the whole content of a cache to another cache. This work is presented in the context of tasks migration, in which a migrated task does not suffer cache warm-up delay. Similarly, Pyka et al. [138, 137] introduced the on-demand coherent cache (ODC^2). Unlike [143], this work aims specifically for a predictable coherence mechanism for real-time systems. The ODC^2 has two modes, share mode and private mode. In the private mode it works as a regular non-coherent cache. Whereas, in the share mode it provides coherency. The share mode is only activated in critical sections. When exiting a critical section, all the cachelines accessed during the critical section are invalidated and flushed-back to main memory. Although the (ODC^2) provides predictable coherency mechanism, it still uses main memory for data sharing, which can reduce the performance.

Nowotsch et al. [121] indicates that in COTS systems performance can be degraded by just enabling cache coherence even with no data sharing. In addition, [166] studied MESI-based coherence protocols for real-time suitability. The study concluded that to provide a statically predictable timing behavior of the MESI-based cache coherence protocol, few conditions have to be met. It requires a time predictable TDMA bus interconnect with an analyzable memory controller and an update-based dual-ported direct-mapped cache using bus-snarfing.

Haasn et al. [72] recently proposed (PMSI): a predictable cache coherence for multi-core systems. The authors extends the classic MSI protocol and enhanced it with transient coherence states to bound the worst-case access latency. This work also reports possible sources of unpredictable behavior on conventional coherence protocols. The results show that, in the worst-case, the arbitration latency scales linearly, whereas the coherence latency scales quadratically with the number of cores. Although, quadratic coherence latency is observed for wost-case scenarios, for general-purpose computing the cost is more reasonable for the average-case scenarios [107]

One-Way Shared Memory [144], proposed a direct communication between cores' local memories via a time predictable NoC without relying on main memory. This is to avoid the unpredictable access time to local memory and the contention on main memory which is a communication bottleneck in the cache-based system. In [144], the distributed local memories are connected in a predictable and coherent way so they appear as one shared memory. Specifically, each local memory is split into TX and RX communication channels such that each core has (m-1) TX and (m-1) RX channels. The Network Interface (NI) continuously synchronizes the local memory with the rest of the network. This NoC adopts a static TDM schedule to arbitrate between injectors and guarantees conflict free communication. The all-to-all communication mechanism implemented in the NoC guaranteed that all communications data are fully propagated to all other destinations within bounded

time (hyper period).

2.2.3 Predictable Network On-Chip Architectures

With the increase in the number of cores in a single chip, the need for a scalable on-chip interconnect has increased as well. Network on-Chips (NoCs) form the communication subsystem for the future chips. It is different from a system bus due to the fact that it applies networking principles and improves scalability as well as power efficiency. As mentioned earlier, interconnect can affect or violate the memory consistency model. Therefore, when designing an interconnect, the adopted memory model has to considered. In particular, the order and the atomicity of operations have to adhere to the specifications of the adopted memory model.

As discussed in Chapter 1, we propose a lean NoC for inter-core communication (HopliteRT), which targets FPGA platforms. HopliteRT routes single-flit packets over a switched communication network using Dimension Ordered Routing (DOR). DOR policy makes packets traverse in the X-ring (horizontal) first followed by the Y-ring (vertical). Hoplite uses bufferless deflection routing and a unidirectional torus topology to save on hardware implementation cost. We use traffic regulators at the network interfaces to control injection rates when the flows are not feasible. In relation to this work, we overview the real-time NoCs in the literature.

Existing literature on real-time NoCs can be summarized in the following broad directions. Some research has built static routing tables for time-division NoCs such as those proposed in [65, 78]. However, this approach requires full knowledge of all communication flows, and is unsuitable for NoCs that needs to support both real-time flows requiring worst case guarantees and best effort flows where average case delay is important.

Specifically, Goossens et al. [65] introduced Aethereal NoC. Aethereal is without contention by design. The basic idea is that each switch or router has a static table called slot table, which maps each input to an output in a specific time slot. This synchronous behavior makes the network easy to implement, as switches or routers do not have buffers and there is no need for complex dynamic routing. Nonetheless, computing the static schedule might take some time. Therefore, re-configuring the network at runtime might be an issue. Generally, NoCs have a constant performance to cost ratio, i.e. the costs linearly depends on bandwidth and latency.

[112] has different assumptions about the communication pattern, ranging from one-to-one restriction to none. It uses Time-Division Multiplexed (TDM) schedule to derive worst-case latency bound and splits traffic into multiple non-interfering flows. However, the

bound limits the maximum rates of the flows depending on the number of non-interfering sets. The TDM schedule may exaggerate the required latency depending on the extent of interference. For HopliteRT, we only need to know the communication pattern and we only limit injection rates when flows are not feasible. In addition, TDMA schedule is more restrictive than HopliteRT as it forces injections at specific time slots.

Other work focuses on wormhole NoCs with virtual channels. The seminal work in [149] proposes priority-based networks, where each virtual channel corresponds to a different real-time priority, thus providing reduced latency for high-priority flows at the cost of low-priority ones. Recent work has extended the analysis to NoCs using credit-based flow control [80], as well as to round-robin, rather than priority-based arbitration [124]. However, these designs are expensive on FPGA, and require full knowledge of communication flows to derive tight latency bounds. In contrast, our approach rely on static modification of routing function as well as client regulators to bound latencies.

Approaches such as Oldest-First [114] and Golden Flit [59] provide livelock freedom on deflection-routed networks similar to Hoplite, but are optimized for ASICs and use a richer mesh topology. On the other hand, minimally buffered deflection NoC [111] is suitable for FPGA and provides in-order delivery of flits eliminating the need for reassembly buffers. However, these reviewed approaches do not provide exact bounds on worst-case times.

The use of regulators to bound the maximum network latency is well-known in the context of network calculus [91]. In [35], the authors show how to use Token Bucket regulators, to control traffic injection on the Kalray MPPA. Unlike HopliteRT, the Kalray NoC is source routed (client computes the complete path taken by the packet), and requires queuing at the client interface and within the NoC.

Part I

Efficient Task Isolation For Real-time Applications

Chapter 3

Partitioned Scratchpad-Centric Scheduling of 3-Phase Real-time Tasks

In this chapter, we present our first main contribution: a set of scheduling techniques, and related schedulability analyses, for real-time tasks executed according to the 3-phase model. In particular, in this chapter we discuss the case of partitioned scheduling, where a set of tasks is statically allocated to each core off-line; Chapter 5 will later cover global scheduling of 3-phase tasks.

We begin by discussing the system model in Section 3.1. While we use a consistent set of rules for scheduling tasks on each core, we consider two different ways in which DMA accesses to main memory can be scheduled among contending cores: at either the hardware or software level. For the former case, each DMA transfer can have a different duration, while in the latter, we are considering a TDMA arbitration based on slots of equal duration. Hence, we first detail the schedule rules and associated schedulability analysis for the case of variable-size DMA operations in Section 3.2, and we then extend the analysis to the case of fixed-size DMA operations in Section 3.3. Lastly, Section 3.4 extends the discussion to fault-tolerant scheduling to recover from detectable memory soft errors that cannot be corrected automatically in hardware by the Error-Correcting Code (ECC) unit.

As mentioned in Section 1.3, our proposed 3-phase scheduling scheme has been implemented on two different embedded platforms, which we detail in Chapter 4. Since the evaluation of the proposed schedulability analyses is necessarily dependant on tasks'

parameters, which are affected by the execution platform, we defer it to Chapter 4.

3.1 System Model

Our main objective is to schedule the shared resources (CPU time, DMA time and local memory space) between different tasks in a predictable manner while hiding latency caused by accessing the main memory. We developed novel scheduling algorithms, based on fixed-priority scheduling, that are able to dynamically manage the system resources and schedule the tasks to run predictably and achieve better system schedulability. The proposed scheduling approach is based on the following assumptions.

Hardware Assumptions

We consider an identical multi-processor system composed of m cores. As shown in Figure 1.2-A, each core has a private scratchpad memory (SPM), and has access to a DMA engine. We further assume that the SPM is dual-ported, so that accesses to different memory addresses can proceed concurrently. Consequently, we assume that the DMA and the processor core can access different portions of the local memory concurrently without causing mutual interference.

Execution Model

To achieve strong timing isolation between running tasks on different cores, tasks are only executed from the local memories. We consider a 3-phase task model where all code and data needed by a task are first loaded into the local memory. After that, the task executes from the local memory without accessing main memory, thus avoiding any contention. Lastly, after the task finishes, all modified data are written back to main memory. The local memory is further partitioned into two partitions. Unlike other schemes [104, 178], our partitioning approach is independent of the number of tasks; thus it is more realistic. Since the two partitions can be accessed in parallel, the CPU can execute one task out of one partition and the DMA can unload and load the other partition at the same time. This technique allows us to hide the latency of accessing main memory by overlapping the execution of one task from one partition of the local memory while fetching the next task from main memory to the other partition of the local memory.

Figure 3.1 depicts a working example of how the shared resources are scheduled among three jobs of three different tasks. This example consists of three sporadic tasks that are all

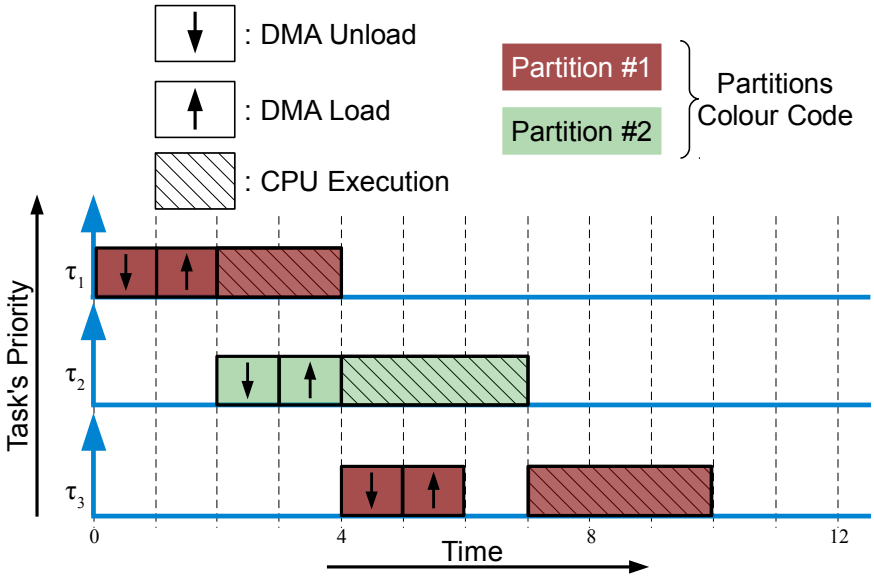


Figure 3.1: Example schedule showing how tasks are executed to hide memory access latency.

released at the same time. Since it is a static fixed-priority schedule, the highest priority task, τ_1, get scheduled first. At the beginning, the scheduler chooses which local memory partition to load τ_1 into, it is the first partition in this case. After that, the DMA is instructed to offload the data section[1] of any previously loaded task from this partition back to the main memory. DMA is then instructed to load code and data of τ_1 into this partition. After that, τ_1 is able to run. While the CPU is executing τ_1 out of partition P1, the DMA operations are scheduled in parallel, and offload partition P2 and reload it with τ_2. At the same manner, loading τ_3 is overlapped with the execution of τ_2. τ_3 has to run out of partition P1. Therefore, τ_1 is evicted first, by writing its modified data back to main memory; then τ_3 is loaded into this partition. The next task that comes after τ_3 is scheduled to run out of partition P2 and so on. Note that it is possible for a DMA operation to take longer than a task execution, not shown in this example. For instance, τ_2 can not start executing until both τ_1 execution and DMA operations on partition P2 are done, in this case DMA operations and the CPU execution take the same time. In a fairly loaded system, this approach can completely hide the memory latency as long as the memory subsystem is fast enough to keep DMA operations shorter than tasks executions.

[1]We do not need to offload the code of the task since it cannot be modified.

In the worst case, a portion of this latency is hidden by the overlapping mechanism, which leads to a better schedulability.

We consider two scheduling schemes for DMA operations. In the first scheme, we assume the presence of an hardware arbiter, that can arbitrate fairly among DMA operations of different cores (for example, using the memory controller in [96]), by ensuring that each core is guaranteed to receive $1/m$ of the memory bandwidth. In this case, we consider DMA operations of variable lengths (time), where the amount of time required to load and unload a task is based on its size, after considering that the DMA engine receives $1/m$ of the memory bandwidth. In the second scheme, we do not make any assumption on the hardware arbiter; hence, to ensure a predictable behavior, we instead schedule the DMA engine in software. In this case, mainly for simplicity of implementation, we consider a Time Division Multiple Access (TDMA) scheme where each core receives one time slot of fixed size σ. We further assume that σ is enough time to load or unload any task in the system, in this case assuming that the DMA engine receives the full memory bandwidth. We discuss the first scheme in Section 3.2, while Sections 3.3 and 3.4 are based on the second scheme. In either case, the memory activity of each core becomes isolated from the activity of each other core, so that our schedulability analysis can consider each core in isolation.

Note that since the CPU must execute without accessing main memory, our model implies that the code and data of each task must fit entirely in local memory partition. Although this assumption may appear restrictive, we make the following considerations. First, as will be shown in the evaluation in Chapter 4, some modern COTS scratchpad-based micro-controllers provide scratchpad memories that have a size in the same order of magnitude as the main memory. Second, hard real-time control tasks are typically compact in terms of memory size. Third, if a task violates this size constraint, known methodologies exist [128, 145, 94] to split a large application into smaller segments that are individually compliant with the imposed constraint. Each segment executes sequentially, and we assume that the code and data of the segment, rather than the entire task, can be fully allocated in local memory partition. The same scheduling scheme can then be employed; after a segment of a task finishes executing, we can offload the data of the segment and load the code and data of the next segment of that task while the CPU executes a different task. We show how to extend the schedulability analysis for variable-size DMA operations to the multi-segment case in Section 3.2.2.

Task Model

We statically partition the set of tasks among the available m identical cores. On each core, we focus on fixed-priority scheduling of sporadic tasks. Specifically, the system comprises a set Γ of N sporadic tasks, $\{\tau_1, ..., \tau_N\}$, each with different priority whereby τ_1 has the highest priority and τ_N has the lowest priority. Each task τ_i is further characterized by a period T_i and a relative deadline D_i, with $D_i \leq T_i$ (constrained deadline). τ_i generates a (potentially) infinite sequence of jobs, with arrival times of successive jobs separated by at least T_i time units.

We use $C_i, load_i, unload_i$ for the CPU execution time, DMA load time and DMA unload time for a task, respectively. Note that $load_i$, and $unload_i$ denote the variable-length DMA load time and unload time of a task τ_i respectively, when considering the hardware-based DMA scheduling scheme where each core receives $1/m$ of the memory bandwidth for DMA operations. In the case of software-based TDMA scheduling of DMA, any task can be loaded or unloaded within the fixed-size time slot σ dedicated for a DMA operation. In the case of a multi-segment task covered in Section 3.2.2, we use K_i to denote the number of segments in a task τ_i, where $\{\tau_i^1, \tau_i^2, ..., \tau_i^K\}$ represents the set of segments in a task τ_i. Table 3.1 summarizes the notation used for task's parameters.

The goal of the (sufficient) schedulability analysis will be to compute the worst-case response time R_i for each task τ_i; a task set is deemed scheduable if $R_i \leq D_i$ for all tasks in the task set. Given the 3-phase execution model, it is however important to precisely define when a job is considered completed in order to determine its response time. We consider two possible cases: (1) in the first case, the job is considered completed when it finishes its execution on the core. This model is suitable for systems where any output operation (for example, writing to an output peripheral) that the task needs to carry out is performed during the execution of the task itself. Hence, as long as its core execution completes by the deadline, we consider the job feasible. (2) In the second case, the job is considered completed when its DMA unload operation finishes. This model is suitable for systems where the data written back to main memory in the unload operation contains output data, or in the case of inter-task communication, where some unloaded data needs to be passed to other tasks; in particular, the COTS platform implementation in Section 4.2 follows this approach. In this case, the unload operation of each job must complete by its deadline for the job to be feasible.

In summary, considering the two different models for scheduling of DMA operations, and the two response time models, we have four different cases. Since covering each case individually would lead to much redundancy, for simplicity we first cover the case of variable-length DMA operations with deadline based on the job execution in Section 3.2,

and we then modify the analysis for the case of fixed-length DMA operations with deadline based on the unload operation in Section 3.3.

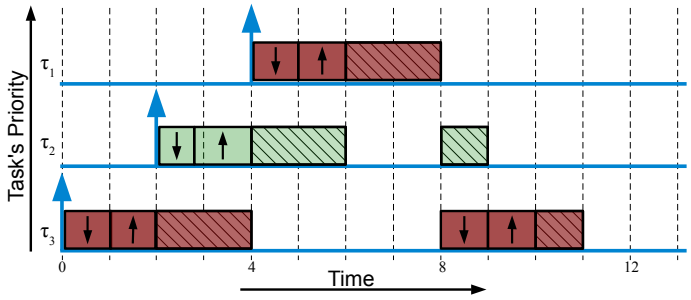
Table 3.1: Task's Parameters

Term	Definition
τ_i	a task in the system
T_i	task's minimum inter-arrival time (MIT)
C_i	task's CPU execution time including all overheads
$load_i$	task's DMA load time for its code and data sections
$unload_i$	task's DMA unload time for its data section
σ	TDMA slot size for the fixed-size DMA operation case
K_i	The number of segments a task comprises
τ_i^K	The K^{th} segment in a multi-segment task
R_i	task's response time

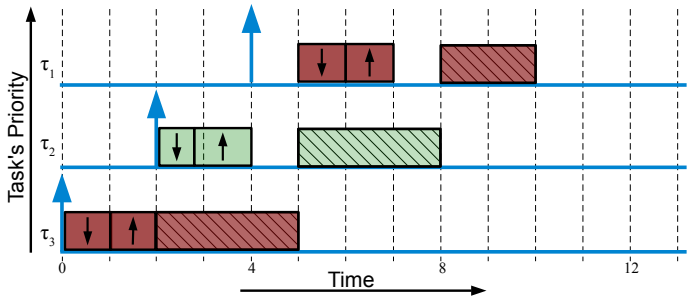
3.2 Case (I): Scheduling 3-Phase Tasks with Variable-size DMA Operations

We now detail a formal set of rules for our scheduling algorithm. As discussed above, our goal is to avoid contention for access to memory resources. Since the CPU does not access main memory while executing a task, the goal can be met as long as we ensure that CPU and DMA always access different local memory partitions. From a scheduling perspective, we achieve this by dividing the time line into a set of *time intervals*. During each interval, the CPU executes a task out of one partition, while the DMA first unloads and then loads the other partition; partitions are then swapped between CPU and DMA in the next interval. It remains to discuss how to order the loading and execution of tasks in successive intervals. In general, we would like to follow task priorities. However, as we intuitively show next, allowing preemption of either DMA operations or task execution can lead to increases in CPU stall time and schedule complexity. Therefore, our scheduling algorithm executes non-preemptively.

Notice that in the example in Figure 3.1-B there was no preemption needed since the release time of all the jobs was the same. To depict the effect of CPU preemption on the ability of hiding memory latency, Figure 3.2-A shows another example of three jobs of three different tasks arriving at different time. As shown in the figure we allow CPU preemption.



(A) Preemptive CPU Execution



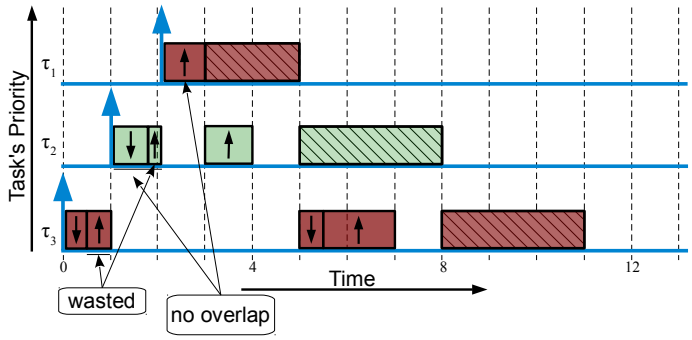
(B) Non-preemptive CPU Execution

Figure 3.2: Illustrative schedule for preemptive CPU execution

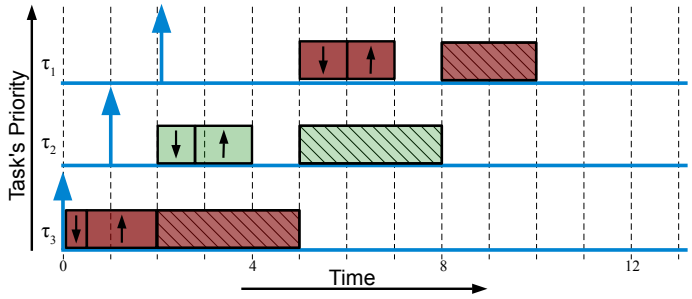
As a result, the response time of this schedule is now greater than in the previous example. The response time of the schedule is now 11 time units instead of 10 time units.

As you can notice, the preemption causes τ_3 to be loaded twice as it has been evicted by τ_1. In addition, the second load of τ_3 is not overlapped very well, only with some portion of τ_2. Furthermore, there is no DMA operation to overlap with the execution of τ_1. Prohibiting CPU preemption results in a better schedule, Figure 3.2-B. DMA operations are now overlapped with CPU execution. In addition, no extra DMA operations are required for reloading as there is no task eviction. The response time of this schedule is exactly matching the schedule in the first example in Figure 3.1-B, which is 10 time units.

In the case of having a job of higher priority task arrived at the middle of a DMA operation, allowing DMA preemption can make a difference; when the DMA preemption is allowed, the response time is again affected negatively. Figure 3.3-A depicts the case when DMA is preemptive. From the figure, it is clear that there are DMA operations that are



(A) Preemptive DMA Operation



(B) Non-preemptive DMA Operation

Figure 3.3: Illustrative schedule for preemptive DMA operation

not overlapped with CPU execution. In addition, there are aborted DMA operations which results in wasted time. The response time of this schedule is now 11 time units. On the other hand, when prohibiting DMA preemption, as shown in Figure 3.3-B, the task that is scheduled to be loaded has to complete loading and thus executes without any interruption. In this case, the result of prohibiting DMA preemption is a schedule with response time of 10 time units. Finally, prohibiting CPU and DMA preemption is motivated by the greedy choice based on a small example as shown above, and it does not generalize that a non-preemptive scheduling is better than a preemptive schedule. However, in this particular architecture, non-preemptive schedule is less complex to implement.

Based on the provided intuition, we can now formally define the scheduling rules for our algorithm, where a task is said to be active if a job of the task has been released but it has not yet completed execution:

1. The algorithm implicitly divides the schedule into a set of time intervals; scheduling decisions are only made at the beginning of an interval.

2. At the beginning of each interval, the CPU is scheduled to execute the task that has been loaded into one of the two partitions during the previous interval, if any. The DMA is scheduled to load the highest priority task among the set of remaining active tasks (if any exists) into the other partition; this involves an unload operation if needed.

3. An interval ends at the completion of the longest operation: either the CPU execution of the task or the DMA unload/load operation (if an operation is not performed, it is treated as having a length of zero).

4. If there is any ready task in the system, an interval starts at the end of the previous one; otherwise, an interval starts when the system transitions from idle to active, e.g. when a new job arrives while the system is idle.

The next section discusses how a schedulability analysis for the described algorithm can be developed and proves its correctness.

3.2.1 Schedulability Analysis of The 3-Phase Tasks with Dynamic-size DMA Operations

Given that the online scheduling algorithm enforces the aforementioned rules, we can calculate a safe bound of worst case execution time based on all tasks' parameters. Note that we assume that the task's execution time, C_i, is actually the adjusted execution time in which all the overheads are included, such as the context-switch and the DMA setup routines.

Figure 3.4 depicts an illustrative example of the worst case scheduling scenario (critical instant) for an example task set, where the task under analysis τ_3 completes execution in the fifth interval, $Intrv5$, after its arrival right after the beginning of interval $Intrv1$. At the beginning of the first interval ($Intrv1$), at time zero of this schedule, τ_5 is scheduled to execute out of partition P1 assuming it has been loaded to this partition in the previous interval which is not shown in this figure. Simultaneously, in the same interval, τ_4 is scheduled to be loaded into partition P2 after unloading the previous task, τ_u, that has been loaded into this partition two intervals ago and executed in the previous interval. Slightly after the beginning of the first interval and after all the operations have been

scheduled, three jobs of three different tasks τ_3, τ_2 and τ_1 arrive in the system. In this case τ_3 will suffer interference from the previously scheduled lower-priority tasks such as τ_5 and τ_4. In addition, τ_3 will suffer interference from all higher-priority tasks that arrive (release) before τ_3 is scheduled to load (locked), such as τ_1 and τ_2. Furthermore, τ_3 will suffer from the unload operation of τ_u. We assume that τ_u can be any task in the system with the largest data section so it takes the longest time to unload. In addition, because we do know what tasks are previously scheduled, we assume that τ_5 and τ_4 can be any two different lower-priority tasks virtually constructed with the longest execution, DMA load and DMA unload times.

To formally express the response time for τ_3, we divide the interference into two groups. First, the interference caused by τ_u and the two previously scheduled lower-priority tasks, we call this the blocking time (B). Second, the interference caused by the higher-priority tasks, we call this (H). Now the response time of τ_3 can be expressed as $R_{\tau_3} = C_3 + B + H$. As already mentioned, CPU execution and DMA operations can be overlapped during the course of an interval. Therefore, only the longest one of the two is effectively contributing to the response time of the task under analysis τ_3; therefore we take the maximum duration between the DMA operations and CPU execution. Furthermore, DMA operations are actually two parts, unloading and loading of two different tasks respectively. Therefore, the DMA operation, which is overlapped with the CPU execution, is actually the sum of an unload and a load operations. For the case in Figure 3.4, a more detailed expression of τ_3's response time can be written as follows:

$$
\begin{aligned}
R_{\tau_3} &= C_3 + B + H \\
B &= max(C_5, unload_u + load_4) \\
H &= max(C_4, unload_5 + load_1) \\
&\quad + max(C_1, unload_4 + load_2) \\
&\quad + max(C_2, unload_1 + load_3)
\end{aligned}
$$

Building on this synthetic example, we designed an algorithm that can compute the worst case response time for the task under analysis (τ_i). $hp(i)$ and $lp(i)$ are the sets of higher-priority tasks and lower-priority tasks than τ_i, respectively. J_{hp} is the set of jobs of all higher priority tasks. τ_{l1} and τ_{l2} are the two worst lower priority tasks similar to τ_4 and τ_5 in the figure, respectively. Algorithm 3.1 shows the steps of how to calculate the worst case response time of any task under analysis τ_i. The algorithm addresses the problem in three main steps, calculate the blocking time B, calculate the interference caused by

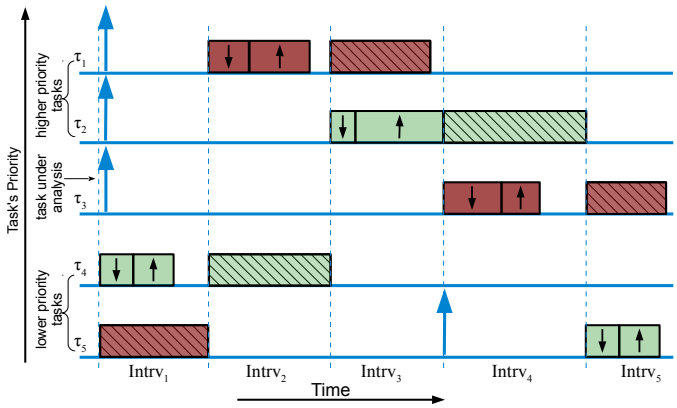


Figure 3.4: An illustrative schedule of a worst case response time of a task under analysis

the higher-priority tasks H, and apply the recursive response time analysis to determine the final response time for the task under analysis. Calculating the blocking time B is straight forward as the tasks involved in B are already scheduled by the time the task under analysis is released. Step 2 in Algorithm 3.1 calculates B. On the other hand, calculating H is not straight forward because we do not assume in which order the jobs of the higher-priority tasks arrive. Therefore, we do not know which CPU execution is overlapped with DMA operation. As a result, the algorithm methodology is to determine an upper bound regardless of the order in which the jobs of the higher-priority tasks are released.

The algorithm constructs three lists, E for execution times, LD for DMA load times and UD for DMA unload times. The execution times, C, of all released higher-priority jobs are inserted into the E list. In addition, τ_{l1} is inserted into the E list as it is overlapped with some DMA load operation of a higher-priority job. For example, in $Intrv_2$ of Figure 3.4, C_4 overlaps with $load_1$. Similarly, the LD list contains all DMA load times for all higher-priority jobs and the DMA load time for the task under analysis, τ_i, as it has to overlap with the execution of some higher-priority job. For example, in $Intrv_4$ of Figure 3.4, $load_3$ overlaps with C_2. Finally, the UD list contains the DMA unload times for all higher-priority jobs and the DMA unload times of τ_{l1} and τ_{l2}.

The execution of the task under analysis τ_i can overlap with the DMA unload of a higher-priority job while it is not contributing to the response time of the task under analysis τ_i. For example, in $Intrv_5$ of Figure 3.4, C_3 overlaps with $unload_2$. However, this overlapping DMA operation is not contributing to the response time of τ_3. Therefore, not

all DMA unloads of the higher-priority jobs are contributing to the response time of the task under analysis τ_i. However, we do not know which one it is. As a result, the UD list contains the DMA unloads of all higher-priority jobs. By doing this, the number of elements in UD will be one element more than the number of elements in each one of the other lists, E and LD. As shown in the example in Figure 3.4, the number of interfering intervals I is the number of the higher-priority jobs plus one, $I = len(J_{hp}) + 1$, where $len(J_{hp})$ is the number of element in J_{hp}. Therefore, the number of elements in each list E and LD is I and the number of elements in the list UD is $I + 1$. In fact, $I + 1$ is the number of scheduled intervals between the release time and the execution time of the task under analysis τ_i.

After constructing the three lists, the algorithm picks the longest times from each list to form an upper bound worst-case response time. The algorithm does that in two steps. First, the two DMA lists, LD and UD, are sorted in a descending order from the worst (longest) at the top to the best (shortest) at the bottom. After that, a unified DMA-times list (DMA) is constructed by selecting the worst combinations of the two DMA lists, LD and UD. Basically, this is done by selecting the first I elements from the two sorted lists, LD and UD, and insert them into the unified DMA list as follows: $DMA_j = LD_j + UD_j$ for $j = 1....I$. By doing this we assure that the algorithm only ignores the least contributing (shortest unload time) element in the UD list, which is what we need. Second, since the CPU execution and the DMA operation can overlap, the algorithm selects one element from the both lists, DMA and E, for each interval based on the most contributing elements (the longest ones). It does that by merging the DMA and E lists into one list M (Maximums). Then, it sorts the M list in a descending order from the longest at the top to the shortest at the bottom. Since we only have I interfering intervals, the algorithm sums the first I elements from the list M and concludes H, which is the interference contributed by the higher-priority jobs, as follows: $H = \sum(M_j)$ for $j = 1....I$. The selected most contributing elements can be all from the list E or can be all from the list DMA or can be mix of both. Algorithm 3.1 calculates H in steps 8-16.

The final step in calculating the response time is to apply the standard response time analysis where the response time is defined as a function of itself. Algorithm 3.1 does this in steps 3-7 and 17. In step 7, Algorithm 3.1 updates J_{hp}, which is the set of all higher-priority jobs, in each iteration to accommodate all higher-priority jobs that will be released before the execution of the tasks under analysis τ_i, $\tau_j \in hp(i)$. We used the expression $R_i - C_i$ instead of just R_i because, as introduced earlier, once a task is scheduled to load it is locked and it will execute to finish with no preemption.

Lemma 3.1. *Let J_{hp} be the set of higher priority jobs that interfere with the task under analysis τ_i. Then the value of H computed by Algorithm 3.1 is a safe upper bound to the*

Algorithm 3.1 Calculating the safe execution bound of the task under analysis τ_i

1: $\tau_u \in \Gamma$; τ_{l1}, τ_{l2} two virtual tasks $\in lp(i) : \tau_{l1} \neq \tau_{l2}$;
2: $B = max(C_{l2}, unload_u + load_{l1})$
3: $R_i \leftarrow C_i + B$
4: set $PrevR_i \neq R_i$
5: **while** $(R_i \neq PrevR_i)$ **do**
6: $\quad PrevR_i \leftarrow R_i$
7: $\quad J_{hp} \leftarrow \bigcup_{\tau_j \in hp(i)} \bigcup_{1 \ldots \left\lceil \frac{R_i - C_i}{T_j} \right\rceil} \tau_j$
8: $\quad E \leftarrow C_{l1} \bigcup C_{J_{hp}}$
9: $\quad LD \leftarrow load_i \bigcup load_{J_{hp}}$
10: $\quad UD \leftarrow unload_{l1} \bigcup unload_{l2} \bigcup unload_{J_{hp}}$
11: $\quad dsort(LD)$
12: $\quad dsort(UD)$
13: $\quad DMA \leftarrow \bigcup_{1 \leq j \leq len(LD)} LD_j + UD_j$
14: $\quad M \leftarrow E \bigcup DMA$
15: $\quad dsort(M)$
16: $\quad H \leftarrow \sum_{1 \leq j \leq len(E)} M_j$
17: $\quad R_i = C_i + B + H$

length of all intervals where higher priority jobs delay the task under analysis.

Proof. As discussed above, the maximum number of intervals where higher priority jobs interfere with the task under analysis τ_i is $I = len(J_{hp}) + 1$: one interval where the first higher priority jobs is loaded, plus $len(J_{hp})$ intervals to execute each higher priority job. In each interval there is one CPU execution that overlaps with a DMA load/unload operation; therefore, the length of each interval is the maximum of the two. We can then obtain H as the sum of the lengths of I intervals, where each length is either a CPU execution time or a DMA load/unload time. Now, our algorithm simply takes all CPU execution times and all DMA times in those intervals, as shown above, and picks the maximum I elements among all of them; hence, this must result in a upper bound to the actual combined length of the I intervals.

It remains to show that the way the DMA-time list (DMA in step 13) is constructed indeed leads to the worst-case. Let DMA_j be the j-th element of the DMA list sorted in descending order (similarly for other lists). Assume that in step 16 the algorithm selects the maximum k elements from the E list and the maximum k' from the DMA list, where

$k + k' = I$. It is easy to show that for any $k', \sum_{1 \leq j \leq k'} DMA_j$ is the maximum combined DMA time that can be constructed out of k' unload times and k' load times; this is because $DMA_j = LD_j + UD_j$ and both LD and UD are sorted in descending order, hence any element $LD_j, j \leq k'$ is larger than any element $LD_j, j > k'$ (similarly for UD). By contradiction, assume that the worst case is instead found by picking $k' + 1$ elements from DMA and $k - 1$ from E. Then the contribution of DMA times would be increased by $\sum_{1 \leq j \leq k'+1} DMA_j - \sum_{1 \leq j \leq k'} DMA_j = DMA_{k'+1}$ and the contribution of E would be decreased by E_k. However, since the algorithm selected k and k' elements out of E and DMA, respectively, by picking the maximum elements from the combined list $E \bigcup DMA$, it follows that $DMA_{k'+1}$ cannot be larger than E_k. This causes a contradiction, hence the lemma holds. □

Theorem 3.1. *Algorithm 3.1 calculates a safe upper bound to the response time of the task under analysis τ_i.*

Proof. Algorithm 3.1 calculates the upper-bound response time for the task under analysis τ_i by summing the three contributing times, the blocking time caused by lower-priority jobs B, the interference caused by higher-priority jobs H, and the execution time of the task under analysis C_i. Therefore, the response time of task under analysis τ_i is $R_i = B + H + C_i$. R_i is a safe upper bound since it is a sum of upper bounds: in particular, B is an upper bound because we are computing it as the maximum of the computation time and DMA load/unload of any task that can interfere with τ_i in the first interval of the busy period (during which τ_i arrives). Now note in our system, when a task is scheduled to load, it will be executed in the next interval with no preemption. Therefore, if R_i is the response time of the task under analysis (computed at the previous iteration), then the maximum number of instances of higher priority task τ_j that can interfere with τ_i is $\lceil \frac{R_i - C_i}{T_j} \rceil$. Since J_{hp} is computed based on this formula at step 7 of the algorithm and because of Lemma 3.3, it then follows that H is also a safe bound. Finally, C_i is an upper bound by definition.

It remains to show that the release time of the task under analysis τ_i and all higher-priority tasks, which are arriving at the same time just $\varepsilon > 0$ time after an interval has started, with ε arbitrarily small, is indeed the worst case. In particular, if τ_i arrives before the interval starts, then it will be locked and scheduled to load and execute without preemption before the higher-priority tasks, which cannot lead to the worst case. If instead τ_i arrives δ time after the beginning of the first interval, but still within the interval, then the schedule would not change but the response time of τ_i would be reduced by δ, which is not the worst case either. Similarly, if a higher-priority task arrives before the interval starts, τ_i will not suffer any interference from the lower-priority tasks, reducing its response

time. Finally, if a higher-priority task arrive within the same interval but δ time after the beginning of the interval, the number of higher-priority jobs that will be released before τ_i is scheduled is $\lceil \frac{R_i - C_i - \delta}{T_j} \rceil$. Since this number is never larger than $\lceil \frac{R_i - C_i}{T_j} \rceil$, it follows that releasing the task ε time after the start of the interval always maximizes the worst case response time of τ_i. Hence, the theorem follows. □

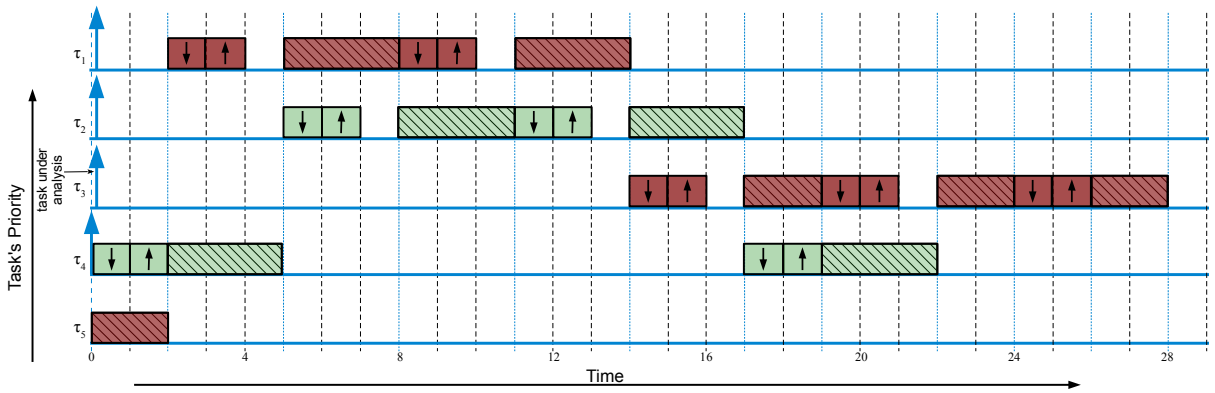


Figure 3.5: Illustrative schedule of multi-interval tasks

3.2.2 Analysis of Multi-Segment Tasks

In this section, we extend the schedulability analysis to cover the case of multi-segment tasks. Scheduling Rules 1-4 still apply, except that when the DMA is scheduled to load a task, only the code and data of the next segment of the task is loaded, and in the next interval, the CPU only executes the code of that segment (rather than the entire task). Note that while a segment of a task is executed by the CPU, the DMA cannot load the next segment of the same task, since a latter segment might depend on a previous segment's data. As a matter of fact, Rule 2 specifies that the DMA can only load a different task than the one executed by the CPU in the same interval. As a result, if a task τ_i is composed of K_i segments, even without any interference by other tasks it will take $2K_i$ intervals to complete the task: one interval to load a segment/unload the previous one and one interval to execute the segment itself.

A worst case scheduling scenario for a task under analysis τ_3 is shown in Figure 3.5, where the task under analysis τ_3 and all higher-priority tasks, τ_1 and τ_2, arrive at the same time just after the lower-priority task τ_4 is scheduled to load. Our goal is to compute the

response time R_{τ_3} for the last interval of the task under analysis, which is the interval where the last segment τ_3^3 of τ_3 executes ($[26, 28]$ in Figure 3.5). Similarly to the single-interval case, we compute R_{τ_3} as the sum of three components: $R_{\tau_3} = C_3^{K_3} + B + F$. B is the blocking time caused by the previously scheduled intervals ($[0, 2]$ in the figure), while $C_3^{K_3}$ is the computation time of the last segment of τ_3. Finally, F is the interference caused by all higher-priority intervals and lower-priority intervals or the first $K_i - 1$ intervals of τ_i ($[2, 26]$ in the figure). Notice that in the worst case, whenever a segment other than the last one of the task under analysis is executed by the CPU ($[17, 19]$ and $[22, 24]$ in the figure), no higher priority job is active; hence, either no DMA operation is overlapped with it ($[22, 24]$), or the DMA load for a lower priority job can be overlapped ($[17, 19]$). In the latter case, during the next interval the lower priority job will be executed on the CPU. In summary, for a task under analysis τ_i, F is composed by $len(I_{hp})$ intervals, where I_{hp} is the set of interfering higher-priority segments, plus $K_i - 1$ intervals for the segments of the task under analysis (except the last), plus K_i intervals where either a lower priority job is executing, or a segment of the task under analysis is loaded without being overlapped with any execution. Note that in the case of Figure 3.5 where $K_3 = 3$, this results in $4 + (3 - 1) + 3 = 9$ intervals.

Algorithm 3.2 calculates the worst-case response time for task under analysis τ_i. The higher-priority intervals, which belongs to higher-priority jobs, are expressed as $hpi(i)$ and the lower-priority jobs' intervals, which belongs to lower-priority jobs, are expressed as $lpi(i)$. The response time of τ_i is expressed as $R_i = C_i^{K_i} + B + F$, which is the response time of the last interval ($\tau_i^{K_i}$) of τ_i. The blocking time B is calculated as in Algorithm 3.1 except it is expressed in terms of intervals. I_{hp} is the set of all interfering higher-priority intervals. Since $\tau_i^{K_i}$ can also be interfered by the lower-priority intervals or by the precedent intervals of τ_i, Algorithm 3.2 considers both the higher-priority intervals and the lower-priority intervals in the steps 9-17. The lists E_{hp}, LD_{hp} and UD_{hp} contains the execution times, the DMA load times and the DMA unload times of all interfering higher-priority intervals respectively. On the other hand, the list E, LD and UD contains the execution times, the DMA load times and the DMA unload times of the precedent intervals of τ_i and the possibly $K_i - 1$ interfering lower-priority intervals respectively. In general, we do not know which lower-priority interval will interfere with the interval $\tau_i^{K_i}$. Therefore, we always assume it to be the longest lower-priority interval τ_l^B. τ_l^B is a synthetic interval that has the longest execution, DMA load and DMA unload times. Algorithm 3.2 calculates the upper-bound interference F the same way Algorithm 3.1 calculates the upper-bound interference caused by higher-priority jobs H. F is the summation of the upper-bound interference caused by $len(I_{hp}) + 2K_i - 1$ intervals. F is calculated in steps 9-23. Note that in step 8, Algorithm 3.2 forms the set I_{hp}, which is the set of all interfering higher-

Algorithm 3.2 Calculating the safe execution bound of the task under analysis τ_i when tasks are multi-interval tasks

1: $\tau_u \in \Gamma$; τ_{l1}, τ_{l2} two synthetic tasks $\in lpi(i) : \tau_{l1} \neq \tau_{l2}$
2: $\tau_l^B \in lpi(i)$ is a synthetic longest lower-priority interval
3: $B = max(C_{l2}^S, unload_u^S + load_{l1}^S)$
4: $R_i \leftarrow C_i^{K_i} + \mathrm{B}$
5: set $PrevR_i \neq R_i$
6: **while** $(R_i \neq PrevR_i)$ **do**
7: $\quad PrevR_i \leftarrow R_i$
8: $\quad I_{hp} \leftarrow \bigcup_{\tau_j \in hp(i)} \bigcup_{1 \ldots \left\lceil \frac{R_i - C_i^{K_i}}{T_j} \right\rceil} \bigcup_{v=1 \ldots K_j} \tau_j^v$
9: $\quad E_{hp} \leftarrow C_{l1}^S \bigcup C_{I_{hp}}$
10: $\quad E \leftarrow (\bigcup_{1 \leq v < K_i} C_i^v) \bigcup (\bigcup_{1 \leq v < K_i} C_l^B)$
11: $\quad E_{global} \leftarrow E_{hp} \bigcup E$
12: $\quad LD_{hp} \leftarrow load_{I_{hp}}$
13: $\quad LD \leftarrow (\bigcup_{1 \leq v \leq K_i} load_i^v) \bigcup (\bigcup_{1 \leq v < K_i} load_l^B\ )$
14: $\quad LD_{global} \leftarrow LD_{hp} \bigcup LD$
15: $\quad UD_{hp} \leftarrow unload_{l1}^S \bigcup unload_{l2}^S \bigcup unload_{I_{hp}}$
16: $\quad UD \leftarrow (\bigcup_{1 \leq v < K_i} unload_i^v) \bigcup (\bigcup_{1 \leq v < K_i} unload_l^B)$
17: $\quad UD_{global} \leftarrow UD_{hp} \bigcup UD$
18: $\quad dsort(LD_{global})$
19: $\quad dsort(UD_{global})$
20: $\quad DMA \leftarrow \bigcup_{1 \leq j \leq len(LD_{global})} (LD_{global_j} + UD_{global_j})$
21: $\quad M \leftarrow E_{global} \bigcup DMA$
22: $\quad dsort(M)$
23: $\quad F \leftarrow \sum_{1 \leq j \leq len(E_{hp}) + 2K_i - 1} M_j$
24: $\quad R_i = C_i^{K_i} + B + F$

priority intervals, by accommodating all the intervals (τ_j^v) of the released jobs (τ_j) of all

higher-priority tasks (τ_i), where $\tau_j \in hp(i)$.

Lemma 3.2. *Let I_{hp} be the set of segments of higher priority jobs that interfere with the task under analysis τ_i. Then the value of F computed by Algorithm 3.2 is a safe upper bound to the length of interfering intervals with the exclusion of the interval during which τ_i arrives.*

Proof. As mentioned above, F is composed of the interference caused by higher-priority intervals and the interference caused by the first $K_i - 1$ intervals of τ_i or by the possible lower-priority intervals. The way we calculate the maximum interference of the higher-priority intervals is similar to calculating H in Algorithm 3.1, e.g., each segment of a higher-priority interfering job will contribute one additional interval to the length of F; furthermore, DMA load and unload operations can overlap with other executions in other intervals.

Since we calculate the end-end response time of τ_i, the last interval τ_i^K is proceeded by $K_i - 1$ CPU intervals and K_i DMA intervals. Therefore, in the worst case there will be $2K_i - 1$ scheduled intervals before the execution of τ_i^K (without considering higher-priority tasks). These scheduled intervals might overlap with lower-priority intervals. As a result, we have two cases, in the first case, when there are no active lower-priority intervals, each CPU interval of τ_i has to be proceeded by the corresponding DMA interval to load it. Therefore, the CPU intervals of τ_i suffer the delay caused by the corresponding DMA intervals. However, this is not the worst case. The second case is when there are long lower-priority intervals to overlap with the intervals of τ_i. In this case, the overlapping lower-priority intervals are selected as the contributing intervals if they are longer than the corresponding interval of τ_i; otherwise, it is similar to the first case. Therefore, it is safe to always consider that there will be a lower-priority interval to overlap with; and that is τ_l^B, which is the longest lower-priority interval. To conclude the proof, it suffices to notice that the same reasoning as in in Lemma 3.3 can be used to show that picking the largest contributing intervals leads to a safe upper bound. □

Theorem 3.2. *Algorithm 3.2 calculates a safe upper bound to the response time of the multi-interval task under analysis τ_i.*

Proof. Algorithm 3.2 calculates the upper-bound response time for the task under analysis τ_i by summing the three contributing times, the blocking time caused by lower-priority intervals B, the interference F caused by all interfering intervals in the system, and the execution time of the last interval of the task under analysis $C_i^{K_i}$. Therefore, the response

time of task under analysis τ_i is $R_i = B + F + C_i^{K_i}$. Similarly to the proof of Theorem 3.1, this is a safe upper bound since it is a sum of upper bounds. In particular, if R_i is the response time of the task under analysis computed at the previous iteration, then the maximum number of instances of higher priority task τ_j that can interfere with τ_i is again $\lceil \frac{R_i - C_i^{K_i}}{T_j} \rceil$. Hence, by virtue of the computation of I_{hp} at step 8 of the algorithm and because of Lemma 3.5, F is a safe upper bound. $C_i^{K_i}$ is an upper bound by definition; finally, the same reasoning as in Theorem 3.1 can be used to show that B is also a safe upper bound, and that the release time of the task under analysis τ_i and all higher-priority tasks, which are arriving at the same time immediately after an interval has started, is indeed the worst case. □

3.3 Case (II): Scheduling 3-Phase Tasks with Fixed-size DMA Operations

Unlike the previous case, where each core is assumed to have a guaranteed memory bandwidth, here we explicitly schedule TDMA slots for each core. Several schemes are known to fairly share a single resource across different consumers. For the scope of our design, we employ a TDMA scheme to serialize task load/unload operations among m cores. The main advantage of the TDMA scheme lies in its simplicity of implementation. In order to perform TDMA-based scheduling of the DMA, time is partitioned into slots of fixed size. In each slot, only a single DMA operation can be performed, either a task load or unload. The slot size is chosen to ensure that the task with the largest footprint in the system can be loaded within the slot time window. Figure 3.6 depicts the sequence of operations in our TDMA scheme for a system with two cores; note that the schedule of execution phases on the core, as well as the sequence of intervals, follows the same Rules 1-4 in Section 3.2 that are used in the variable-size DMA case. In particular, Figure 3.6 depicts three tasks scheduled on one core. Up arrows in blue color represent the arrival times of the considered tasks; we use colors for two different partitions. A task can only run after its load operation has been completed and the previous task on the other partition has completed (see τ_2 to τ_3 and τ_1 to τ_2 for example of the two cases). There might be slots where no load/unload is performed. This happens at time 8: τ_1 finishes right after the beginning of the slot, so both partitions are full at the beginning of the slot and neither load nor unload can be performed. Effectively, the slot is wasted.

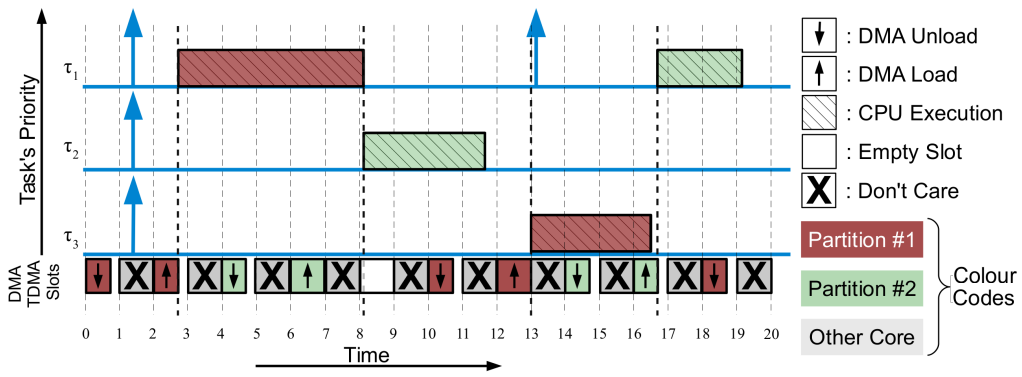


Figure 3.6: Scheduling CPU, DMA and local memory with fixed-size time slots

As depicted in Figure 3.6, the shared system DMA is alternatively assigned to transfer

data for a specific core. Within a single slot, either an unload operation for a previously running task or a load operation for the next scheduled task is performed. The specific operation to be performed is decided as follows:

Rule 1: If a load operation can be performed, a load operation is programmed on the application DMA;

Rule 2: If a load cannot be performed and there is a previously running task to be unloaded, an unload operation is programmed on the application DMA.

Note that Rule 1 can be activated by the following conditions: (i) at least one of the two SPM partitions is available (i.e. has been previously unloaded), and (ii) a task has been released and is ready to be loaded. Similarly, Rule 2 can be activated if no load can be performed, at least one partition is not empty and the task loaded on that partition has completed.

3.3.1 Schedulability Analysis of The 3-Phase Tasks with Fixed-Size DMA Operations

Given the scheduling strategy described in Section 3.3, we can calculate a safe bound on the worst case execution time based on all tasks' parameters in an approach similar to to the previous case in Section 3.2.1. Note that we assume that the task's execution time, C_i, is actually the adjusted execution time in which all the overheads are included, such as the context-switch and the DMA setup routines. Also note that for simplicity we discuss the case with $m = 2$ cores, since it is used in the implementation with fixed-size DMA operations described in Section 4.2, but the analysis could be trivially extended to account for any number of cores.

Figure 3.7 depicts an illustrative example of the worst case scheduling scenario (critical instant) for an example task set where τ_3 is the task under analysis. The schedule depicts a busy period where τ_3 suffers interference from two higher-priority tasks, τ_1 and τ_2. As in Section 3.2.1, we consider the busy period as composed by a sequence of *scheduling intervals* $Interval_1, Interval_2, Interval_3, Interval_4$ (each bounded by bold vertical lines in the figure), followed by a *final interval* $Interval_F$. During each scheduling interval, only one blocking or interfering task runs. During the final interval, the task under analysis runs. Each scheduling interval always starts with a CPU execution and ends either when the CPU finishes executing the task or when the next task finishes being loaded by the

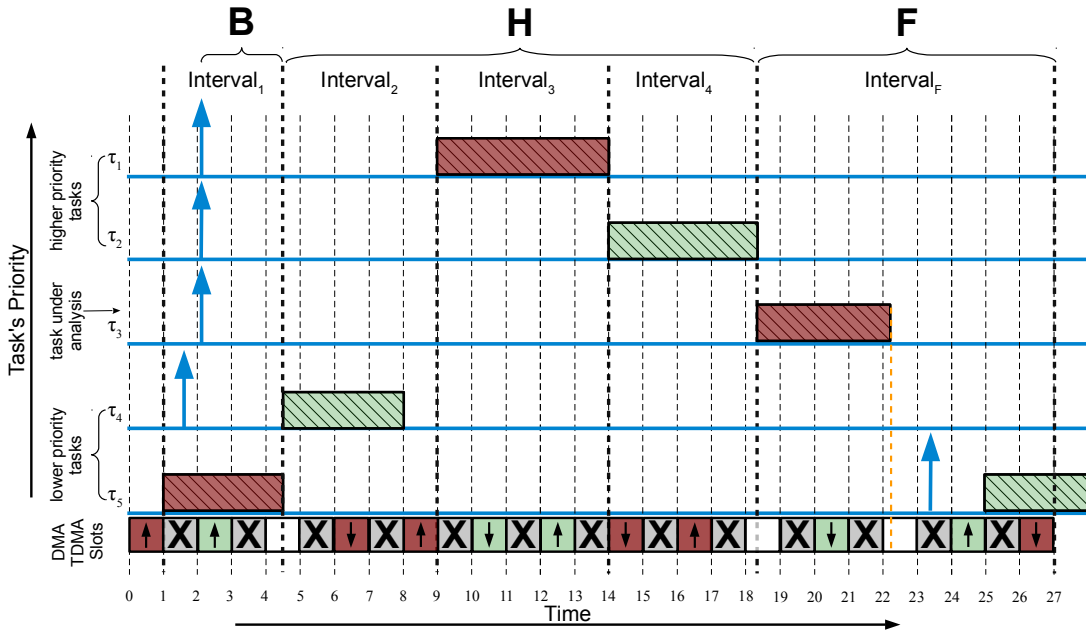


Figure 3.7: Example schedule, intervals are highlighted.

DMA, whichever happen last; at this point, the next interval starts with the execution of the loaded task. The final interval starts with the execution of the task under analysis and finishes when the task under analysis is unloaded.

We say that a scheduling interval is *CPU-bound* when it ends with CPU execution (ex: $Interval_1$, $Interval_3$ and $Interval_4$ in the figure), and *DMA-bound* when it ends with DMA load operation (ex: $Interval_2$). The length of a scheduling interval is the maximum between the execution time of the task running in the interval and the DMA operations required to load the next task. We denote the size of the TDMA slot as σ; since in the worst case a load/unload operation can occupy the entire slot, we upper bound the length of DMA operations as a multiple of σ.

Building on the above-mentioned definitions, we can use Algorithm 3.1 detailed in Section 3.2.1 to compute the worst case response time for the task under analysis, by showing that the problem is equivalent to the one in Section 3.2.1. Algorithm 3.1 computes the response time by adding three components: (1) the blocking time B caused by a lower priority task that starts executing before the beginning of the busy period; this is $Interval_1$ executing task τ_5 in the figure; (2) the interference H comprising the remaining scheduling

intervals in the busy period, which are $Interval_2, Interval_3$ and $Interval_4$ in the figure. The number of such intervals is equal to the number of interfering higher priority jobs plus one, since an extra lower priority job that starts loading before the beginning of the busy period (τ_4 in the figure) can execute within the busy period itself; (3) the computation of the task under analysis (C_i). The algorithm builds a list of DMA times and computation times for tasks executed in H, then it derives a provably safe bound on the length of H using standard response time iteration.

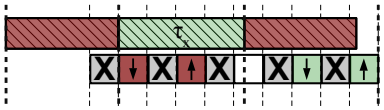
Compared to the analysis in Section 3.2.1, the presented analysis in this section differs in three aspects. First, in this section we use fixed-size DMA operations, while Section 3.2.1 employs dynamic-size DMA operations. Therefore, we need to discuss how to compute the length of the DMA operations that are inserted in the list of DMA times. Second, we need to recompute the length of the blocking time B since the next task to be loaded is determined at a different time. Finally, unlike the analysis in Section 3.2.1, here we consider the task under analysis finished when the task is unloaded, at the end of $Interval_F$. Consequently, we can use the same algorithm to compute the worst case response by replacing C_i with the length of $Interval_F$. We address each point in sequence.

Scheduling Intervals in The Busy Period (H)

When the system is busy with both SPM partitions occupied and at least one pending task, within each interval we need to first unload the previous partition and then load it with the next task. Therefore, for any scheduling interval, it will require four TDMA slots ($4 \cdot \sigma$) to load the next task if the interval was preceded by another DMA-bound interval, such as for $Interval_3$ in the figure. On the other hand, if the interval is preceded by a CPU-bound interval, it might require up to five TDMA slots ($5 \cdot \sigma$) to finish loading the next task in the worst case, as for $Interval_2$. This is because the CPU-bound interval can induce an unused empty TDMA slot in the next interval (slot [4:5] in the figure).

As a result, the length of any scheduling interval can be computed as either $\max(C_i, 4 \cdot \sigma)$ or $\max(C_i, 5 \cdot \sigma)$. We now formally prove that for any CPU-bound interval to cause the worst case scenario, with the exception of the first interval $Interval_1$, the CPU execution has to be strictly longer than four TDMA slots ($4 \cdot \sigma$).

Lemma 3.3. *For any scheduling interval in H, no extra empty slot will be induced in the next interval unless the length of the CPU execution is strictly greater than four TDMA slots ($4 \cdot \sigma$).*



Proof. We show that any scheduling interval in H with CPU execution less than $4 \cdot \sigma$ cannot induce an empty slot in the next interval. By considering the figure above, the execution of τ_x in the middle interval is greater than $4 \cdot \sigma$, and it induces an empty slot in the next interval. Since both partitions must be full during the execution of intervals in H, it follows that the interval to which τ_x belongs must include both a load and an unload operation; in the worst case showed in the figure, the interval could start with the unload operation. Still, clearly if the execution time of τ_x is reduced to less than $4 \cdot \sigma$, the interval will finish before the next slot assigned to the core under analysis is reached, and hence no empty slot will be induced. Note that if the execution of τ_x is reduced further, it could make the interval into an DMA-bound interval, which would end right after finishing the load of the next task; thus, the next interval would not suffer from an empty induced slot either. □

Based on Lemma 3.3, we can construct the list of DMA times used by the algorithm as follows: we insert in the list a number of $5 \cdot \sigma$ time values equal to the number of tasks executed in H with length greater than $4 \cdot \sigma$, plus one task (to account for the task in $Interval_1$, which can cause an extra empty TDMA slot as in the figure). The remaining DMA times in the list are equal to $4 \cdot \sigma$.

Critical Instant and Blocking Time (B)

At the beginning of the example schedule in Figure 3.7, the system has two free local SPM partitions at time zero. In $Interval_1$, the task under analysis τ_3 is released along with all higher-priority tasks after an arbitrarily small time (ε) when all free partitions have been loaded or have started loading lower-priority tasks (τ_5 and τ_4); this is ε after time 2 in the figure. The task under analysis τ_3 cannot run until the pre-loaded lower-priority tasks (τ_5 and τ_4) plus all higher-priority tasks (τ_1 and τ_2) finish execution. We now prove that the discussed scenario is indeed the critical instant for our system, leading to the worst case response time for the task under analysis.

Lemma 3.4. *The critical instant is produced when the task under analysis τ_i and all higher priority tasks arrive immediately after a lower priority task has started loading into a partition, and the other partition was loaded with another lower priority task as late as possible (i.e., two slots before).*

Proof. We first show that in the worst case, both τ_i and all higher priority tasks must arrive ε time after the beginning of a slot where a lower priority task is loaded. If either τ_i or a higher priority task would arrive at or before the beginning of the slot, then such task would be loaded and executed in place of the lower priority task. Hence, the length of the busy period would decrease by one scheduling interval, which cannot produce the worst case response time for τ_i. If instead τ_i arrives some δ time later during the busy period, then the finishing time of τ_i would not change, but the response time of τ_i would decrease by δ. Finally, if a higher priority task arrives later during the busy period, the number of interfering jobs of the task could only be lower or equal compared to releasing it immediately after the beginning of the slot. Hence, the described activation pattern must lead to the critical instant.

For what concerns the lower priority task pre-loaded in the other partition, it suffices to notice that loading the task as late as possible (i.e., two slots before τ_i arrives, which is slot [0:1] in Figure 3.7) maximizes the amount of execution of the task within the busy period. □

Based on Lemma 3.4, the worst case blocking time B can be obtained as the length of $Interval_1$ minus σ, where the length of $Interval_1$ is bounded by $\max(C_u, 2 \cdot \sigma)$; here, C_u represents any low priority task executed in $Interval_1$, while the length $2 \cdot \sigma$ accounts for the fact that the next task is loaded in the second slot of the interval (slot [2:3] in the figure). Similar to Section 3.2.1, since we can make no assumption on which lower priority tasks execute in $Interval_1$ and $Interval_2$, the algorithm simply considers the two lower priority tasks with the longest execution times.

Final Interval (F)

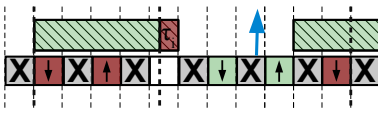
The length of the final interval $Interval_F$ can be computed as $\max(C_i + 5 \cdot \sigma, 7 \cdot \sigma)$, where τ_i is the task under analysis. In the example depicted in Figure 3.7, the length of $Interval_F$ is $C_3 + 5 \cdot \sigma$. The other case can happen when C_3 is short enough and slot [22:33] is utilized, in the worst case, to load the next task. In this situation, up to seven TDMA slots are required to finish unloading τ_3, as formally proven below.

Lemma 3.5. *The length of $Interval_F$ is upper bounded by* $\max(C_i + 5 \cdot \sigma, 7 \cdot \sigma)$.

Proof. Similar to scheduling intervals in H, we need to consider two cases: (1) the length of the interval is bounded by C_i plus the time required to unload the task; (2) the length of the interval is bounded by the time required to unload/load the other partition before

unloading C_i. For the first case, note that differently from scheduling intervals in H, there might be no pending task at the start of $Interval_F$, since the last task in the busy period (task under analysis) is running. Therefore, a new job of any task could be release later during $Interval_F$ and start a load operation. In particular, the new job could arrive just before the unload of the task under analysis (τ_i), as shown in Figure 3.7. In this case, since there is one free partition and load operations have priority over unload operations (Rules 1,2), the new job has to be loaded first; thus, the unload of τ_i is delayed by up to $5 \cdot \sigma$ in the worst case (one empty slot plus four other slots, as shown in Figure 3.7 for slots [22:27]; no more than 5 slots are possible since after the load at [24:25], both partitions are full and thus an unload must happen next). In this case the length of $Interval_F$ is upper bounded by $C_i + 5 \cdot \sigma$.

The following figure shows the other case where the length of $Interval_F$ is $7 \cdot \sigma$ in the worst case. This case happens when the execution of the task under analysis is very small to the point that $C_i + 5 \cdot \sigma$ is smaller than the required number of TDMA slots to actually unload τ_i. When CPU execution of τ_i is sufficiently small, the load of the next task has to be after $5 \cdot \sigma$ at most regardless of the release time of the next task, otherwise τ_i would be unloaded by the fifth slot. If the next task is indeed loaded before the unload of τ_i as shown in the figure, then in the worst case it takes two more slots to unload τ_i (given that both partitions are full after loading the next task), hence resulting in a bound of $7 \cdot \sigma$. To conclude, by taking the maximum of the two cases we guarantee to capture the worst case.



□

Theorem 3.3. *The worst-case response time of the task under analysis (R_i) is $R_i = B + H + F$.*

Proof. Based on the proofed lemmas (lemmas: 3.3, 3.4, and 3.5), the theorem directly follows. □

3.4 Fault-Tolerant Scheduling of 3-Phase Tasks

The ability to recover from soft errors and effectively extend the Mean Time To Failure (MTTF) is particularly relevant when considering safety-critical systems. This is because real time embedded devices are often deployed in hostile environments such as production plants, aircraft, and satellites. In such environments, extended exposure to various kind of radiations, such as alpha particles, high and low energy cosmic rays, as well as strong electromagnetic fields, can increase the probability of temporary "bit flips", i.e. soft errors, in the circuitry. The rate at which soft errors occur is called the Soft Error Rate (SER). The commonly used unit of measure for SER is the Failure In Time (FIT). One FIT is equivalent to one failure in 10^9 device hours.

In the previous sections, the proposed system attains the goal of achieving predictability in a multi-core environment. However, in the event of a memory error, it does not provide any recovery countermeasure. In order to augment the predictable SPM-centric management to recover from a memory error, we leverage the existing redundancy that is built-in by design in multi-phase task system. In our original system, a running task has two copies, one in the SPM and one in the main memory. Only the read-only data is considered as a replica, since the R/W data in the SPM might be altered during the execution of the task. In the proposed fault-tolerant system, two copies of the task are kept inside the main memory to recover from memory errors. The proposed recovery mechanism assumes that only one memory error could occur in every two periods of any task in any memory module. Since the period of a real-time task is typically tens or hundreds of milliseconds long, we deem this assumption to be satisfied in the vast majority of embedded systems. In addition, we assume the existence of Error-Correcting Code (ECC) unit that is capable of detecting memory bit flip errors that cannot be automatically corrected in hardware, such as double-bit errors. First, we describe how the overall system with redundant task copies works, then we explain how a fault in each is handled.

Note that, the recovery mechanism can be applied to both variable-size DMA operations and fixed-size DMA operations. For simplicity, in what follows we describe the recovery mechanism for the fixed-size DMA operations, since this is the case for the fault-tolerant implementation discussed in Section 4.2.

Conceptually, there are two identical copies of a task in main memory during regular operation, one is considered as a working copy and the other is a backup copy. The task is usually loaded to the SPM from the working copy. After the task completes its execution from the SPM, it is first unloaded to update the working copy in the main memory. And then, upon the successful unload, another unload is initiated to update the backup copy.

Memory errors are detected either while the CPU is executing a task from the local SPM, or while the DMA is loading or unloading a task. In the former case, the execution is aborted, the current SPM partition is marked empty, and the task is considered for rescheduling. If an error is reported when the DMA was loading a task, the corrupted word is recopied from the redundant backup copy. If the error is reported while the DMA was unloading a task to the working copy (first unload), the corresponding SPM partition is marked empty, and the task is considered for rescheduling from the backup copy. If the error is reported while the DMA was unloading the task to the backup copy (second unload), the task is considered successfully finished and no need for rescheduling. The faulty backup copy will be corrected in the next task unload. Note that no other error is assumed to occur during the operation of the recovery as long as the task meets its constrained deadline and the error only occurs once in every two periods. Based on this, any faulty copy of a task in main memory (working or backup) will get corrected after the recovery operation is done. Anytime a faulty task is considered for rescheduling, it is considered as the highest-priority task in the system to bound its recovery time.

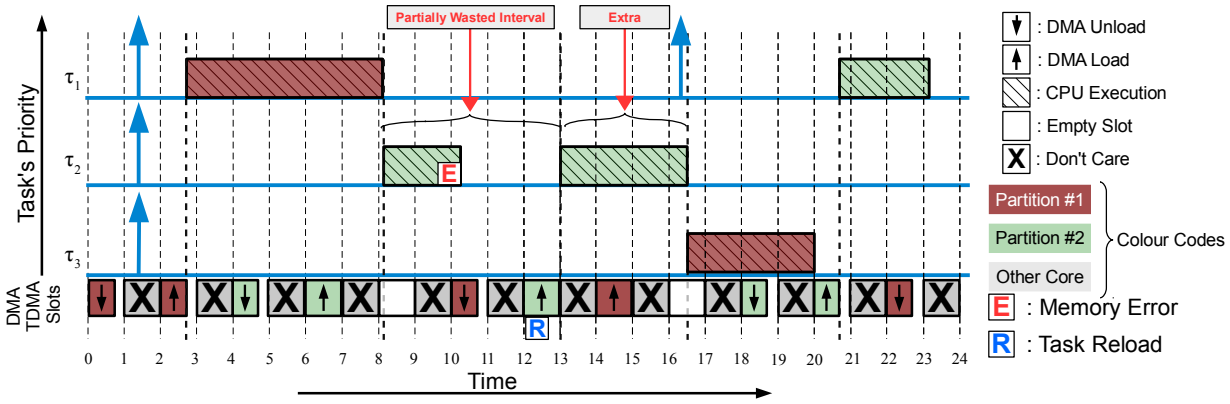


Figure 3.8: Scheduling example with the proposed error recovery mechanism

Unlike the normal system operation as depicted in Figure 3.6, in case of memory errors, Figure 3.8 shows an example of the additional operations needed to recover of an error occurs during the execution of τ_2. Basically, the execution of τ_2 is aborted and the task is reloaded to the same partition right in the next TDMA slot of the corresponding core since it becomes the highest priority task.

3.4.1 Extending Schedulability Analysis for Error Recovery

Based on the description mentioned above, in this section we derive a safe bound on the worst case response time for the task under analysis τ_i. Since we follow a similar execution model, we employ the same analysis framework introduced in Section 3.3.1. However, we need to account for the recovery overhead in the analysis. During the analysis we assume that C_i is the adjusted worst-case execution time of τ_i which includes all overheads, such as the context-switch and any needed recovery operation done on the core.

For simplicity, we discuss the case in which only one memory error can occur in two consecutive period of any task in a two cores system. However, the analysis could be easily extended to account for more frequent errors and with more than $m = 2$ cores.

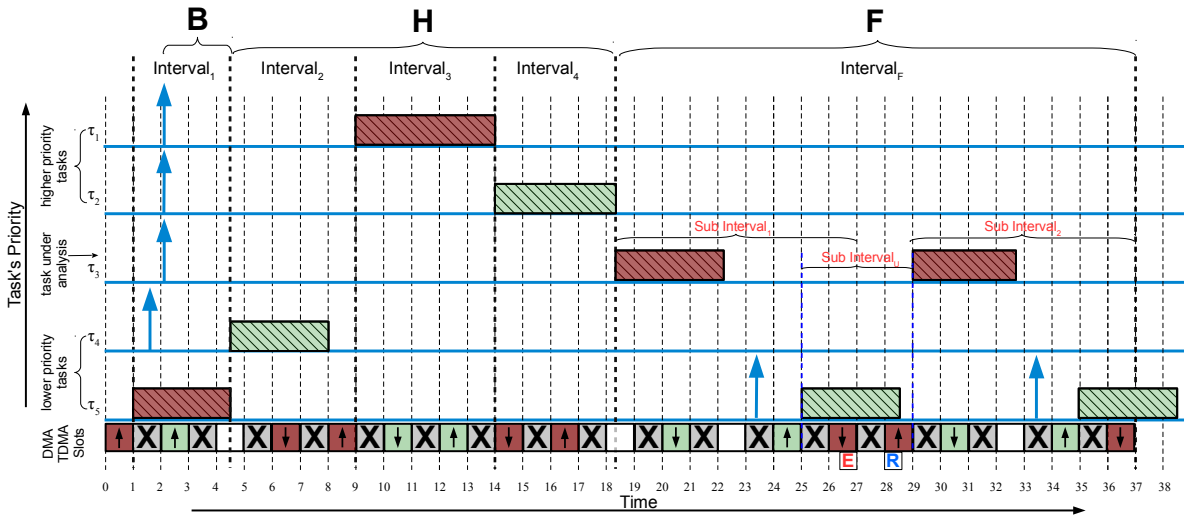


Figure 3.9: Task scheduling with illustration of error recovery mechanism

Figure 3.9 depicts an illustrative example of the worst case scheduling scenario (critical instant at time $t = 2$ and following busy interval) for an example task set where τ_3 is the task under analysis. The schedule depicts a busy period where τ_3 suffers interference from two higher-priority tasks, τ_1 and τ_2.

As in Section 3.3.1, we consider the busy period as composed by a sequence of *scheduling intervals* $Interval_2, Interval_3, Interval_4$ (each bounded by bold vertical lines in the figure), followed by a *final interval* $Interval_F$. During each scheduling interval, only one blocking or interfering task runs. During the final interval, the task under analysis runs. Similar to

Section 3.3.1, the size of the TDMA time slot is σ, which is assumed to be long enough to load or unload any task including any extra overhead due to managing the recovery mechanism.

3.4.2 Response Time Calculation

Building on the above-mentioned definitions, we can follow the same technique detailed in Section 3.3.1 to compute the worst case response time for a task under analysis τ_i (τ_3 in Figure 3.9). In Section 3.3.1, the response time of τ_i is computed by adding three components: (1) the blocking time B caused by a lower priority task that starts executing before the beginning of the busy interval; this is $Interval_1$ executing task τ_5 in the figure; (2) the interference H comprising the remaining scheduling intervals in the busy period, which are $Interval_2$, $Interval_3$ and $Interval_4$ in the figure. The number of such intervals is equal to the number of interfering higher priority jobs plus one, since an extra lower priority job that starts loading before the beginning of the busy period (τ_4 in the figure) can execute within the busy period itself; (3) the worst case length F for the final interval $Interval_F$ during which the task under analysis is executed, up to the finish time for the unload operation of τ_i. Therefore, the response time of the task under analysis is $R_i = B + H + F$; since the length H of the interfering intervals depends on R_i, this is computed using a standard iterative method. In particular, notice that the number of interfering higher priority jobs is computed based on $R_i - F$ rather than R_i: once τ_i starts executing in $Interval_F$, newly arriving higher priority jobs cannot delay its execution anymore.

As proved in Lemma 3.4, the critical instant is produced when the task under analysis τ_i and all higher priority tasks arrive immediately after a lower priority task has started loading into a partition, and the other partition was loaded with another lower priority task as late as possible (i.e., two slots before). Based on the critical instant, we then obtain $B = \max(C_l, \ \ 2 \cdot \sigma) - \sigma$, where τ_l is the lower priority task with the largest execution time. Finally, based on Lemma 3.5 in Section 3.3.1, the maximum length of the final interval is $F = \max(C_i + 5 \cdot \sigma, \ \ 7 \cdot \sigma)$.

Compared to the analysis in Section 3.3.1, here the analysis accounts for the possible memory error/recovery that might lead to a task reschedule. We show that we can use the same response time iteration as in Section 3.3.1 to calculate the response time of the task under analysis after extending H or F depending on when the memory error/recovery takes place.

3.4.3 Accounting for Error Recovery

Since we assume that no more than one error can occur for any two consecutive periods of any task, it follows that during the busy interval of the task under analysis τ_i there can be at most one task that suffers one error. A failed task is then rescheduled within bounded time.

Lemma 3.6. *A failed task that is rescheduled with highest priority will be reloaded after at most m TDMA slots.*

Proof. Since the failed task will be raised to be the highest priority task in the system and the load has priority over unload, it is guaranteed to reload the failed task in the same partition during next TDMA slot of the corresponding core, which is once every m slots. Therefore, the failed task is guaranteed to be reloaded after m TDMA slots, 2 in the case of 2 cores as shown in Figure 3.9. □

At the task level, the error might occur during the load, execute, or the unload of the task. However, to produce the worst-case workload induced by a task to the schedule, the memory error must happen as late as possible during the unload of the task.

Lemma 3.7. *A task generates the worst-case workload induced into the busy interval when the memory error occurs as late as possible, i.e. during the unload phase.*

Proof. There are three cases in which the error can occur, (1) during the load, (2) during the execution, and (3) during the unload. Case (1) does not incur a rescheduling overhead. This is because the error is recovered during the load of the task. The associated overhead is already included in the TDMA slot size by accounting for the time of copying the corrupted word from the relevant redundant memory. However, in case (2), the execution of the task is aborted (hence partially wasted) and the task has to be rescheduled to load again after m TDMA slots. Finally, in case (3) the task is fully executed and unloaded before being rescheduled and reloaded after m TDMA slots; since this case results in the most (wasted) time added to the busy interval, it is the worst case. □

Based on Lemma 3.7, we assume that a memory error always occurs as late as possible during the unload phase of a task to capture the worst case.

At the schedule level, we classify the memory error/recovery based on when it occurs with respect to the task under analysis, (1) prior to the final interval in which the task under analysis runs or (2) during the final interval.

Error Recovery Prior to The Final Interval ($Interval_F$)

Lemma 3.8. *For an error that occurs prior to F, adding an extra interfering interval to H executing the task in the system with the largest execution time other than τ_i leads to the worst case response time for the task under analysis.*

Proof. By definition, the error has to be in H or B to affect the response time of the task under analysis. Based on Lemma 3.7, the error should occur during the unload of a task; hence, the recovery mechanism forces the failed task to be rescheduled, i.e., a new scheduling interval is added to the schedule. The rescheduling of the failed task will account for the execution and the load/unload operations of the failed task. Furthermore, regardless of the tasks priority, the rescheduled (failed) task always runs as the highest priority in the system, thus this additional scheduling interval causes interference to the task under analysis.

Since we cannot make any assumption on which task might fail, lower-priority task or higher-priority task, it is safe to assume that the longest executing task in the system other than the task under analysis will be scheduled to run in the induced interval. Finally, note that even if the task that fails is the one executed in B, the rescheduled task will execute during H including its load/unload memory operations. Since Algorithm 3.1 is able to correctly upper bound the interference in H caused by any task without making any assumption regarding their order, we then simply add the induced interval to H to capture the worst-case response time for τ_i. □

Based on Lemma 3.8, let H_{rec} be the computed length of interfering intervals including one restarted task. To give a concrete example on how an error happening prior to the final interval ($Interval_F$) extends H, refer to Figure 3.10. In this example, τ_5 fails during unload; it is reloaded at time 8-9 and executes in a new third interval in place of τ_1; τ_1 then executes in the fourth interval and τ_2 in a fifth interval. Since our analysis bounds the length of the intervals in H without making any assumption on the order of the tasks, this is safe. Note that the analysis is based on H always having tasks ready to load by definition, which is not the case for $Interval_F$, thus requiring a separate analysis in Lemma 3.9.

Error Recovery in The Final Interval ($Interval_F$)

Lemma 3.9. *For an error that occurs within $Interval_F$, the maximum length of the interval with $M = 2$ is $F_{rec} = 2 \cdot \max(C_i + 5 \cdot \sigma,\ \ 7 \cdot \sigma) + \max(C_u,\ \ 4 \cdot \sigma) - 2 \cdot \sigma$, where τ_u is the task in the system with the largest execution time other than τ_i.*

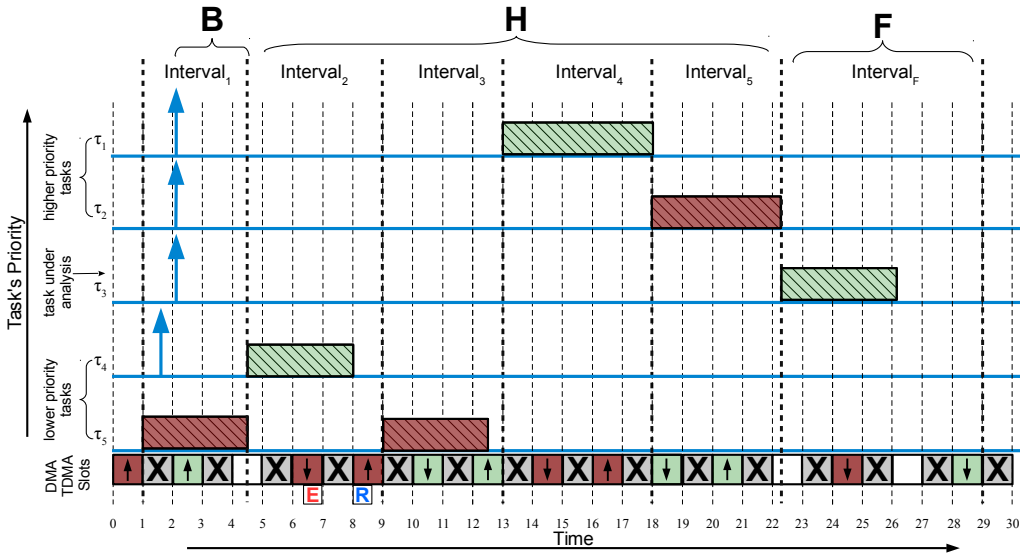


Figure 3.10: Example showing how H is extended by an induced interval due to error recovery prior to $Interval_F$

Proof. As defined earlier, $Interval_F$ starts with the execution of the task under analysis τ_i and finishes with the end of the unload phase of τ_i. Based on Lemma 3.7, in the worst case, the error occurs during the unload phase. Therefore, the error must occurs in the unload phase of τ_i, otherwise τ_i will unload successfully and the interval finishes.

Based on Lemma 3.5, in the normal case, the unload operation for τ_i completes after at most $max(C_i + 5 \cdot \sigma, \ 7 \cdot \sigma)$ time units from the beginning of the interval; in Figure 3.9, this occurs at time 27. However, in the case of memory error during the unload operation and task recovery, $Interval_F$ is extended. To simplify the computation of the worst-case length F_{rec} of $Interval_F$ accounting for recovery, we divide the interval in three sub intervals. The first execution of τ_i is contained in the first sub interval $SubInterval_1$, which finishes with the first (failed) unload of τ_i. The last sub interval $SubInterval_2$ starts with the second execution of τ_i after the reload and ends with the (successful) second unload, time 37 in Figure 3.9. In the worst case, there can be a middle interval $SubInterval_u$ in which another task τ_u is loaded into the other partition and executed.

Based on Lemma 3.6, τ_i will be reloaded after m TDMA slots (2 in our case). $SubInterval_u$ finishes and $SubInterval_2$ starts either when the reload operation is complete (case shown in the figure) or when τ_u finishes, whichever happens last. Since in the worst case $SubInterval_u$ starts after τ_u has been loaded, and we need m TDMA slots for the failed unload and

m TDMA slots for the reload of τ_i, the length of $SubInterval_u$ can be upper bounded as $\max(C_u, \quad 4 \cdot \sigma)$ when $m = 2$. The lengths of the sub intervals $SubInterval_1$ and $SubInterval_2$ are calculated the same way as in Lemma 3.5 as discussed above. However, as shown in Figure 3.9, $SubInterval_1$ and $SubInterval_U$ overlap by two TDMA slots (time 25 to 27 in the figure). This is because the length of $SubInterval_1$, as in Lemma 3.5, is calculated up to the end of the unload phase, while $SubInterval_U$ starts $m = 2$ slots before. To overcome this overlap, we subtract the overlapped time which is $2 \cdot \sigma$ when $m = 2$.

As a result, the length of $Interval_F$ is computed as:

$$\begin{aligned} F_{rec} &= SubInterval_1 + SubInterval_U \\ &\quad + SubInterval_2 - 2 \cdot \sigma \\ &= \max(C_i + 5 \cdot \sigma, \quad 7 \cdot \sigma) + \max(C_u, \quad 4 \cdot \sigma) \\ &\quad + \max(C_i + 5 \cdot \sigma, \quad 7 \cdot \sigma) - 2 \cdot \sigma, \end{aligned}$$

which is the same as the value in the hypothesis. □

In general, we do not know which case will lead to the worst response time calculation for τ_i, when the error occurs before $Interval_F$ (case 1) or within $Interval_F$ (case 2). As a result, we independently calculate both iterations:

$$R_i^1 = B + H_{rec} + F, \tag{3.1}$$

$$R_i^2 = B + H + F_{rec}, \tag{3.2}$$

and take the maximum response time among R_i^1, R_i^2. In particular, note that it is easy to see that $F_{rec} - F$ is larger than the size of the additional interval added for case 2. This is the main reason why we chose to reschedule failed tasks at the highest priority: it minimizes the worst case length of $Interval_F$ in case the task under analysis fails. On the other hand, rescheduling the task at higher priority means that we have to consider any task, rather than just higher priority tasks, for the extra interval in case 2, but as discussed this is generally not the worst case. However, note that we cannot formally avoid computing the iteration in Equation 3.1 because the interfering window for higher priority jobs is based on $R_i^1 - F = B + H_{rec}$, which is larger than $R_i^2 - F_{rec} = B + H$.

3.5 Summary

In this chapter, we have discussed our strategy for executing real-time tasks according to the 3-phase model. We assume a partitioned system where tasks are statically pinned to

cores. Each core is augmented with a dual-ported SPM divided in two partitions: this allows us to execute the current task out of one partition, while the DMA engine unloads the previous task and loads the next one in the other partition. As long as the length of DMA operations is less than the execution time of the task, the memory transfer time is completely hidden. Furthermore, since the task accesses local memory only, its execution time is not affected by contention to shared memory resources.

Isolation among cores is guaranteed through DMA scheduling. In particular, we consider two different scenarios: in the first case, the schedule happens at the hardware level, so that each DMA operations initiated by each core are guaranteed to receive a fixed portion of memory bandwidth. In the second case, we explicitly enforce a TDMA arbitration between DMA operations of different cores by programming the DMA engine in software. We formally described a set of rules to co-schedule core execution and DMA transfers in both cases, and developed corresponding sufficient schedulability analyses for sporadic task sets based on fixed-priorities. To avoid wasting time loading tasks that are only partially executed, we consider non-preemptive task scheduling. However, we also extend the analysis to a limited-preemption model where each task is divided into a set of sequential intervals, and preemption is possible between intervals. Finally, we demonstrated how our strategy can improve fault-tolerance by restarting faulty tasks from the copy stored in main memory.

In the next chapter, we will show how the discussed techniques can be practically implemented based on the case study of two different embedded platforms. We also provide an evaluation of the presented scheduling schemes against previous work based on measured benchmarks running on the platforms.

Chapter 4

System Implementation

In Chapter 3, we introduced a set of scheduling mechanisms and associated schedulability analyses for 3-phase tasks in a partitioned real-time multi-core system. To prove the applicability of the discussed techniques, in this chapter we discuss the system realization of the proposed execution and scheduling methodologies on two different embedded platforms. In particular, in Section 4.1, we describe the system realization on an Xilinx FPGA platform using soft-core processors; since we have complete control over the hardware, this implementation relies on the variable-size DMA operation scheme detailed in Section 3.2. Then, in Section 4.2, we describe the system realization on a commercial embedded platform, the Freescale MPC5777M micro-controller unit (MCU)[12]. Since this platform does not support per-core DMA engines nor a predictable memory arbiter, we rely on the fixed-size DMA operation scheme based on a TDMA schedule of the DMA engine, as discussed in Section 3.3. The COTS platform also satisfies the requirement for the implementation of the fault-tolerant scheme described in Section 3.4. We evaluate the proposed systems with a set of both synthetic and embedded automotive benchmarks from the EEMBC suite [134].

4.1 Implementation on an FPGA Platform

Since we consider a partitioned task system where each core gets a specific set of tasks, in this implementation, we discuss the details of one core system. We assume that the system can be scaled to multiple cores by duplicating the architecture. Note that, when more than one core are realized, the memory bandwidth must be controlled and divided between the

cores in a predicable manner [96]. Each core runs a full software stack (including the OS), and is fully isolated from other cores.

4.1.1 Hardware Architecture

To recap our assumptions about the hardware mode, we consider a system where each processor is augmented with a dual-ported local memory (SPM) and a DMA component. We further assume that the CPU and the DMA can access different portions of the local memory concurrently without causing mutual interference. We utilized the MicroBlaze [10] soft-core processor in our implementation. We disabled the use of cache and replaced that with on-chip block RAM to implement the SPMs. Since MicroBlaze has separate instruction and data interfaces, we implemented two local SPMs, I-SPM and D-SPM. The size of the SPM is limited by the available on-chip RAM. In our implementation, each SPM was 64-KB. This was to match the maximum cache size that MicroBlaze can be configured with using its configuration tool. Later in Section 4.1.4, we compare the hardware resource utilization for our proposed SPM hardware architecture against similar implementation using caches instead of SPMs.

As shown in Figure 4.1, The CPU accesses memory through the Real-time Scratchpad Memory Units (RSMUs). The RSMUs are connected to the CPU using the Local Memory Bus that provides single-cycle access latency. The RSMUs direct the memory access either to the local SPM or to the system bus based on the virtual address provided by the CPU. In addition, the RSMU provides internal memory-mapped control registers to allow for software management. Specifically, the RSMU provides address translation for the loaded tasks in the SPM. According to our design methodology that relies on the knowledge from the scheduling level, we were able to simplify the address translation mechanism. The RSMU translate addresses in constant time without access any lookup tables as in conventional MMU. For this reason, unlike other SPM management units [174, 175, 176], the RSMU hardware is scalable without any dependence on the number of translated tasks or memory blocks. The access latency to the local SPM through the RSMU is one cycle.

4.1.2 Address Translation

As depicted in Figure 4.2-(a), each RSMU exposes a range of memory addresses. If the CPU generate and address outside the RSMU address range, the RSMU forwards the access to the system bus. This allows the CPU to access main memory and other memory-mapped devices, such as I/O peripherals. On the other hand, if the CPU-generated address

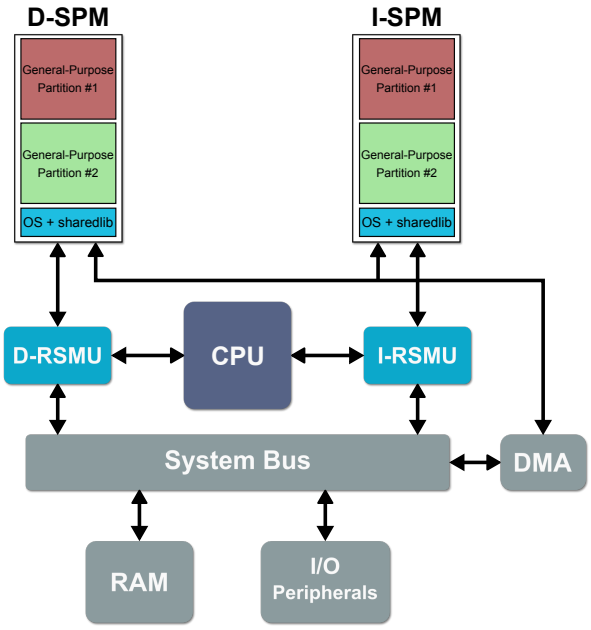


Figure 4.1: System-Level Block Diagram

corresponds to the RSMU address range, it falls into two categories, 1) the RSMU Physical space and 2) the RSMU virtual space. The physical space corresponds to the locally connected SPM, as shown in Figure 4.1. In other words, the RSMU's physical address range is directly mapped to the local SPM. For example the address range from `0x05000000` to `0x05010000` (64-KB) is directly mapped to the SPM address range from `0x00000` to `0x10000` (64-KB). This address range is suitable for permanent residing software that get loaded at boot time, such as important OS functions and shared libraries. Whereas, accessing the RSMU virtual address space needs translation to the physical address space. This address space is suitable for real-time task that needs to be loaded into the SPM at runtime. Basically, real-time tasks are compiled and then linked to a non-overlapping address ranges withing the RSMU virtual address space, as show in Figure 4.2-(b). At boot time, the tasks are loaded into main memory based on their load addresses as defined in the linkage stage, Figure 4.2-(c). Finally, when a task is loaded into the SPM, at the context switch, the RSMU is configured to perform address translation for the executing task. Based on how the RSMU is configured at runtime, the RSMU performs address translation according to the following simple function:

$$\texttt{SPM}_{\texttt{physical}} = (\texttt{CPU}_{\texttt{virtual}} > \texttt{TER})? \quad \texttt{CPU}_{\texttt{virtual}} - \texttt{ATOR} \ : \ \texttt{CPU}_{\texttt{virtual}}$$

Where `TER` is the translation-edge register, *e.g.*, `0x05010000` in our case, and the `ATOR` is

the active task offset register. For example, if the value stored in the `ATOR` is `0x06020000` and the value stored in `TER` is `0x06010000`, then the CPU-generated address, such as `0x060200AA`, will be translated into `0x000AA`. The value of the `ATOR` register is determined by the scheduler based on where in the SPM the task is loaded. Note that, given the fact that only one task is executing out of the SPMs at any given time, the RSMU design is significantly simplified to translate only one task at a time.

In this implementation, the RSMU address space covers 1 MB range, which is sufficient to link all critical tasks in our evaluation to the virtual space of the RSMU. Note that, the RSMU only needs 20-bits comparators to perform the translation form its virtual address space to the physical address space, which leads to a faster hardware.

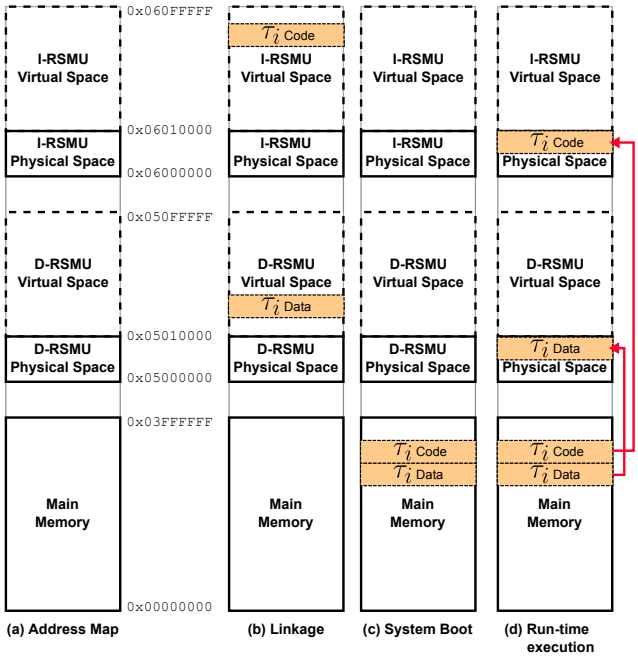


Figure 4.2: RSMU Memory Translation

4.1.3 Software Implementation

To support the proposed scheduling and execution models, we extended FreeRTOS to manage the SPM space, DMA transfers, and RSMUs operations. All software components including the OS are compiled together to produce a single system binary image. The SPM is partitioned in software to three partitions. As shown in Figure 4.1, the SPM has

two general purpose partitions mainly to load tasks into. The third partition is to load OS-level critical functions, such as the task scheduler and a small system heap.

To simplify task loading and unloading to and from the SPM, we developed a linker script to combine all task's related code sections in one location (adjacent memory blocks). Similarly, we combine all task's related data sections in one location. This two memory blocks makes up what we call the task image. The space required for the execution stack of the task is also included in the image, hence get loaded and unloaded with the task image. Note that, the software system is compiler independent, and only needs some inputs via the linker script to setup tasks images and their load addresses in main memory. At boot time, the boot loader loads each task to its load address in main memory. Note that, since tasks virtual addresses are linked to the RSMU virtual space, they must be loaded into the SPM before they can run.

Since we mainly consider Fixed-priority non-preemptive scheduling, the scheduler is invoked either by the DMA finish interrupt, or when a task yield the execution (finish). To support task preemption and limited preemption schemes, the scheduler can be invoked by other devices interrupt, such as I/O peripherals and the system timer. At every context switch, the scheduler can make tow scheduling decisions. First, the scheduler programs the DMA to load the top priority pending task into one SPM partition. If the partition is empty, *e.g.*, the previously loaded task finished but not unloaded yet, the scheduler program the DMA to unload the previous task before loading the next one. Second, the scheduler configure the RSMU translation registers for the already loaded task on the other partition if any. Lastly, the scheduler context switches to the scheduled task if any. Not that, the scheduler does not block waiting for the DMA to finish as the DMA operations will run in parallel and will be overlapped with the CPU execution of the newly scheduled task. Therefore, on the DMA setup time is incurs during the context switch routine. The overhead of the OS context switch routine is accounted for in the schedulability analysis.

The proposed system is transparent to application, just like in a cache, as the management of the hardware resources (SPM space, DMA, and RSMU) and task execution mechanism is all done at the OS level. This ensures that porting applications to our system is straightforward; software does not need code annotation, special machine-instructions or special compilers to take advantage of the described management technique. The main trade-off is that our scheme requires the task image to fully contained in one SPM partition. However, as discussed earlier in Section 3.1, known methodologies exist [128, 145, 94] to split a large application into smaller segments that are individually compliant with the imposed constraint.

4.1.4 Evaluation

The evaluation is divided into three subsections, hardware architecture, performance, and system schedulability. In addition, in this evaluation, we adopt the scheduling scheme discussed in Section 3.2, in which the DMA time is dynamic base on the tasks sizes.

Hardware Architecture

To evaluate the hardware, we compare the proposed hardware architecture, shown in Figure 4.1, to the baseline hardware architecture, shown in Figure 4.3. The realization of both systems has the same amount of software-usable on-chip memory, 128KB: 64KB for instructions and 64KB for data. The hardware prototyping is done on Xilinx ZC706 FPGA evaluation board [6]. The CPU is a Microblaze 32-bit soft-core processor [10]. In addition, the platform has an external 1-GB DDR3-SDRAM, which serves as main memory. The FPGA chip on the ZC706 board has much more on-chip memory than the 128KB. However, the configurable caches (I-cache and D-cache) on the Microblaze are limited to maximum of 64KB each. Therefore, for the sake of fair comparison we have limited the SPMs to the same sizes.

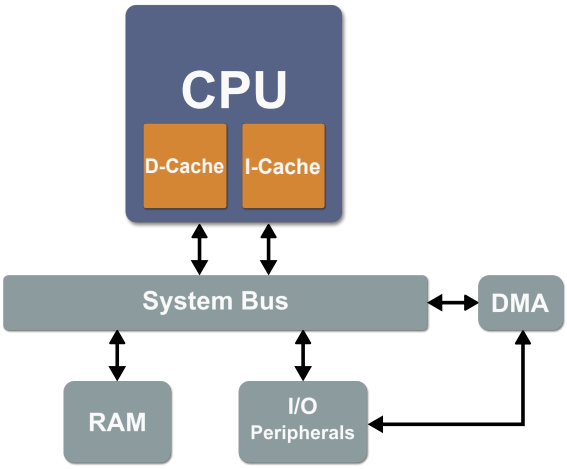


Figure 4.3: The baseline hardware architecture with cache memory

Tables 4.1 and 4.2 show that the architecture with SPM only is more resource saving and can run on higher operating frequency than the architecture with caches only. The saving in the hardware resources (area) can be re-utilized. For example, it can be used to

have more on-chip memory, which can lead to more performance of the system or can be used to implement any extra functionality. To depict the effect of design complexity on the operating frequency as pure as possible, we conducted the hardware synthesis on minimal design that has a Microblaze CPU and the on-chip memory (caches or SPMs). This way, we can closely capture the complexity of the implemented, direct-map-8-words-per-line, cache memory of the Microblaze without interference of the other components that can reduce the frequency either for their complex designs or due to suboptimal place and rout.

Table 4.1: Maximum operating frequency comparison

Hardware Resource	**Cache**	**SPM**	**Gain Ratio**
maximum operating frequency	200.3 MHz	243.7 MHz	1.22 X

Table 4.2: Hardware resource comparison:
in this specific implementation, each block RAM is 36 Kilobits (Kb).

Hardware Resource	**Cache**	**SPM**	**Saving**	**Ratio**
flip-flops (FF)	5478	3071	2407	1.78
lookup tables (LUT)	5729	3436	2293	1.67
memory LUT (MLUT)	410	169	241	2.43
block RAM (BRAM)	37	33	4	1.12

The performance of the DMA is entirely predictable as the DMA does not interfere with the CPU to access the main memory. The DMA timing is evaluated by, first, measuring the amount of time required to transfer some fixed amount of data from main memory to scratchpad and vice-versa. After that, the estimated timing for a transfer of any size is derived based on linear regression. The estimated timing for a transfer of any size is:

$$DMA(b) = 215 + 0.303 * b,$$

where $DMA(b)$ is in cycles and b is in bytes.

The above equation does not include the DMA setup overhead required to prepare the DMA transfer. The DMA setup overhead along with other OS overheads are shown in Table 4.3.

Table 4.3: Software Overheads

	Cycles
Context Switch C_s	450
DMA Set-up Overhead DMA_{setup}	53
Timer Overhead t_s	226

Software Performance

About 1050 lines of code were added to the FreeRTOS kernel to support our scratchpad management scheme. Overall, the compiled size of the OS is small ($\simeq$ 18.5KB) and, depending on the required features and libraries needed by the real-time tasks scheduling and resource management, only tiny part of the OS is required to be stored in the SPM, starting from 4.5 KB for code and 324 B for data.

A set of both synthetic and real benchmarks is executed on the platform in order to obtain data towards performance and schedulability analysis. The a set of the well-known automotive EEMBC benchmark suite [134] is selected to represent real applications used in the automotive real-time domain. The benchmarks are configured to consume all input data based on the number of iterations. Thus, the execution time is related to the size of the input data. Benchmarks larger than the size of the local SPM partition are broken up into smaller chunks, e.g., we have configured each benchmark so that it does not consume more than half of the SPM size (32KB). In addition, benchmarks are configured not to run for very long time to resemble real-life embedded real-time applications; we limited the benchmarks to run up to $1ms$ maximum. Table 4.4 reports, for each benchmark, the size of code and data, the execution time out of the local memory (cold cache, hot cache and SPM), the execution time ratio between the SPM case and the cold cache case, and the time taken to load code and data, using DMA, into the SPM and the time taken to unload only data from the SPM. All times are reported in cycles (cyc) and the sizes are reported in bytes (B). Note that, in this evaluation we considered variable DMA load and unload times. Therefore, we reported these times for each application as shown in the table.

We evaluate the performance of the reported application, Table 4.4, on both systems. The evaluation is based on simulations and uses measures extracted from the running hardware, as reported in Table 4.4. In the case of the cache system, building an accurate analytical model of the complex cache is difficult. Therefore, a very abstract and optimistic cache model is considered. Although the hardware reported in Table 4.1 and Table 4.2 are for simple direct-mapped cache, a fully-associative cache is modeled with LRU replacement policy. The degraded search performance usually associated with fully-associative cache

Table 4.4: Benchmarks Results

Benchmark	Code size(B)	Data size(B)	Cold(cyc)	Hot(cyc)	SPM(cyc)	Difference(cyc)	Ratio(%)	load(cyc)	unload(cyc)
a2time	3640	5340	100811	97672	97276	3535	3.51	3152	1834
aifftr	5040	4956	108885	107003	106405	2480	2.28	3460	1717
aifirf	3204	3024	110192	109434	109180	1012	0.92	2318	1132
aiifft	4592	4940	96596	94102	94065	2531	2.62	3319	1712
basefp	2636	2320	102822	101275	101099	1723	1.68	1932	918
bitmnp	19716	2968	95917	92219	91888	4029	4.20	7408	1219
cacheb	4236	6524	99100	97719	97644	1456	1.47	3691	2192
canrdr	3080	9892	113842	111692	104833	9009	7.91	4362	3213
idctrn	10908	3402	123548	120881	106959	16589	13.43	4767	1246
iirflt	6648	2144	106889	103545	97278	9611	8.99	3095	865
pntrch	3256	6356	87147	86067	80781	6366	7.30	3343	2141
puwmod	3896	1936	122314	120160	103177	19137	15.65	2198	802
rspeed	1956	5732	99406	98678	94326	5080	5.11	2760	1952
tblook	4228	1672	102847	97984	97645	5202	5.06	2251	754
ttsprk	10764	5704	101760	99780	97272	4488	4.41	5421	1944
synthetic	128	63496	73290	7421	7409	65881	89.87	19709	19455

is not considered here, and the same access latency of the SPM is assumed, which is one clock cycle. Despite the access latency of the cache, the complicated circuit of the fully-associative cache can negatively affect the operating frequency of the system as it is part of the critical data path, and this is not considered in the model as well. This very optimistic model of the cache is aimed to favor the cache system so that the comparison results will be more authentic toward the SPM system. It is highly expected that in real hardware the cache system will do worse than in this evaluation.

The total execution time of an arbitrary fixed schedule on both systems is captured. This will depict the difference in the software execution performance on both systems regardless the fact that the SPM system provides predictable execution. The experiment was to execute the applications in Table 4.4 sequentially (one after another) and report the total execution time for each application. Then the experiment is repeated few millions times but with different application order in each time. The order is picked randomly with no repetition. Finally, to favor the cache system, we reported the average execution time of each application instead of the worst case execution time. In the case of the SPM system, the execution time of an application is always fixed (predictable). Figure 4.4 compares the two systems in terms of applications execution time. Note that, our execution model exploits the overlap between CPU execution and DMA loading to hide access latency to main memory.

From Table 4.1, we notice that the SPM system can operate at higher frequency. Therefore, Figure 4.4 compares three systems: the cache system, the SPM system running at the same frequency as the cache system, and the SPM2 system running at 22% higher fre-

quency. This figure depicts that the SPM system, with the all associated overhead (Table 4.3), can be a strong candidate to replace the traditional hardware-managed cache. Figure 4.5 shows the applications speedup if they run on the SPM system instead of the cache system. Among the reported benchmarks, idctrn and puwmod run about 1.5 faster on the SPM system than the average case on the cache system, which reflects their memory intensive behaviors.

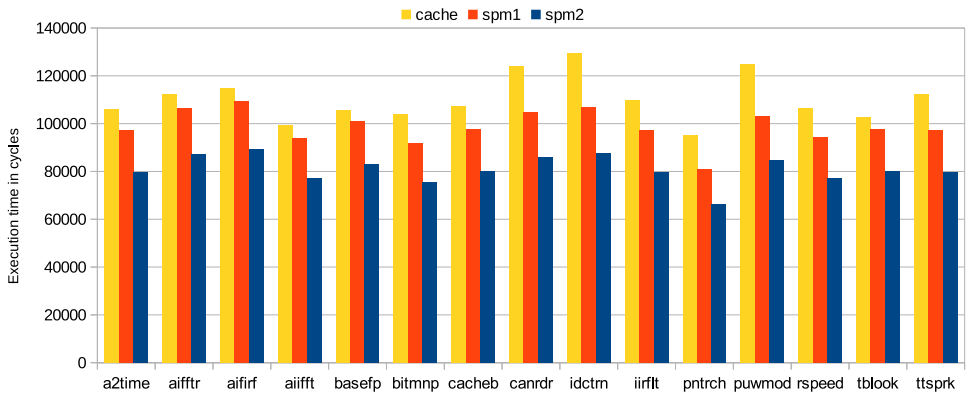


Figure 4.4: Execution time comparison between Cache and SPM:
SPM1 runs at the same frequency as cache, SPM2 runs at 22% higher frequency

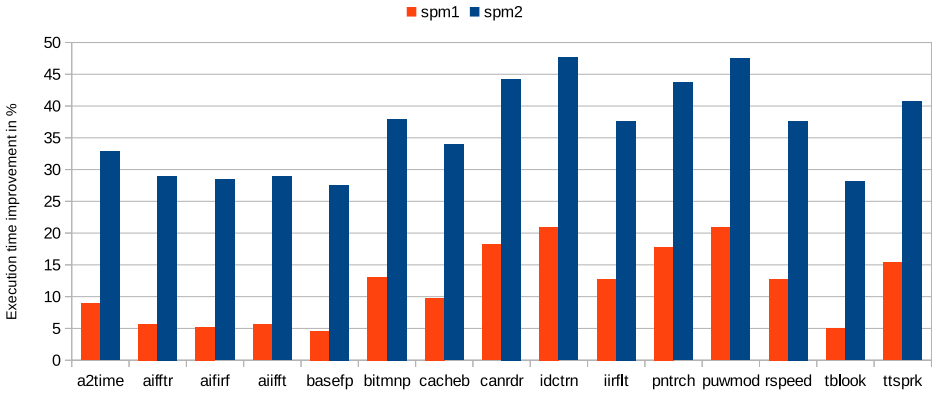


Figure 4.5: Application execution speedup:
SPM1 runs at the same frequency as cache, SPM2 runs at 22% higher frequency

Schedulability

The introduced fixed-priority scheduling scheme, as in Section 3.2, is evaluated against other scheduling schemes, either based on caches such as PREM [178] or based on SPM such as Carousel [176]. As discussed in Section 2.1, PREM uses the CPU to prefetch the task into the local cache, while Carousel stalls the CPU until the DMA loading is finished. While both approaches effective wasting the CPU time while loading a task, we pipeline the CPU execution and the DMA transfer to hide memory access latency.

Applications in Table 4.4 are used to generate sets of random tasks. Given a system utilization, each application is randomly selected and assigned a random period in the range between 5 ms to 100 ms. The task's utilization is then computed based on the application's execution time out of the SPM and the selected period. At every iteration a new task is randomly generated. The generation stops when the sum of the individual tasks' utilizations reaches the required system utilization. After that, the overhead is added, such as context-switch and DMA setup.

In the case of multi-intervals tasks, similarly to the case of single-interval, each application is randomly selected but assigned a random period in the range between 15 ms to 300 ms. Then, each generated task is randomly assigned a number of intervals between one and five. Task's intervals are equally sized; the interval size is the size reported in Table 4.4. We increased the random periods proportionally to the average increase in the size of a task, which is $3X$, to maintain the same average number of tasks and per-task utilization across all cases. Carousel and PREM schedulability is verified by applying response-time

analysis as described in [176] and [178] respectively; the analysis incorporates the overhead of context-switch as well as the blocking time due to non-preemptive DMA operations.

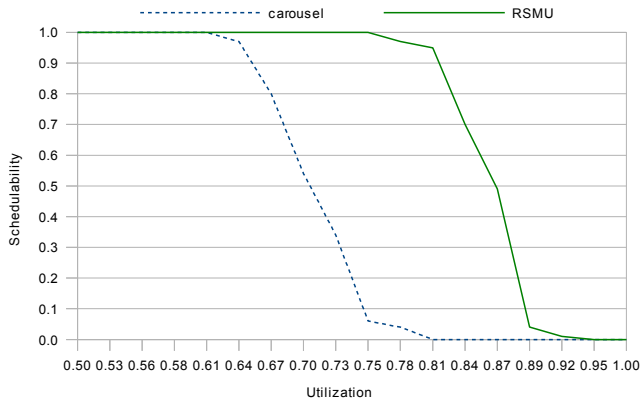


Figure 4.6: Single-core schedulability comparison between carousel and our approach

Figure 4.6 shows the single-core case. The comparison, in this case, is only valid between Carousel and our approach (RSMU) as PREM targets multi-core systems. The figure shows the results in terms of proportion of schedulable task sets for Carousel and our proposed approach. Each point in the graph, as for all other graphs presented in the following figures, represents 200 task sets. As shown in the figure, our approach is able to schedule a significantly higher number of task sets compared to Carousel. We believe that such result shows that hiding the latency of memory accesses can have a very significant impact on schedulability, even for systems such as the one in Table 4.4, where the time required to load data from main memory can be relatively small compared to execution time. The price we pay for such improvement is additional scratchpad memory: under Carousel, only the currently executing task must be loaded in SPM. In our approach, an additional partition must be reserved in SPM to pre-load the next scheduled task. Figure 4.7 compares Carousel and our system for the multi-interval case. The graph shows similar results to the single-interval case where our approach is still able to schedule more task sets compared to Carousel as expected.

For the partitioned multi-core systems, we simulate the number of cores by slowing the memory by a factor of $4X$ for a four-cores system and by $8X$ for a eight-cores system. This is applied to both Carousel and our approach. The following four figures compares the three systems, Carousel, PREM and our system (RSMU). Figures 5.10 and 4.9 compare the three systems, in the case of four-cores, for both single-interval and multi-interval systems

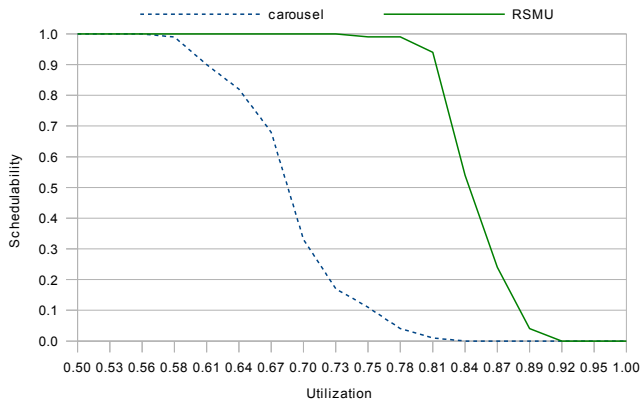


Figure 4.7: Multi-interval single-core schedulability comparison between Carousel and our approach

respectively. Similarly, Figures 5.11 and 4.11 compare the three systems, in the case of eight-cores, for both single-interval and multi-interval systems respectively. The obvious observation is that as the number of cores increases the schedulability of the three systems decreases due to the longer time needed to load tasks into the local memory. However, RSMU is negligibly affected by the increase in number of cores because RSMU successfully hides the latency of the loading process of a task by overlapping it with the execution of another task. In the single-core case the tasks' execution times are dominating over the DMA times due to the fact that the selected application reported in Table 4.4 have small load and unload times compared to their execution times. Therefore, the schedulability is not really affected by the memory performance. However, in the multi-core cases the memory performance is degraded and the load and unload times take longer to finish.

In terms of single-interval versus multi-interval comparison, Carousel and PREM show similar results in the multi-interval case to the single-interval case. On the other hand, RSMU schedulability is slightly affected in the multi-interval case compared to the single-interval case. This is mainly because of the increased number of intervals where a long lower priority task can interfere with the task under analysis. However, RSMU is still able to schedule significantly higher number of task sets compared to the Carousel and PREM. Finally, in Figure 4.12 we show the effect of memory performance on schedulability of each system. The figure shows that RSMU is more resistant against memory performance degradation than the other two systems. The graph is plotted at 70% utilization in order to show the three systems in the same graph from 100% to 0% schedulability. Based on this graph, the experimented tasks set with 70% utilization can be scheduled in our system with up to 20 cores sharing the bandwidth of the main memory, seven cores in PREM case,

and only two cores in Carousel case.

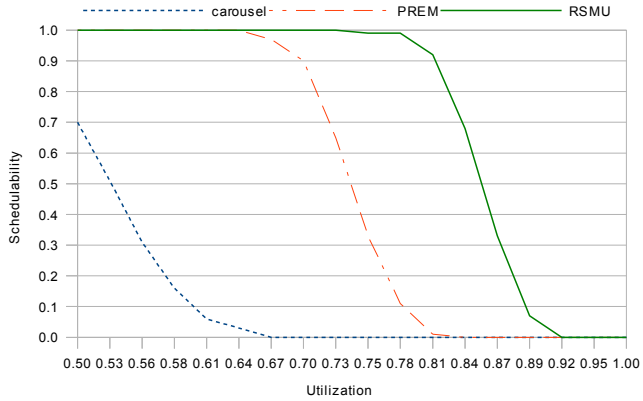


Figure 4.8: Single-interval 4-cores schedulability comparison between the three systems

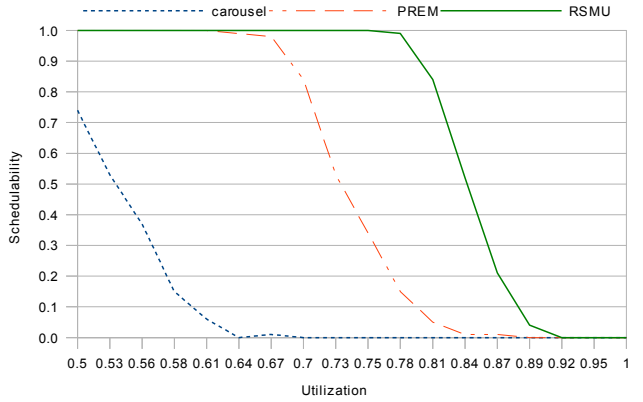


Figure 4.9: Multi-interval 4-cores schedulability comparison between the three systems

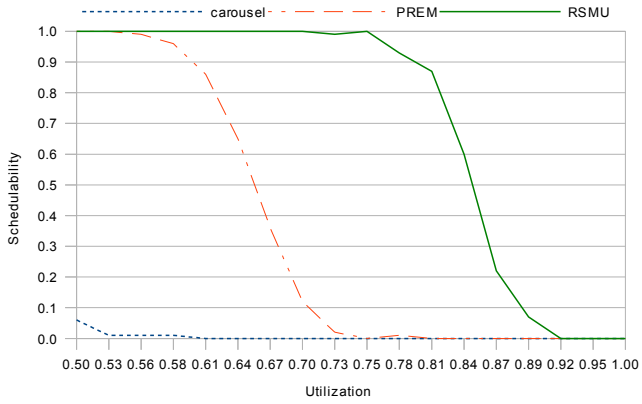


Figure 4.10: Single-interval 8-cores schedulability comparison between the three systems

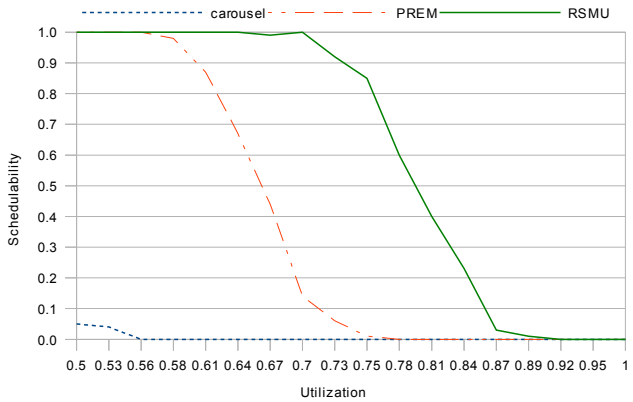


Figure 4.11: Multi-interval 8-cores schedulability comparison between the three systems

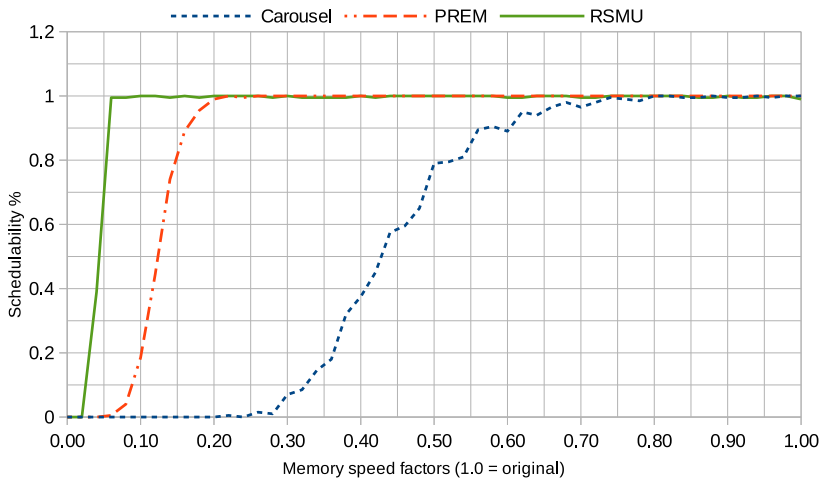


Figure 4.12: Each system's tolerance to the degradation in memory speed @ 70% utilization with multi-interval tasks

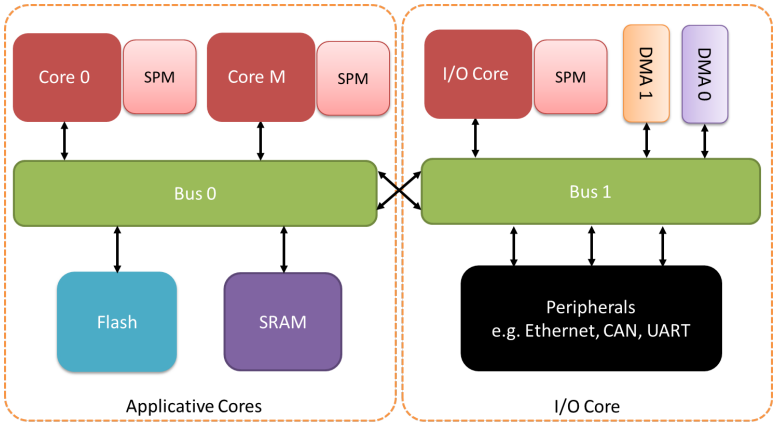


Figure 4.13: MPC5777M Block Diagram.

4.2 Implementation on a COTS Platform

In this section, we present a full system implementation of the proposed predictable execution and scheduling methodologies on a COTS embedded multi-core platform, the Freescale MPC5777M MCU. OS support was implemented using Evidence Erika Enterprise[1]. Erika Enterprise is an open-source RTOS that is compliant with the AUTOSAR[2] (Automotive Open System Architecture) standard. AUTOSAR is an open standard for automotive architectures providing a basic infrastructure for vehicular software. Erika Enterprise features a small memory footprint, supports multi-core platforms and implements common scheduling policies for periodic tasks. Our implementation enhances predictability and simplifies the OS design by exploiting hardware features that vendors are now including in some modern families of multi-core platforms designed for the embedded market, such as: separate I/O and memory buses, the presence of dual-port memories with DMA support, and core specialization. We discuss the platform features in Section 4.2.1, followed by the OS design in Section 4.2.2 and the evaluation in Section 4.2.3.

4.2.1 Platform Description

A brief summary of the architectural features of the MPC5777M MCU is provided in Table 4.5, while a block diagram is reported in Figure 4.13. The SoC platform includes

[1] http://erika.tuxfamily.org/drupal/
[2] http://www.autosar.org/

Table 4.5: Characteristics of Freescale MPC5777M SoC

Chip Name	MPC5777M (Matterhorn)
Manufacturer	Freescale
Architecture	Power-PC, 32-bit
CPU Unit	2x E200-Z710 + 1x E200-Z709 + 1x E200-Z425 (I/O)
CPU Frequency	Application Cores (300 Mhz) I/O Core (200 Mhz)
Processing Unit	CPUs, DMA, Interrupt Controller, NIC
Operational Modes	Parallel + Lockstep (on one applicative core)
ECC Protection	Cache, RAM, Flash Storage
Cache Hierarchy	L1 (Private Instructions + Data) + Local Memory
Local Memory (SPMs)	Instructions (16 KB) + Data (64 KB)
L1 Cache Size	Instructions (16 KB) + Data (4 KB)
SPM Size	80 KB
SRAM Size	404 KB
Flash Size	8 MB
Main Peripherals	Ethernet, FlexRay, CAN, I2C, SIUL
MEMU	MEMU For SRAM, Peripheral RAM and Flash

two application cores, which we use to execute 3-phase real-time tasks, and an I/O core, which we use to run the OS and all device drivers. Each core has a dedicated scratchpad memory, whose size (80 KB) is smaller, but comparable to SRAM main memory (404 KB).

The platform includes a separate I/O bus, which allows the designer to route I/O traffic without directly interfering with CPU-originated memory requests. The idea of co-scheduling CPU activity and I/O traffic is not new and specific solutions have been proposed in [41, 129]. However, traffic transmitted over the dedicated I/O bus needs to be handled, pre-processed and scheduled before reaching the application cores; the I/O core performs such operations. Just like the application cores, the I/O core features a scratchpad memory that is used to buffer I/O data before they are delivered to applications.

Typically, devices that support high-bandwidth operations are DMA-capable. Instead, slower devices expose memory-mapped input/output buffers that can be read/written using generic platform DMA engines. Without loss of generality, we assume I/O data transfers from/to the I/O core are performed by DMA engines and that data from I/O devices can directly be transferred into the I/O core's scratchpad memory. In other words, I/O devices are not allowed to initiate asynchronous transfers directly towards main memory.

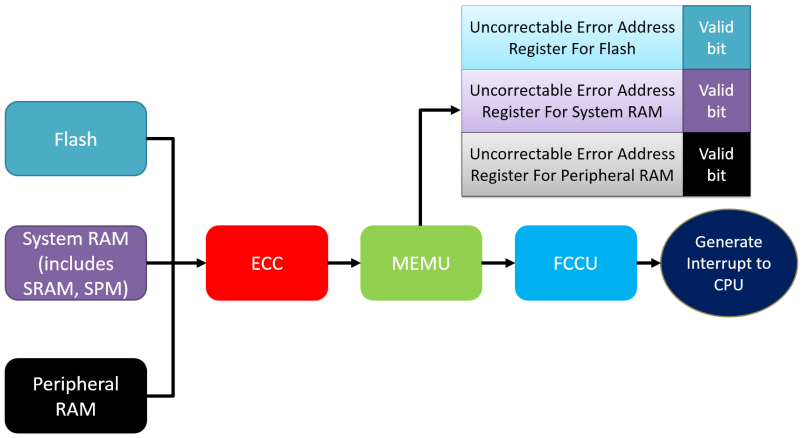


Figure 4.14: Block diagram of error handling circuitry.

This design choice allows us to perform co-scheduling of CPU and I/O activities to achieve higher system predictability.

Note that no MMU is available on this platform. Hence, there is no support for virtual memory. Instead, we use compiler techniques to generate position independent code that can be loaded in either partition of an application scratchpad, as we discuss in Section 4.2.2. In order to test and verify the safety features of the SoC, the chip also implements fake error injection mechanisms that are helpful to verify the reaction to various faults. We use these fault injection mechanisms to evaluate our system.

In the considered MPC5777M platform, there is a memory error management unit (MEMU) that is responsible for collecting and detecting the faults in different memory subsystems such as SRAM, SPM and Flash. In particular, the platform implements Hsaio codes that provide single bit error correction and double bit error detection (SEC-DED). Hsiao Code [73] for correction and detection is popular in modern embedded platforms. The MEMU implements separate tables for reporting correctable and uncorrectable errors. There are separate tables for each kind of memories. These tables contain the address of the fault that caused error, moreover, there is a register inside the MEMU that tells if the fault that occurred is a correctable or uncorrectable fault. On MPC5777M, there is no way for the MEMU to send an interrupt to the CPU in case of a fault. There is a separate FCCU module present on the chip that collects all the errors that are forwarded to it from the MEMU. The FCCU module can be preprogrammed to take certain actions based on a particular error. Moreover, it is also responsible of generating interrupt to the processor to notify it in case any kind of errors that are being reported to it from the

MEMU. Figure 4.14 shows how different modules are connected to each other. In order to detect the faults in the SPM, we registered a FCCU interrupt with application cores. This interrupt gets generated and sent to all cores when an error is reported by the MEMU to the FCCU in one of the memory subsystem.

As discussed in Section 3.4, the proposed recovery mechanism are effective as long as no more than one bit error occurs in any of the memory subsystems (SRAM, SPM, Flash) every two periods of any task. It is estimated that the FIT of SRAM memories is about 0.001, i.e. on average one bit upset is observed every 10^{11} hours of operation [151]. Flash memory, on the other hand, shows low SER susceptibility [146]. Since the period of a real-time task is typically tens or hundreds of milliseconds long, we deem this assumption to be satisfied in the vast majority of embedded systems.

Error correction techniques to recover from faults that affect OS memory require special attention and are not within the scope of this work. A promising approach in this context is system check-pointing, i.e. periodically saving the system state of OS and copying it into a more reliable piece of memory. Further research is required to seamlessly integrate check-points into our SPM management scheme.

4.2.2 OS Design

We performed a porting of Erika Enterprise on the MPC5777M MCU, adding support for UART communication interface, interrupt controller, caches, memory protection unit (MPU), data engines (DMA), and Ethernet controller. The following subsections discuss the design of individual OS components, including task configuration, task and DMA scheduling, I/O management, and error recovery.

Task Configuration

In order to implement our SPM-centric OS, we have augmented Erika Enterprise to support position-independent (relocatable) tasks. We rely on the compiler[3] support for `far-data` and `far-code` addressing modes. In this way, tasks are compiled to perform program-counter-relative jumps and indirect data addressing with respect to an OS-managed base register. We have extended the default task loader to exploit DMAs for transferring task images from SRAM to local memories and vice-versa.

[3]Applications and OS are compiled using the WindRiver Diab Compiler version 5.9.4 - http://www.windriver.com/products/development-tools/

In Erika Enterprise, tasks are compiled and linked directly inside the image of the OS. For each task in the system, Erika-specific meta-data need to be defined. Additionally, meta-data that extend the task descriptors for SPM-centric operations are required. Manually configuring these parameters is tedious and error-prone; hence, we developed an OS configurator. The tool uses high-level task definitions and generates the final configuration for our SPM-centric OS. Specifically, each core is associated with a set of configuration files that describe: number of tasks, their priority, task entry points, initial status and so on. When a task is added, these files need to be configured accordingly.

First, the body of all the tasks is placed in an ad-hoc file. Similarly, task-specific data that need to be preserved across activations are defined in different files and surrounded with appropriate compiler-specific `PRAGMA`. This is fundamental to ensure that: (A) specific linker section is used to store task code and data images; and (B) position-independent data and instructions are generated. A separate file also defines the relocatable task table, which stores the status of each relocatable task. This structure includes: (A) position in SRAM of the task code and data images; (B) position of the task's I/O data buffers; (C) current status of the task (e.g. loaded, completed, unloaded); (D) SPM partition of last relocation.

Task and DMA scheduling

The central idea of the proposed OS design is resource specialization. As previously mentioned, a specialized I/O core and I/O bus are used to handle peripheral traffic. Similarly, a specific role is assigned to different memory resources in the system. Specifically, three types of memory resources exist in our system, as depicted in Figure 4.13. First, flash memories are used to persistently store application/OS code, read-only data, as well as initialization values of read-write portions of main memory. Next, the SRAM (main) memory contains writable application and system data that represent the time-variant state of the system. Finally, scratchpad memories temporarily store a copy of code and data images for those tasks that are currently being scheduled for execution.

In our solution, applications are never executed directly from main memory, thus we adopt the following strategy: (1) task images are permanently stored in flash and loaded into main memory at system boot; (2) a dedicated DMA engine is used to move task images to/from SPM upon task activation; (3) a secondary DMA engine is used to perform I/O data transfers between devices and I/O core; (4) tasks always execute from SPM; (5) only task-relevant I/O data are transferred upon task load from the I/O subsystem. In the implemented system, a DMA engine is used to position the image of a relocatable task

inside a SPM for execution. We refer to this DMA engine as *application DMA*. Similarly, we refer to the platform DMA used for I/O transfers as *peripheral DMA*.

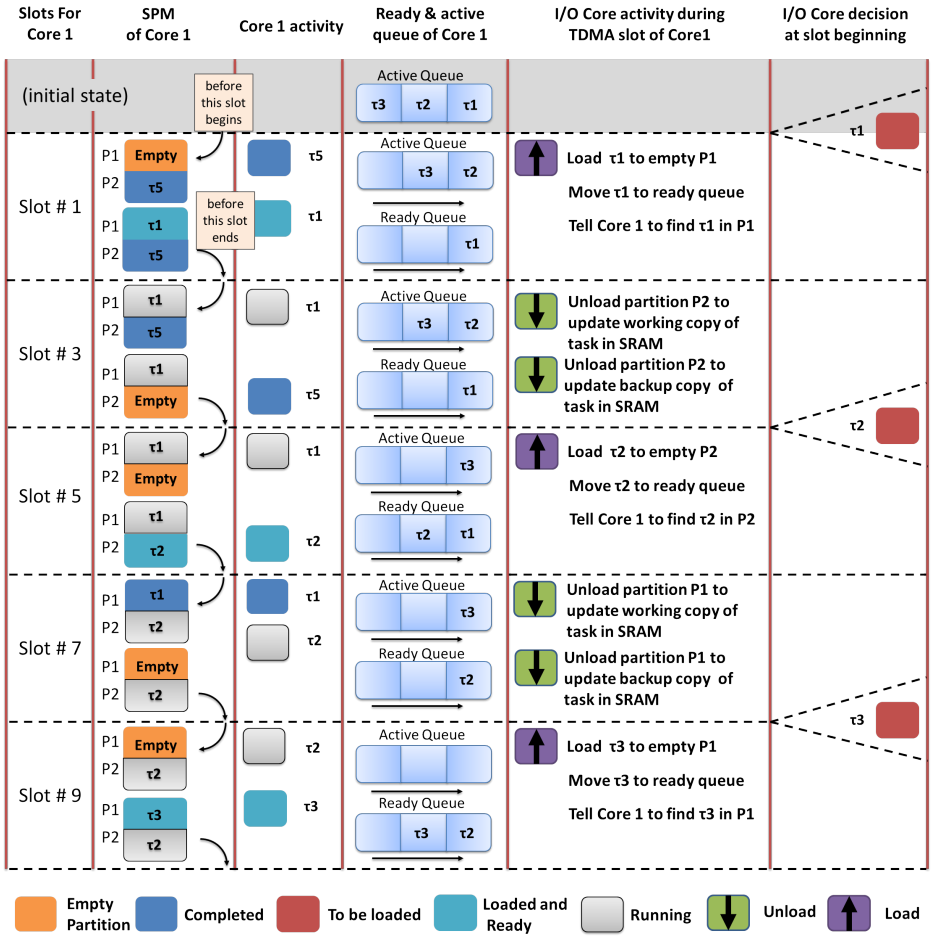


Figure 4.15: Interaction between I/O Core and Core 1 for task scheduling.

The work-flow followed by an applicative core and the I/O core at the boundary of each TDMA slot is depicted in Figure 4.15; this work-flow implements the scheduling rules for a fixed-size DMA operation system described in Section 3.3. Specifically, at each time slot, the I/O core checks the status of the queue of active tasks belonging to the considered core. If a task that is active for execution but not ready (i.e. not relocated in scratchpad) is found, the I/O core checks which SPM partition (P1 or P2) is empty on the application core. If any partition is found to be empty (Slot #1), the I/O core programs the application DMA to load the topmost active task to the empty partition. Once the load is complete,

the I/O core updates the active and ready queues of the considered application core. The latter operation allows the application core to begin the execution of the task (Slot #2). Note that since only one task can be in running state on the CPU, there is always a SPM partition that is available for load/unload operations.

I/O Subsystem Design

Together with memory resources, applications typically need to communicate with peripherals and thus require I/O data to operate. We propose an I/O subsystem design that enforces a complete separation between task execution and the asynchronous activity of I/O peripherals: this goal is achieved by offering to application tasks a synchronous view of I/O data. It is achieved by distinguishing between data production and their dispatch to/from tasks. In fact, we allow I/O data to flow from/to I/O subsystem to tasks only at the boundary of load/unload operations.

As previously mentioned, a dedicated bus connects the SPM of I/O core with peripherals. Hence, asynchronous peripheral traffic can reach the I/O subsystem without interfering with task execution. For each device used in the proposed system, the OS defines a statically positioned *device buffer* on the I/O core scratchpad. A device buffer is further divided into a *input device buffer* and a *output device buffer*. The input and output device buffers represent the positions in memory where data produced by devices and tasks (respectively) is accumulated before being dispatched to tasks or devices.

In our design, peripheral drivers can operate with an interrupt-driven or polling mechanism. For DMA-capable peripherals supporting interrupt-driven interaction, the driver only needs to specify the address in SPM of the device buffer from/to where data are transferred. The driver is also responsible for updating device-specific buffer pointers to prevent a subsequent data event from overwriting unprocessed data. For interrupt-driven interaction with non-DMA-capable devices, the driver uses the platform peripheral DMA to perform data movement. Similarly, the device driver is periodically activated and the peripheral DMA is used to perform data transfer for polling-based interaction with devices.

In general, device-originated interrupts as well as timer interrupts for device driver activations are prioritized according to how critical is the interaction with the considered device. Nonetheless, all the device-related events are served with priority levels that are lower than task-scheduling events, such as: (i) TDMA slot timer events and (ii) completion of application DMA loads/unloads.

In order to interface with a peripheral, application tasks define subscriptions to I/O flows. A subscription represents an association between a task and a stream of data at

the I/O device. For instance, a given task could subscribe for all the packets arriving at a network interface with a specific source address prefix. Task subscriptions are metadata that are stored within the task descriptor.

For each task in the system, a pair of buffers (for input and output respectively) is defined on the SPM of the I/O core to temporarily store data belonging to subscribed streams. Since the content of these buffers will be copied to/from the application cores upon task load/unload, we refer to them as *task mirror buffers*. Consider the arrival of I/O data from a device. As soon as the interaction with the driver is completed, the arrived data is present in the corresponding device buffer. According to task subscriptions, the OS is responsible for copying the input data to all the mirror buffers of those tasks subscribed to the flow.

The advantage of defining mirror buffers lies in the fact that when a task needs to be loaded, all the peripheral data that need to be provided are clustered in a single memory range. Consequently, during the loading phase of a task, the application DMA is programmed to copy the content of the mirror input buffer together with task code and data images to the application core. The reverse path is followed by task-produced output data during the task unload phase.

Since I/O data are delivered to applicative tasks at the boundary of load/unload operations, the approach presented in Section 3.3 for the calculation of tasks' response time can be reused to reason about end-to-end delay of I/O-related events.

Error Recovery

The design proposed so far attains the goal of achieving predictability in a multi-core environment. However, in the event of a memory error, it does not provide any recovery countermeasure. In order to augment the OS to recover from a memory error, we leverage on the existing redundancy that is built-in by design in multi-phase task system.

In fact, in our system any read-only data of a running task (in SPM) is duplicated in SRAM. On top of that, two copies of the R/W portion of the task are kept inside the SRAM. Although one can also keep two copies of the read-only data of the task inside the SRAM, this is not required because an additional copy of the read-only data is always available in flash.

The redundant copies of a task inside the SRAM are used to recover the system from a faulty state and allow to correclty handle one error every two periods of the same task in any of the memory modules. First, we describe how the overall system with redundant

task copies works, then we explain how a fault in each of the memory modules (i.e. SRAM, SPM and flash) is handled.

During normal operation of the OS, both the two copies of R/W data and one read-only copy of the task in the system are the working copies. The OS data structures are initialized with proper information about both copies as working copies and one of them is marked as currently being used. When a task becomes active on an applicative core and the TDMA slot for this core arrives on the I/O core, the I/O core first checks if there is an empty partition available in one of the partition of the SPM. If an empty partition is available, the I/O core programs the DMA to move the task from the SRAM copy marked as currently being utilized into the SPM. Upon DMA completion interrupt, the I/O core sends an interrupt to the applicative core for which a load is being performed. If both of the partition are found to be full and none of the task on these partition is marked as completed, then the I/O core does nothing during this slot. However, if any of the SPM partition has a task that is marked as completed. The I/O core programs the DMA to unload the task from the SPM into the SRAM. Once this operation is successful, the I/O programs another unload DMA operation to download task from SPM into the SRAM to update the second copy of the task in the SRAM. An overview of the scheduling approach in case of normal operation is depicted in Figure 4.15.

As described in Section 3.4, In case of memory errors, additional operations need to be performed as shown in Figure 3.8. In this case, the on-chip ECC modules detect memory errors only when a read is performed on block affected by the bit flip. Upon detection of an error, the ECC modules are programmed to (i) generate an interrupt to the application cores and to (ii) report the memory address where the error occurred. In our system, we consider the occurrence of memory errors in three different kinds of memories: SRAM, SPM and flash. We now describe the proposed error recovery strategies.

Fault happens inside the SPM: Based on when a fault can be detected, the fault inside the SPM can be categorized into two types: the first case corresponds to a fault that happens in the read-only or read/write memory of the task during its execution; in the second case, a fault in read/write memory is detected during unload.

Whenever a fault is detected, all the applicative cores receive an interrupt from the error detection logic. Upon receiving the interrupt, all the applicative cores execute ISR 4.1 and check if the memory location that caused the error lies within their SPM memory range. Only the applicative core whose SPM was affected by the fault further executes the ISR 4.1. The affected core further checks if the error happened during the execution of the task or during the unload phase. This can be determined based on the SPM partition affected by the fault, since the OS keeps track of the state and location of each task.

ISR 4.1 ISR on Application Cores From Error Detecting Logic

```
1  MEMU ISR on Applicative Cores()
2      if ErrorAddress within local core SPM Range then
3          if Error during execution then
4              Mark the SPM partition as empty
5              Reschedule task with highest priority
6          else if Error during unload then
7              if Error at first unload then
8                  Update descriptor to use second copy of R/W data for next task
                    load
9                  Mark SPM partition as empty
10                 Reschedule task with highest priority
11             else if Error at second unload then
12                 Mark second copy of R/W data as faulty
13                 Handle task as successfully completed
14             end
15         end
16     end
```

In case the error is detected during the execution of the task, the applicative core marks the SPM partition from where the task was executing as empty and reschedules the task with the highest priority. This guarantees that the task is reloaded at the next TDMA slot for this particular core, thus improving the worst-case response time of the task as discussed in Section 3.4.1. This case is captured in ISR 4.1 at lines 3-5.

In the second case, the error is detected during the unload phase of the task. As previously mentioned, during an unload operation, only read/write data are copied from SPM to SRAM twice. The error can occur during the first or the second redundant copy. In the first sub-case (A), the task had correctly terminated its execution phase, and the error is detected before the first copy of task R/W data from SPM to SRAM is completed. In this case, the second copy in SRAM is not updated, so that valid data from the previous task execution are not overwritten with faulty data. Conversely, the first (faulty) copy in SRAM is marked as faulty, so that the backup copy will be used at next reload. Next, we mark the SPM partition from where the task was unloaded as empty and reschedule the task with highest priority, like in the previous case. This scenario is handled in ISR 4.1 at lines 7-10. In the second sub-case (B), the error is detected inside the SPM after the first copy was successfully unloaded, and while the redundant copy was being updated. In this case, we mark the second copy in SRAM as faulty and update the OS data structures to use the first copy in the SRAM at next load. Since the task has successfully completed, no task restart is required. This scenario is captured in ISR 4.1 algorithm at lines 11-14.

Fault happens inside the SRAM: As described earlier, there are two copies of the

R/W data inside the SRAM and one copy of the read-only data inside the SRAM, whereas, a second copy of for the read-only data resides in flash. An error in SRAM can be detected only when a task is transferred from SRAM to SPM (load). Potentially, the fault could be directly reported to the I/O core by registering the corresponding interrupt. For faults in SRAM, however, we do not register and interrupt with the memory error management unit. Instead, we follow a synchronous approach: at every DMA completion interrupt, the OS checks if any error was reported by the ECC circuitry. In case of a positive outcome, the faulty address is derived.

If the faulty address lays within the memory range being loaded from SRAM, two cases are possible. In case (A) the error affected the copy of R/W data used for the transfer, the task is not marked as "ready" and its descriptor is updated to repeat another load at the next TDMA slot using the backup R/W data copy. In the second scenario (B), the fault affects the read-only data of the task in SRAM. In this case, the I/O core directly copies the word that was corrupted by the error from flash to both the SRAM and the SPM. No task re-load needs to be performed. The complete procedure handling cases A and B is described in ISR 4.2.

ISR 4.2 DMA completion Interrupt

```
1  DMA Completion Interrupt()
2      if DMA completion interrupt for load then
3          if Unrecoverable error bit is set then
4              if Error in SRAM R/W data region then
5                  Directly copy the word that caused error from backup SRAM copy
                     to faulty SRAM copy as well as to the SPM
6              else if Error in SRAM read-only data region then
7                  Directly copy the word that caused error from flash to SRAM as
                     well as to the SPM
8              end
9          Resume regular operations for DMA completion – such as IPI to application
             cores after moving task into ready queue.
10     end
```

Fault happens inside the Flash: The flash in our system is used during the bootup process. We keep two copies of the OS image in flash. At bootup, we bring the task read-only data from flash into SRAM. Moreover, we also allocate the R/W data of each task in SRAM. At anytime during the bootup, if an error is detected in the first working copy of the flash, we switch to the second copy of the OS image in flash. Next, we repeat the bootup procedure from the new location in flash.

4.2.3 Evaluation

To validate the proposed design and implementation, we performed a series of experiments, whose results are summarized in this section. First, we investigate the overhead of SPM management. Next, we consider the performance and predictability benefits of our approach with synthetic as well as real benchmarks. The achievable I/O bandwidth supported by our design is also measured. Overhead of Software versus Hardware error recovery is also evaluated. Finally, we investigate the schedulability results of the proposed strategies.

SPM-Centric OS Overhead Evaluation

The most important parameter of our proposed system is the size of the TDMA slot. The slot needs to be long enough to account for the load/unload of any task in the system. In order to calculate the upper bound, we restrict the slot size to be greater or equal to the footprint of the task with maximum size among the benchmarks. In addition, in the case of recovery mechanism, the TDMA slot size must be long enough to allow two unloads of the completed task. Therefore the TDMA slot size is equal to $\max(worst\_task\_load, 2 \cdot worst\_task\_unload) + IO\_core\_ISR\_overhead$. Table 4.6 shows the system parameters including DMA times and TDMA slot size.

We have measured the DMA modules configuring overhead. In addition, since the interrupt from the memory error management unit is delivered to all the application cores, we have measured the overhead of ISR 4.1. This ISR has minimum and maximum overhead. The minimum overhead occurs when the ISR only checks if the error address is relevant or not and concludes by its irrelevance. On the other hand, the maximum overhead is experienced in the case of a relevant memory fault. We also measured the maximum overhead of I/O core ISR 4.2 for DMA completion interrupt. All of these results are reported in Table 4.6.

Results of Achievable I/O Bandwidth

The performance of the proposed I/O subsystem (see Section 4.2.2) depends on the frequency of load/unload operations. In order to measure the achievable I/O bandwidth of proposed design, we have implemented support for the onboard Fast Ethernet Controller (FEC). The FEC is capable of transmitting data at the highest bandwidth among all the devices of the considered MCU. Hence, it represents the best I/O component to stress-test our design.

Table 4.6: Details of OS Parameters

Parameter	Time (μs)
DMA Load time (Largest Code, R and R/W Data size)	209
DMA Unload time (Largest R/W Data size)	61.5
DMA setup	3.16
Context switch	0.46
Minimum ISR overhead on Applicative cores	0.70
Maximum ISR overhead on Applicative cores	8
Maximum ISR overhead on I/O core for DMA completion	2
TDMA slot size	215

We have connected the FEC to an external node which generates constant-rate traffic. Specifically, the traffic source generates a 1 KB packet every 100 μs (1000 Hz, about 82 Mb/s). The payload of each packet contains a flow-ID chosen from 4 different values in round-robin. On used MCU, each applicative core runs two tasks that have subscribed to I/O data flows based on packets' flow-IDs. Device buffers and task (mirror) I/O buffers have been dimensioned to accommodate a single packet per task, with an overwrite policy.

With this setup, we have derived the raw achievable bandwidth considering two different values of TDMA slot size. Specifically, we measured the data rate of packets that are processed and looped back on the network interface using the Wireshark packet analyzer[4]. Our experiments revealed an achievable bandwidth for the outgoing traffic of 4 Mb/s with a TDMA slot of 800 μs, and 8 Mb/s with a TDMA slot of 400 μs. Although this represents a fraction of the physically available bandwidth (100 Mb/s), being able to sustain a bandwidth higher than 1 Mb/s constitutes a promising result given that the platform operates at a clock frequency of few hundred Hz.

[4] https://www.wireshark.org/

Results of Synthetic Benchmarks

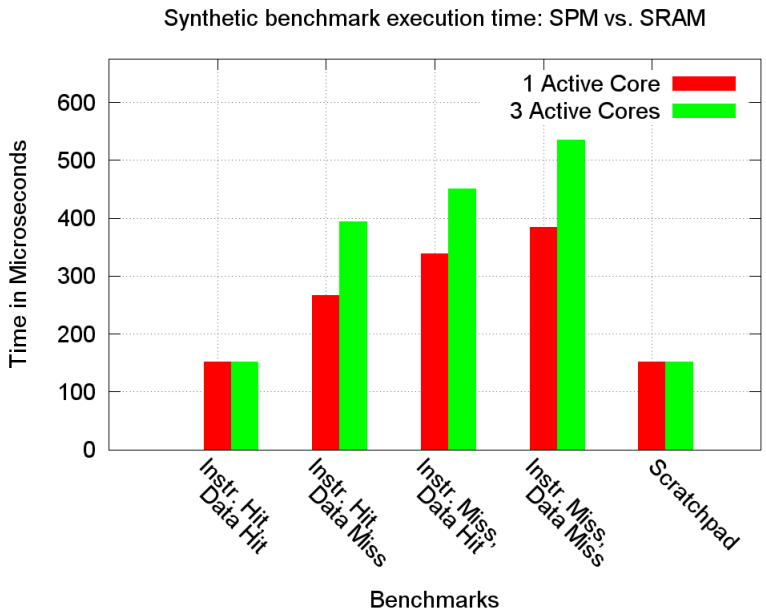


Figure 4.16: Experimental execution time for synthetic benchmarks.

We investigate the performance of SPM-based execution as opposed to a traditional execution model. For this purpose, we have developed a set of synthetic benchmarks that exhibit different memory access patterns. Figure 4.16 depicts the runtime for such benchmarks on one of the two applicative cores. The first cluster of bars refer to the runtime of the benchmark that exhibits good data locality. Hence, when it is executed from SRAM, caches are effective at hiding SRAM access latency and significantly reduce task execution time. The next two clusters of bars show that misses suffered for only instruction fetches or only data fetches already induce a significant execution slowdown (around 2x). The need for accessing SRAM data also introduces runtime fluctuation (about 25%) as a result of inter-core interference. Such effect becomes even more severe with applicative code that experiences misses while accessing both instructions and data. If the cost of accessing SRAM memory together with the slowdown due to inter-core interference are considered, an overall 3.5x slowdown is experienced when compared to what has been observed in the ideal case (100% cache hits). Finally, notice that if a task is able to entirely execute from scratchpad, its execution time is comparable to the ideal case and inter-core interference is

prevented. These results are a strong motivation to best use available scratchpads in order to improve performance and avoid inter-core interference.

Results of EEMBC Benchmarks

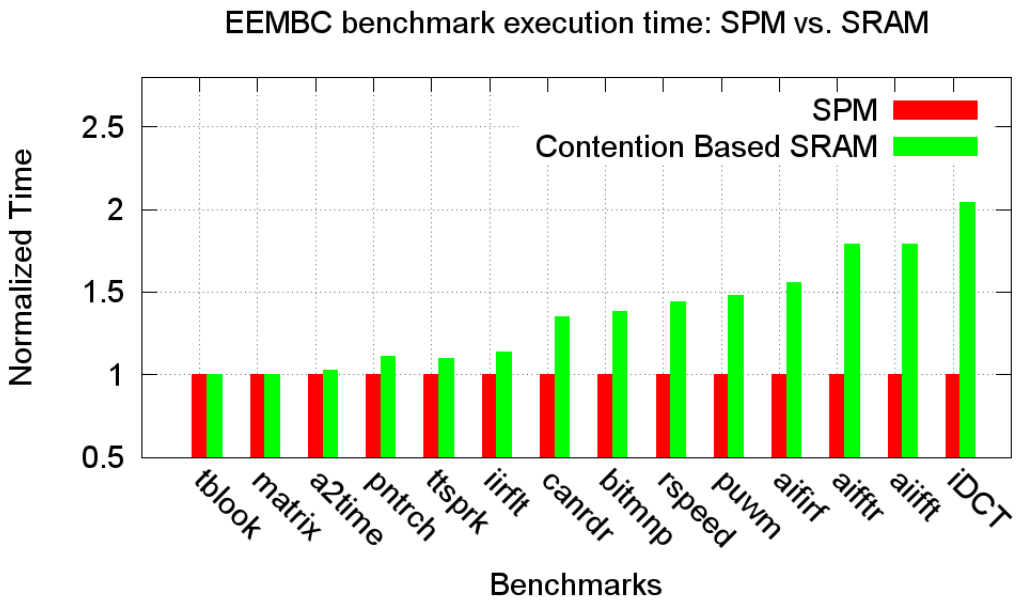


Figure 4.17: Experimental execution time for EEMBC benchmarks.

Next, we investigate the behavior of EEMBC benchmarks on the selected platform. For this purpose, we have ported and measured the execution time of the full suite of automotive EEMBC benchmarks under two scenarios: traditional contention-based execution from SRAM and proposed SPM-based execution. The results of normalized execution times are shown in Figure 4.17. From the results, we note that computation intensive benchmarks do not benefit from SPM-based execution. Conversely, for memory intensive benchmarks SPM-based execution determines substantial speed-ups (up to 2.1x).

Table 4.7 shows the execution time of the EEMBC automotive benchmarks. Furthermore, Table 4.7 also provides the footprint size of the considered benchmarks. It can be noted that all the considered benchmarks fit into a single scratchpad partition. These results validate the applicability of proposed design in real scenarios. Note that, we split

Table 4.7: Details of EEMBC Benchmarks.

Benchmark	SPM Time (μs)	SRAM Time (μs)	Relocatable Code Size (bytes)	Read only Data Size (bytes)	Read/Write Data Size (bytes)
tblook	1013	1015	1892	10916	60
matrix	1053	1054	4774	12188	124
a2time	1002	1029	2538	1704	148
pntrch	1036	1145	1398	4800	128
ttsprk	383	425	4772	2592	4848
iirflt	1040	1189	3512	888	248
canrdr	1009	1359	1562	12276	56
bitmnp	990	1389	3282	72	1494
rspeed	1012	1457	1208	13200	40
puwm	1036	1540	2500	2400	180
aifirf	1005	1564	2286	1120	84
aifftr	916	1642	4458	2304	1912
aiifft	1170	2092	3540	3072	1656
idct	1045	2126	4690	244	1788
Total size			42412	67776	27766

the sizes of the read-only data and read-write data of the applications as it is relevant to our recovery mechanism.

Overhead of Software versus Hardware Recovery

The proposed software-based recovery technique is able to correct errors that can be detected (only) but not corrected by the hardware. If the hardware provides double-bit error correction (DEC) capabilities, then our approach is redundant and should not be used. However, if the hardware, as it is often the case, supports only SEC-DED, then we introduce a timing penalty to recover 2-bit errors that are detected but not corrected by the hardware. Under SEC-DED, 1-bit errors are still corrected only by the hardware. Introducing a timing overhead effectively means trading schedulability for reliability. We quantify this trade-off in Figure 4.18 and 4.19. Clearly, when software-based recovery is used, the time overhead can be significant. Nonetheless, we believe that it is reasonable to trade a portion of the CPU utilization to make sure that safety-critical tasks produce

correct outputs.

Let us focus on the typical ECC support provided by commercial hardware, i.e. SEC-DED. We hereby compare the **overhead in space** introduced by our software-based recovery mechanism with an equivalent hardware-based implementation, i.e. DEC support in hardware. Introducing DEC support in hardware is possible, but its implementation is complex and costly. In fact, it has been shown [119] that the ASIC area can grow up to 13x. The software approach, conversely, can be deployed on less expensive hardware and only requires to keep redundant copies of the R/W portion of application memory. The SEC-DED requires 8 check bits to implement the detection and correction mechanism for a 64-bit data word, while DEC requires 14 check bits to correct a double bit error [119] in a 64-bit data word. Using this information, we compute the number of extra bits required for hardware-based (HW) SEC-DED, hardware-based DEC, and our proposed software-based (SW) recovery mechanism for DEC in Table 4.8. In the table, we assume that our taskset is comprised of all the benchmarks in Table 4.7. First, note that our technique relies on SEC-DED and only duplicates R/W data and ECC bits. For read-only (R-only) memory, only the extra bits required to implement SEC-DED are considered, because a second copy of read-only data is always available in less expensive flash memory. Moreover, for a fair comparison, we assume that HW DEC support is only provided for application memory, instead of the whole main memory. In this setting, the SW overhead is about 10%. We argue however that (i) not requiring additional ECC check logic in the SW approach partially offsets the cost for the additional memory bits; and that (ii) if extra DEC bits were considered on the whole main memory, the SW approach is to be preferred in terms of overhead: in fact, with 512 KB of main memory, DEC would require about 917,504 extra bits.

Next, we consider the **overhead in time** for hardware-based recovery and the proposed software-based technique. According to [119], hardware implementations of SEC-DED introduce a latency of 1.3 ns, whereas DEC implementations introduce a latency of 2.2 ns on every memory transaction. Our SW approach on the other hand still requires SEC-DED, but only introduces a time overhead if a double-bit error occurs. Depending upon which part of the task is affected by the two-bit error, the recovery overhead differs. For instance, an error in the read-only data only requires one word to be copied from a redundant copy, whereas, an error in read/write data of the task in the worst-case may require the entire task to be re-loaded and re-executed. It is hard to compare the time overhead of software-based and hardware-based approaches, because it depends on the number of memory accesses performed by the tasks under analysis. Nonetheless, we expect that hardware-based DEC implementation can be more efficient in time. As vendors typically only provide SEC-DED support, however, we believe that offering the choice at design time to recover double-bit

Table 4.8: Space Overhead of HW versus SW recovery

	HW SEC-DED (bits)	HW DEC (bits)	SW DEC (bits)
Number of extra bits required for R/W data (12,766 bytes)	12,766	22,344	127,660
Number of extra bits required for R-only data (42,412 + 67,776 bytes)	110,192	192,836	110,192
Total number of extra bits	80,542	215,180	237,852

errors in software still represents a valuable contribution in the context of safety-critical systems.

Evaluation of Schedulability Analysis

For the schedulability evaluation, we consider the scheduling approach in Section 3.3 with fixed-size DMA time slots, and its associated analysis. For the error recovery case, we consider the analysis in Section 3.4. We first present the schedulability curve of normal case when there are no errors and compare it with the case when we have errors. We also show the contention based approach where we have no error recovery. The case referred as "contention" corresponds to the case where no scratchpad management is implemented and in which tasks execute directly from SRAM utilizing caches, but contending for bus access.

We computed the response time of the same workload for the three cases: the traditional SPM-centric scheduling mechanism with no errors as proposed in Section 3.3; the augmented SPM-centric OS with error recovery mechanisms using the analysis in Section 3.4 and with artificial error injection; and the contention based execution using standard response time analysis. For the evaluation, we have considered the EEMBC benchmarks in Table 4.7 and the overheads in Table 4.6.

For a given system utilization, each application is randomly selected and assigned a random period in the range between 10 ms to 100 ms. The tasks utilization is then computed based on the measured execution time of the applications and the selected period.

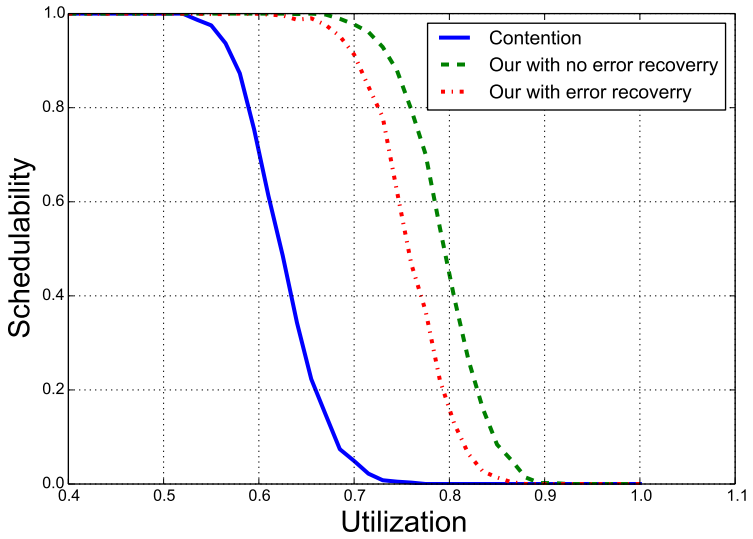


Figure 4.18: Schedulability with SPM-based and traditional scheduling models.

Tasks are randomly generated until the sum of the individual tasks utilizations reaches the required system utilization.

Figure 4.18 shows the result of the schedulability analysis of our proposed SPM-centric schemes with and without error recovery and compare them with the case of contention-based system. The figure shows the results in terms of proportion of schedulable task sets for both approaches. Each point in the graph represents 1000 task sets. The results show that our proposed schemes significantly improve the system schedulability compared to the contention. Hence, the described SPM-centric scheduling not only improves the predictability of task execution, but it also improves task set schedulability by hiding the main memory access latency, especially for memory intensive applications.

Based on the figure, we can conclude that there is limited degradation in schedulability for supporting the recovery mechanism. This degradation can be justified by the fact that the system is both predictable as well as fault tolerant. Moreover, from the Figure 4.18 we can also see that our SPM-centric approach with error recovery still performs significantly better than the contention-based case where no error recovery is performed.

Figure 4.19 shows the system utilization when 50% of the task sets are schedulable. The X-axis represents the window of periods used to generate the task sets. As expected in any non-preemptive scheduler, with tight periods all three mechanisms degraded due to

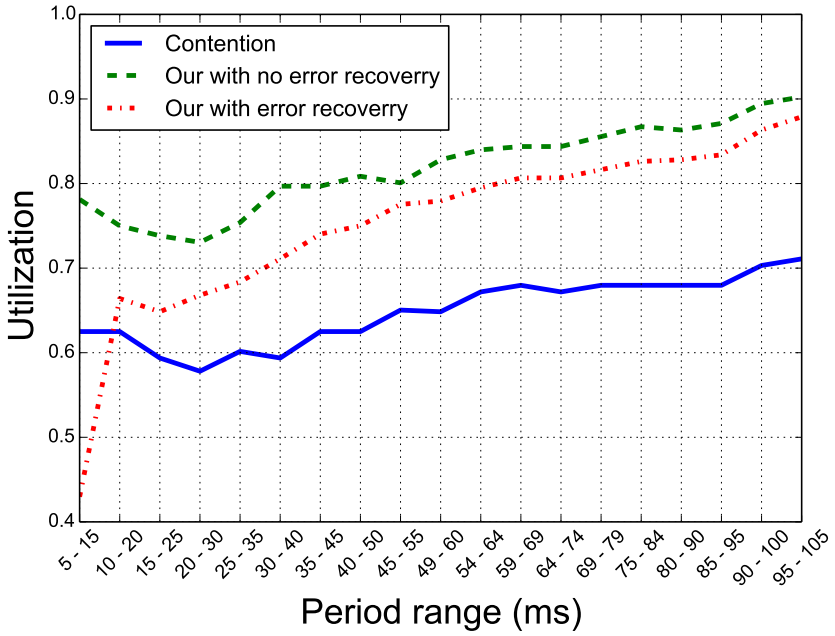


Figure 4.19: Utilization degradation as a function of tasks periods

the blocking time. However, with error recovery the degradation is more severe in the case of very small periods due to the extra overhead paid for recovery. In most cases, with error recovery, the proposed strategy achieves better utilization compared to the contention-based approach. Additionally, the loss in utilization arising from additional error recovery overhead remains within an acceptable range, and marginally decreases for larger task periods.

4.3 Summary

In this chapter, we have demonstrated how our proposed 3-phase task execution model can be realized on actual embedded platforms. The FPGA-based platform detailed in Section 4.1 allows us full control over the hardware implementation; we took advantage of it by implementing a simple and efficient address translation scheme which simplifies the task of relocating programs between scratchpad partitions. As such, only small modifications are required to support the proposed scheduling scheme in the FreeRTOS kernel. On the

Table 4.9: Suitable Commercial Multicore COTS platforms

Features	MPC5777M	MPC5746M	TMS320C6678
Scratchpad	✓	✓	✓
DMA engines	✓	✓	✓
Dedicated I/O bus	✓	✓	✗

other hand, the platform does not integrate I/O with the proposed 3-phase model, and does not support fault-tolerant task execution.

The Freescale MPC5777M COTS platform described in Section 4.2 represents a more complete implementation, including I/O management and ECC memory support. It is also provides higher performance thanks to the use of hard, rather than soft cores. However, limitations on the availability of DMA engines and arbitration for main memory forced the implementation of a more complex DMA scheduling mechanisms, which results in a slightly less efficient, fixed-size DMA operation scheme. In this case, program relocation is achieved through compiler, rather than hardware support. While the described implementation is specific to the analyzed MPC5777M SoC, we would like to stress that the required hardware features for COTS implementation are becoming increasingly common in micro-controllers used for safety-critical applications. Table 4.9 provides a list of some of the available COTS platforms that provide relevant features.

Finally, in order to validate the proposed execution model and implemented platforms, we have combined experimental results from synthetic and automotive EEMBC benchmarks. In addition to the strong temporal predictability achieved by enhancing inter-core isolation, we are able to exploit the performance benefits of scratchpad memories. This results in improved schedulability compared to both the traditional contention case, where cores contend for access to main memory, as well as compared to previous approaches based on the 3-phase model.

Chapter 5

Global Scratchpad-Centric Scheduling of 3-Phase Real-time Tasks

In Chapter 3, we showed how to execute 3-phase tasks based on a fixed-priority, partitioned scheduling scheme. Although the proposed partitioned approaches showed good results in terms of hiding access latency to main memory, a partitioned system is not always preferable as it requires design-time decisions to statically assign tasks to cores. Furthermore, for systems with dynamic task admission, the whole system needs to be re-partitioned, making the system inflexible. On the other hand, global scheduling is often preferable for systems comprising both hard and soft tasks, and parallel tasks with utilization greater than one. Therefore, in this chapter we consider a global scheduling scheme to overlap task execution with memory access. Compared to the approaches in Chapter 3, our global scheme directly handles contention among cores for access to main memory without relying on either a fair hardware arbiter, or software-based TDMA arbitration.

The main contributions of this chapter are (1) a global scheduling algorithm for a set of sporadic, sequential real-time tasks that efficiently co-schedules the cores and the DMA to hide the memory access latency and provide a predictable system behavior. In particular, since main memory is the unique shared resource in the system, we show that global scheduling decisions must be driven by DMA operations. We also derive (2) a novel schedulability analysis for our proposed algorithm. The nature of co-scheduling DMA and cores requires us to largely re-define the existing concepts of workload and interference in global real-time scheduling. We show that a new concept of *scheduling interval* is required

to account for the workload generated by both computation and memory activities. (3) We demonstrate the performance of our system by intensive simulations based on measured benchmark characteristics.

The rest of the chapter is organized as follows. In Section 5.1, we introduce our task model. Note that the model is similar to the one employed in Chapter 3, and furthermore we make the same assumptions regarding the hardware platform (that is, the system provides a DMA engine in conjunction with per-core, dual-ported SPM). Section 5.2 describes our scheduling algorithm, and Section 7.2 details its associated schedulability analysis. We evaluate our approach in Section 7.3, and summarize our results in Section 7.5.

5.1 Task Model and Notations

We consider the global fixed-priority scheduling of a system consisting of n sporadic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, sorted by decreasing priorities such that τ_i has greater priority than τ_j if and only if $i < j$. We use J_i to denote any system job. The time of computation phases is denoted by x_i. Similarly, the time of load phases is denoted by $load_i$ and the unload phases by $unload_i$. The relative deadlines D_i are assumed to be less than or equal to the periods p_i (i.e., constrained deadlines). The absolute deadline for a job J_i is $d_i = r_i + D_i$, where r_i is the release time of J_i. We use $hep(k)$ to indicate the set of all tasks with priority higher or equal to τ_k, and we use $lp(k)$ to indicate the set of all tasks with priority lower than τ_k. Finally, we define the *total CPU utilization* as $U^{cpu} = (\sum_{i=1}^{n} x_i/p_i)$.

We use $J_a \rightarrow J_b$ to denote that J_b is the next job to run immediately after J_a on the same core. We use $t_s(J_a)$ to denote the start time of the computation phase of J_a, and $proc(J_a)$ to denote the core where J_a is executed. In addition, we use $\text{top}(Y, x)$ to return the summation of the largest x items of set Y. All time values are assumed to be non-negative integers and expressed as cycles of the most precise clock in the system.

5.2 Scheduling Algorithm

In this section, we first start with a description of our scheduling algorithm followed by a working example; we then move in Section 5.2.1 to discuss the design of our scheduler.

We note that in our execution model, each task must execute its load phase on the DMA before its computation phase on a core. After the computation phase, modified data has to be unloaded to main memory. However, having to schedule two DMA operations

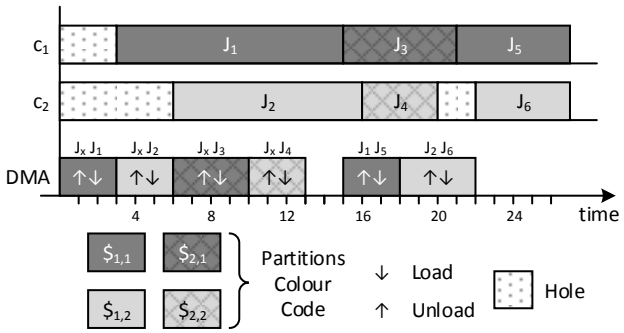


Figure 5.1: An example of scheduling 6 jobs on 2 cores.

(load and unload) complicates the schedulability analysis for dynamically scheduled tasks. Thus, we propose to combine the unload of one job to the load of the next job executed out of the same partition. Suppose that $J_a \rightarrow J_b \rightarrow J_c$. We note at this point that the consecutive execution of jobs on the same core alternate between its local memory partitions. Hence, J_c is the next job to execute after J_a out of the same partition. Then, when J_c is scheduled to be executed on the DMA, the DMA executes the unload phase of J_a non-preemptively with the load phase of J_c. For simplicity, we refer to the combined unload and load phases as the *memory phase* for the loaded job J_c. In addition, both memory phases and computation phases are executed non-preemptively. We note that the memory phase and the computation phase of one task are not necessarily executed continuously because after loading a task, the core might be busy executing non-preemptively another task out of the other partition. In other words, after loading a job into a local memory partition, its content is locked until the finish of its computation phase.

Example 1. Figure 5.1 depicts a working example for scheduling 6 jobs, generated by 6 different tasks, on 2 cores assuming all jobs are released at the same time. Since this is a fixed-priority schedule, the highest priority job J_1 is chosen first. The scheduler chooses c_1 to execute J_1. The DMA is instructed to unload the previous task from $\$_{1,1}$ back to main memory. Then, the DMA is instructed to load J_1 into $\$_{1,1}$. After that, J_1 is able to run on c_1 with no memory stalls. While c_1 is executing J_1 out of $\$_{1,1}$, the scheduler at time 3 chooses J_2 to be executed on c_2. Here, the DMA is running in parallel with J_1 by unloading $\$_{1,2}$ and loading it with J_2. At time 6, the scheduler chooses the free partition of c_1 to execute J_3. Similarly, J_4 is chosen at time 10 to execute on the free partition of c_2.

At time 13, all four partitions are loaded. Hence, the memory phase of J_5 has to wait until time 15, the finish time of the computation phase of J_1 which indicates that c_1 has again a free partition. Thus, the scheduler at time 15 chooses J_5 to be executed on c_1.

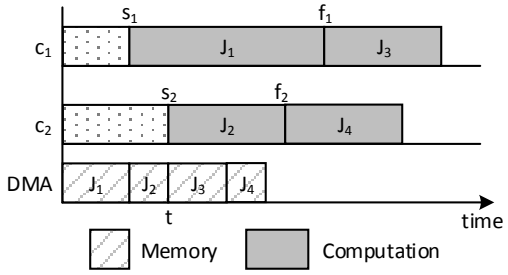


Figure 5.2: The cores are chosen based on the minimum s_k.

Finally, J_6 is scheduled at time 18 to execute on c_2. We note that even though c_2 has finished execution at time 20, J_6 has to wait until time 22 because its memory phase is delayed. This delay induced a schedule *hole* between J_4 and J_6. We define a schedule hole as the time at which the core is idle waiting for a task to be loaded.

As you can see, the memory phases in the example schedule are largely overlapped with computation phases with a few induced schedule holes. This hiding of memory phases, gives our system a better schedulability over other state-of-the-art global scheduling techniques as shown in Section 7.3.

In what follows, we discuss how our scheduler chooses cores to schedule tasks. In Figure 5.2, we show a schedule of 4 jobs and 2 cores. The time pointers s_1 and s_2 indicate the start time of last scheduled jobs on c_1 and c_2, respectively. Similarly, the time pointers f_1 and f_2 indicate the finish time of last scheduled job on each core. Consider the time t at which each core has a free partition. Our scheduler chooses c_1 to schedule J_3 rather than c_2 because c_1 has earlier start time of last scheduled job i.e., $s_1 < s_2$. We design our scheduler to choose cores based on start time of their last scheduled jobs rather than the finish time to avoid the pessimism in the analysis. In particular, it gives us the guarantee to bound the amount of holes between computation phases as we discuss in Section 5.3.2.

5.2.1 Scheduler Design

Our scheduler maintains a global queue Q_r in which ready tasks are ordered according to fixed priorities. Whenever a task is released, it is *inserted* in this global queue. The dispatcher *extracts* from the top of the queue the highest priority task and execute it on the DMA, given that the DMA is idle and there is at least one available partition; otherwise,

the job remains inside the global queue. Furthermore, the scheduler is usually implemented as an interrupt service routine (ISR) triggered by certain events. In our system, we have the following three events: (1) task release, (2) memory phase completion and (3) computation phase completion. `DMA-Dispatcher` procedure below is triggered at time t, corresponding to one of these three events, to schedule a new task on the DMA. In addition, after events (2) and (3), if a new task has already been loaded, the core will do a context switch and execute this task.

For example, consider Figure 5.1 again. At time 3, the completion time of the memory phase of J_1, c_1 will do a context switch to execute J_1 and at the same time J_2 will be scheduled to execute on the DMA. At time 15, the completion time of the computation phase of J_1, c_1 will do a context switch to execute J_3 since it has already been loaded in $\$_{2,1}$ and at the same time J_5 will be scheduled to execute on the DMA because the completion of J_1 computation phase indicates a free partition.

1: **procedure** DMA-DISPATCHER:
2: $i = \texttt{Select-Task}(Q_r)$
3: $i = \texttt{Select-Task}(Q_r)$
4: $(l, j) = \texttt{Select-Core}(\{s_k\})$
5: $t_{dma} = t + unload_j + load_i$
6: $s_l = \max(t_{dma}, f_l)$
7: $f_l = s_l + x_i$

For simplicity, we assume at time t when `DMA-Dispatcher` procedure is invoked that (1) the DMA is idle, (2) at least one partition is available and (3) there is at least one ready task. Otherwise, the procedure will exit, as we assume a non-preemptive execution, and will be triggered again by a later event. Basically, we have: $\{s_k\}$ and $\{f_k\}$, two sets of m time pointers to indicate the start and finish time, respectively, for the last scheduled job on each core, and t_{dma} to indicate the end time of a DMA operation. `Select-Task` procedure in Line 3 returns the index (i) of the highest priority task out of Q_r. Similarly, `Select-Core` procedure in Line 4 returns the index (l) of the core that has the minimum s_j, with ties broken arbitrarily, among all cores with free partition, and the index (j) of the last scheduled job on the selected partition. We need this j to add the unload time of previous job as in Line 5. Note that s_l is updated in Line 6 to be the maximum between t_{dma} and f_l because J_i can start its computation phase if its memory phase is completed and core c_l is idle.

5.3 Schedulability Analysis

Our schedulability analysis relies on the proof technique introduced in [27]. This technique is based on the argument that if the execution time of a job J_k plus its *interference* from other tasks in $[r_k, d_k)$ is less than or equal to D_k, then J_k will meet its deadline because the scheduler is work-conserving. J_k is called a *problem job* and the time window $[r_k, d_k)$ is called a *problem window*. We note that the *interference* is an important concept that can be defined as the time at which the problem job is not able to execute inside its problem window despite being released, due to other tasks.

Since each job in our system executes on both the DMA and a CPU core, one has to consider both in computing the interference for a given job. However, our analysis in this section works constructively by considering only the computation phases on CPU cores. We use $I_k(J_i)$ to denote the workload of individual job J_i including both the computation time and the induced hole as shown in Figure 5.1. We use $I_k(\Gamma)$ to denote the workload of all tasks inside the problem window of J_k. Furthermore, the interference is denoted by δ_k and is computed based on $I_k(\Gamma)$. Since we only consider the computation phases, we define the time window $[r_k, t_l]$ where $t_l = d_k - x_k$ as our new problem window, and we use $L_k = t_l - r_k + 1$ to denote its length; if the problem job starts its computation phase by time t_l, then it completes before its deadline as we assume a non-preemptive computation phases.

In multiprocessor scheduling, finding the maximum interference of a task system over an interval of time is an NP-hard problem [102, 33]. Thus, a widely used technique in real-time literature is to assume that each task cannot interfere with more than its worst case activation of jobs [40]. The interfering jobs of task τ_i in a problem window of task τ_k is composed of three parts. (1) *Carry-in*: the contribution of at most one job with release time r_i before r_k and $r_i + p_i$ after r_k. (2) *Body*: the contribution of jobs with both r_i and $r_i + p_i$ inside the problem window. (3) *Carry-out*: the contribution of at most one job with release time r_i inside the window and $r_i + p_i$ outside the window. To account for the worst-case activation, we should assume that jobs are packed as much as possible within the problem window. That is, carry-in jobs start as late as possible and carry-out jobs start as early as possible.

Compared to classic global scheduling theory, there are two main differences in our system for bounding the interference: (1) long memory phases can induce a "hole" between successive computation phases on the same core; hence, the interference must include such holes. (2) In a classic fixed-priority global scheduling system, a problem job J_k would start on the earliest processor that becomes idle. Hence, the worst case interference for J_k can

be bounded by $I_k(\Gamma)/m$. Since in our system we have to bind a job to a core partition once we start its memory phase, such bound does not hold anymore.

The rest of this section proceeds as follows. In Section 5.3.1, we show how to bound the interfering jobs inside the problem window of J_k, for a given task system Γ. The bound on the workload of individual jobs $I_k(J_i)$ is discussed in Section 5.3.2. We then show in Section 5.3.3 how to derive a global bound on $I_k(\Gamma)$, the workload of all tasks. In Section 5.3.4, we compute the bound on the interference based on $I_k(\Gamma)$. Finally, we present the schedulability condition for our scheduling algorithm in Section 5.3.5.

5.3.1 Bounding the Interfering Jobs

In this section, we first bound the interfering jobs inside the problem window including carry-in, body and carry-out jobs. Then, we populate out of these jobs three sets: X, LD and UD, the set of computation, load and unload phases, respectively. We will use these sets in Section 5.3.3 to derive a global bound on $I_k(\Gamma)$.

A job J_i can interfere inside the problem window of J_k in two different ways: (1) interference caused by non-preemptive scheduling, and (2) interference caused by priority order. Tasks in $lp(k)$ can only have carry-in interference because they cannot start executing after the beginning of the problem window. In contrast, tasks in $hep(k)$ can be activated inside the problem window. Thus, they can have carry-in, body and carry-out jobs. However, assuming all tasks in $hep(k)$ can have carry-in jobs is quite pessimistic. Hence, we start this section by redefining the problem window in order to bound the number of carry-in jobs from $hep(k)$. Then, we show how to bound the interfering jobs that can execute inside the problem window.

Carry-in Limit

We propose to extend the problem window such that it has an earlier starting point t_o and has the same end point t_l. Now, we define a pending job and t_o as follows.

Definition 1. We say that a job is pending if it has been released but it has not started its DMA.

Definition 2. We let t_o to denote the last time instant before r_k such that for any $t \in [t_o, r_k)$ there is at least one pending task from $hep(k)$. If such time does not exist, we let $t_o = r_k$.

The following lemma bounds the number of tasks that can have carry-in inside the problem window that starts at t_o.

Lemma 1. *There are at most $2m$ tasks from $hep(k) \cup lp(k)$ that can have carry-in jobs.*

Proof. We use $t_o - 1$ to denote the time instant before t_o. The complement of Definition 2 states that at $t_o - 1$ there is no pending task from $hep(k)$; otherwise, we could have extended the window. Based on this and the fact that J_k is released at r_k, t_o always corresponds to a release of a task in $hep(k)$. Since we have $2m$ partitions for each core, only $2m$ tasks from $hep(k)$ could have been released at or before t_o without being pending or having completed. It follows that only these tasks can have carry-in jobs. Similarly, no task in $lp(k)$ can start on the DMA at or after t_o because there is always a pending job from $hep(k)$ in $[t_o, t_l]$. Thus, only $2m$ tasks executing at least one time unit before t_o can have carry-in jobs. Since it is not possible to load more than $2m$ partitions at one time, the lemma follows. □

Since we assume tasks have constrained deadlines, i.e., no two releases of one task can be active at the same time, these $2m$ jobs must be from different tasks. In what follows we distinguish two types of carry-in: memory and computation in which computation carry-in means a job that has only computation phase, and memory carry-in means a job that has both memory and computation phases.

Lemma 2. *Only one task from $hep(k) \cup lp(k)$ can have memory carry-in.*

Proof. Tasks in $lp(k)$ cannot start on the DMA at or after t_o; hence, they can only have one memory carry-in. On the other hand, tasks in $hep(k)$ that have not started its DMA at or before t_o cannot have memory carry-in because they would be pending and the window would be extended. Since we have a single DMA, only one memory phase can be active at any time either from $hep(k)$ or $lp(k)$. □

We illustrate in Figure 5.3 the worst case carry-in situation for our scheduling algorithm in which t_o aligns with a beginning of a memory phase and $2m - 1$ partitions have been loaded beforehand. We note that a memory phase of a task in $lp(k)$ has to start one time unit before t_o in order to have a memory carry-in. Since memory carry-in is always greater than computation carry-in, we know based on Lemma 1 and Lemma 2 that the worst case situation is to have $2m - 1$ tasks with computation carry-in and one task with memory carry-in.

Bounding Jobs for $hep(k)$

In Figure 5.4, we show the worst case activation for a task in $hep(k)$ with three different scenarios. We note that since unload phases are determined at run-time by the scheduler,

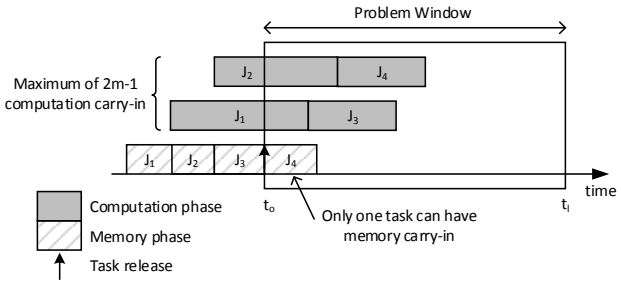


Figure 5.3: Carry-in limit for m=2.

we only consider load and computation phases when bounding the interfering jobs of each task, and we use $e_i = load_i + x_i$ to denote the length of each job. We can then safely assume each computation phase will introduce an unload phase. We construct memory phases out of these load and unload phases in Section 5.3.3. In addition, the unload phases of the first $2m$ jobs to be executed inside the problem window are carried-in from outside the window. Thus, we need to include the largest $2m$ unload phases of tasks in $hep(k) \cup lp(k)$ to account for these unload phases.

For the case where there is no carry-in, the worst activation is to release the task at t_o. On the other hand, for the case where there is a carry-in job, the worst activation is to execute the carry-in job as late as possible such that its computation phase aligns with t_o for tasks with computation carry-in, and the memory phase aligns with t_o for tasks with memory carry-in. We observe that the amount of memory and computation are always greater for a task with memory carry-in than the same task with no carry-in. However, the amount of memory and computation can be less for a task with computation carry-in than the same task with no carry-in. To understand how this could happen, see Figure 5.4. The computation carry-in (the middle one) is obtained by introducing a computation phase of a job executed just before the deadline. This introduced computation pushed out a memory phase from the other end. Even though the computation phase has increased, the memory phase has decreased. Unfortunately, this observation complicates the analysis because it is very hard to determine which case will lead to the worst interference because tasks interact differently as we show in Section 5.3.3. As a result, we will consider the task with no carry-in plus an extra computation phase to account for tasks with computation carry-in since a task with carry-in can have at most one extra job compared to no carry-in case.

We note that the execution of jobs after the end of the problem window has no effect to the analysis; hence, we use min below to only account for jobs within the window. Now,

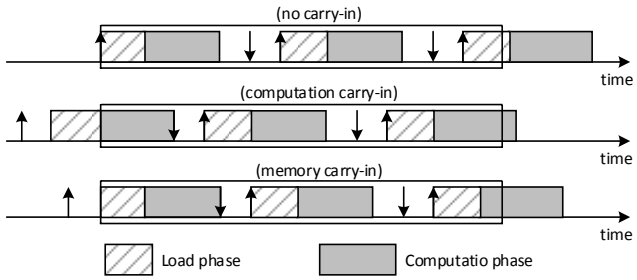


Figure 5.4: The interfering jobs of tasks $\in hep(k)$ with no carry-in, computation carry-in and memory carry-in.

we bound the interfering jobs for a task with no carry-in inside a problem window of J_k as:

- $\lfloor \frac{L_k}{p_i} \rfloor$ body jobs of size e_i.
- The size $\min(e_i, L_k \bmod p_i)$ of one carry-out job.

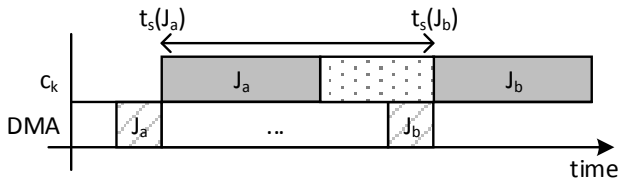
Similarly, we compute the interfering jobs for a task with memory carry-in as:

- $\lfloor \frac{L_k+(D_i-e_i)}{p_i} \rfloor$ body jobs of size e_i.
- The size $\min(e_i, L_k + (D_i - e_i) \bmod p_i)$ of one carry-out job.

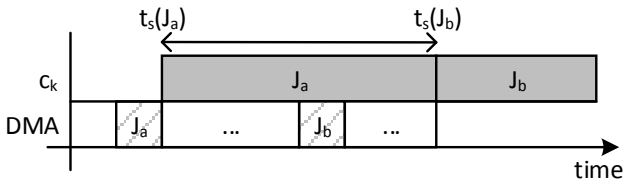
The body jobs are then split into load, computation and unload phases. In contrast, carry-out jobs may include only a load phase, see Figure 5.4 (the top one). Similarly, the computation carry-in jobs include only computation and unload phases.

In summary, we populate X, LD and UD to include computation, load and unloaded phases, respectively, from:

- One task in $hep(k) \cup lp(k)$ with memory carry-in.
- $|hep(k)| - 1$ tasks in $hep(k)$ with no carry-in.
- An extra $2m-1$ largest computation phases computed as $\min(L_k, x_i)$ for $\tau_i \in hep(k) \cup lp(k)$.



(A) A scheduling interval with holes.



(B) A scheduling interval with no holes.

Figure 5.5: Scheduling interval examples.

Since it is difficult to choose which task with memory carry-in will lead to the worst case as tasks have different ratios of computation and memory, we propose to re-calculate the total interference n times (for each task with memory carry-in) and then take the maximum interference. Finally, we need to include the load phase of the problem job as we only consider the computation phases in our analysis.

5.3.2 Bounding the Individual Workload $I_k(J_i)$

We capture the workload of a job on any core, including both computation and holes, by introducing the concept of a *scheduling interval*.

Definition 3 (Scheduling Interval)**.** Assume J_a is running inside the problem window of J_k and $J_a \rightarrow J_b$. We call $[t_s(J_a), t_s(J_b))$ the *scheduling interval* for J_a, and we let $I_k(J_a)$ to denote its length.

Figure 5.5 shows two examples of scheduling intervals. In (a), the scheduling interval of J_a contains a hole because the memory phase of J_b is delayed by memory phases from other tasks and therefore $t_s(J_b)$ is also delayed. In contrast, the scheduling interval of J_a in (b) is followed immediately by the scheduling interval of J_b with no holes because the memory

phase of J_b has finished (completely overlapped) within the scheduling interval of J_a. As we can see, the hole size for each scheduling interval is variable and is dependent on the execution ordering of other tasks. A key idea behind our proof scheme is that we ignore the relative ordering of memory and computation phases within the problem window. Instead, we create a bound on the length of each scheduling interval by determining the maximum number of memory phases that can execute within a scheduling interval, a concept we call a *memory sequence*. We will show how to create an ordering of memory and computation phases that maximizes the total length of scheduling intervals in Section 5.3.3.

Definition 4 (Memory Sequence)**.** A *memory sequence* is the consecutive execution of any m memory phases on the DMA. The length ρ of the memory sequence is the sum of the length of the m memory phases.

The following two lemmas will help us in proving Lemma 5 and Theorem 1.

Lemma 3. *At $t_s(J_a)$, there is always a free partition in $proc(J_a)$.*

Proof. Let t_f be the finish time of the memory phase of J_a, and assume J_a is loaded into $\$_{1,k}$ where $k = proc(J_a)$. Since we assume non-preemptive memory phases, we have the following two cases at time t_f. (1) The partition $\$_{2,k}$ is free. In this case, $t_s(J_a) = t_f$ because there is no job executing out of $\$_{2,k}$. (2) $\$_{2,k}$ is already loaded before the memory phase of J_a, i.e., both partitions are full at t_f. As we assume a non-preemptive computation phases, the computation phase of $\$_{2,k}$ should end at $t_s(J_a)$. The end of a computation phase indicates a free partition which concludes the proof. □

Lemma 4. *Assume $J_a \to J_b \to J_c$. Let t_f be the end time of the memory phase of J_b and t_s be the start time of the memory phase of J_c. Then, the computation phase of J_b must start in the interval $[t_f, t_s]$.*

Proof. We consider two cases. (1) If the computation phase of J_a finishes by t_f, then J_b starts immediately at t_f because its memory phase is already loaded. (2) If J_a is still running at t_f, it follows that both partitions must be full at t_f as J_b is also loaded. Since the memory phase of J_c starts at t_s, at least one partition must be freed by t_s. Hence, the computation phase of J_a must finish by t_s and J_b immediately starts because its memory phase is already loaded at t_f. In either case, J_b starts computing in $[t_f, t_s]$, proving the lemma. □

The following Lemma explains why we design our scheduler to select a core with earlier start time rather than finish time. It basically gives us the guarantee in which our analysis relies on.

Lemma 5. *Our scheduler will never schedule more than m memory phases inside any scheduling interval with holes.*

Proof. It suffices to proof that no two memory phases targeting a same core can execute inside any scheduling interval with holes; this implies that the maximum number of memory phases inside any scheduling interval with holes is m.

Let $J_a \rightarrow J_b$ and they run on c_l. Consider another core c_p and assume by contradiction that two memory phases targeting c_p are executed inside the scheduling interval $[t_s(J_a), t_s(J_b))$ in which the memory phase of J_b executed last as we assume a scheduling interval with hole. Let t_f and t_s be the finish and the start time of these two memory phases, respectively. By Lemma 4, a computation phase must have started on c_p in the interval $[t_f, t_s]$. The core c_l has a free partition at $t_s(J_a)$ as per Lemma 3, and this partition remains free at t_s because the memory phase of next job J_b is executed last. Since $t_s(J_a) < t_f \leq t_s < t_s(J_b)$, our scheduler at time t_s would target c_l rather than c_p. This creates a contradiction, hence the lemma holds. □

The following theorem states the bound of each scheduling interval.

Theorem 1. *The length of $I_k(J_a)$, the scheduling interval of J_a in the problem window of J_k, is upper bounded by the maximum between x_a and the length of any memory sequence.*

Proof. Before we start the proof, we note that within the problem window of J_k, there is always at least one pending task. This is by the definition of t_o in the interval $[t_o, r_k)$ and the fact that J_k is pending after r_k. Now, consider two jobs such that $J_a \rightarrow J_b$ and both run on c_l. Since computation phases are run non-preemptively, the next computation job J_b on the same core cannot start before $t_s(J_a) + x_a$. Hence, x_a is a bound to the length of the scheduling interval of J_a. Now, let us assume that $t_s(J_b)$ is strictly greater than the finishing time of J_a, i.e., there is a schedule hole between J_a and J_b. Since $t_s(J_b)$ is by definition the earliest time that the next job on c_l can start computing, and c_l is idle immediately before $t_s(J_b)$, it holds that the memory phase of J_b must finish exactly at $t_s(J_b)$. From Lemma 3, we know that c_l must have a free partition starting at $t_s(J_a)$. Hence, it follows that the memory phase of J_b would be started at time $t_s(J_a)$, unless the DMA is continuously busy executing other memory phases. In this case, there must be a continuous memory phases executing on the DMA in the interval $[t_s(J_a), t_s(J_b))$ and we know from Lemma 5 that at most m memory phases, a memory sequence, can execute inside any scheduling interval with holes which conclude the proof. □

As an example, consider again Figure 5.5. It is easy to see that $I_k(J_a)$ in (b) is equal to the computation phase x_a while $I_k(J_a)$ in (a) is equal to the memory sequence.

5.3.3 Bounding the Total Workload $I_k(\Gamma)$

In the previous section, we showed how to bound $I_k(J_i)$, the workload of an individual job. However, we only characterized the workload as the maximum between a computation phase and a memory sequence as in Theorem 1. It is clear that $I_k(J_i)$ depends on the computation phase of J_i and a possible hole induced by a memory sequence from other jobs. Therefore, $I_k(J_i)$ should be considered globally to determine a safe bound on $I_k(\Gamma)$.

For a given set of computation phases and memory sequences, we can derive a bound on $I_k(\Gamma)$ as in the following lemma.

Lemma 6. *After sorting computation phases $X = \{\theta_1, \ldots, \theta_{|X|}\}$ such that $\theta_i \leq \theta_{i+1}$ and memory sequences such that $\rho_i \geq \rho_{i+1}$, the following is a valid bound on $I_k(\Gamma)$:*

$$\sum_{i=1}^{|X|} \max(\theta_i, \rho_i).$$

Proof. Based on Theorem 1, the length of each scheduling interval is upper bounded by the length of the corresponding computation phase or a memory sequence; hence, we take $\max(\theta_i, \rho_i)$.

By contradiction, assume that there exists a hypothetical configuration of pairs different than the one in the hypothesis that leads to a strictly higher bound. Since computation and memory sequence lengths are ordered in opposite directions in the hypothesis, it follows that in the hypothetical configuration there must instead exist two pairs (θ_s, ρ_s) and (θ_l, ρ_l), such that $\theta_s < \theta_l$ and $\rho_s < \rho_l$ (i.e., ordered in the same direction). We now show that "swapping" θ_s with θ_l leads to a new configuration with a $I_k(\Gamma)$ bound no less than the previous one. Since we can always obtain the configuration in the hypothesis with a finite number of such "swaps" (for example, using bubble sort), this creates a contradiction.

We let $a = \max(\theta_s, \rho_s)$ and $b = \max(\theta_l, \rho_l)$, and the bound is $a + b$. We have four cases after swapping θ_s with θ_l based on the two terms a and b and given that $\theta_s < \theta_l$: (1) both remain the same. (2) a increases and b remains the same. (3) a remains the same and b decreases. (4) a increases and b decreases. Clearly, (1) and (2) will not decrease the bound. Thus, we only need to consider (3) and (4).

Case(3): To satisfy this case, we should have $\rho_s > \theta_l$ (a remains the same) and $\rho_l < \theta_l$ (b decreases), but we obtain $\rho_s > \rho_l$ which is a contradiction.

Case(4): This case is more elaborate than the previous one. In order to satisfy this case, we should have $\theta_l > \rho_s$ (a increases) and $\theta_l > \rho_l$ (b decreases). Now, we consider two

sub-cases. (i) $\theta_s \geq \rho_s$: based on these assumptions, the bound before the exchange is $\theta_s + \theta_l$ and after the exchange is $\theta_l + \max(\theta_s, \rho_l)$ which is larger or equal. (ii) $\theta_s < \rho_s$: the bound before the exchange is $\rho_s + \theta_l$ and after the exchange is $\theta_l + \max(\theta_s, \rho_l) = \theta_l + \rho_l$, which is larger. After we examine all possible cases, we can conclude that our initial argument is true. □

Lemma 6 assumes a given set of memory sequences. However, we only have from interfering jobs a set of load LD and unload UD phases. As per Definition 4, a memory sequence contains m memory phases. Thus, we first discuss how to construct memory phases out of LD and UD. Then, we show how to construct memory sequences out of these memory phases.

Since the scheduler dynamically combines unload and load phases into one non-preemptive memory phase, the merged unload phase is generally unknown. We could safely assume for each job that its load phase is merged with the longest unload phase in the system, but this is highly pessimistic. Therefore, we instead propose to construct memory phases as in the following lemma.

Lemma 7. *The set of memory phases $\Psi = \{\mu_1, \ldots, \mu_{|\Psi|}\}$ can be constructed by combining the longest load phases from LD with longest unload phases from UD to increase holes as much as possible.*

Proof. Based on the ordering used in Lemma 6, there must exist a worst case configuration for some value $1 \leq j \leq |X|$, where in the first j pairs the length of memory sequences is larger or equal, and in the remaining $|X| - j$ pairs the length of the computation phases is larger or equal. Hence, the bound on $I_k(\Gamma)$ can be obtained by summing the largest j memory sequences and the largest $|X| - j$ computation phases. By adding largest unload phases to largest load phases, we maximize the first j memory sequences. As a consequence, $I_k(\Gamma)$ is also maximized. □

The longest memory sequence is clearly $\rho_{max} = \text{top}(\Psi, m)$ as each memory sequence contains m memory phases. We could again safely assume that all computation phases overlap with the longest memory sequence. However, the following lemma adds a constraint that improves the bound on $I_k(\Gamma)$.

Lemma 8. *Let $J_a \to J_b$ and they both run on c_l. In addition, assume $I_k(J_a)$ and $I_k(J_b)$ are two scheduling intervals with holes. Then, any memory phase executing in the problem window can contribute to either $I_k(J_a)$ or $I_k(J_b)$.*

Proof. Since the scheduling interval $[t_s(J_a), t_s(J_b))$ of J_a contains holes, i.e., it equals to a memory sequence, a memory phase targeting core c_l must finish exactly at $t_s(J_b)$. Hence, such memory phase and all previous memory phases can contribute only to $I_k(J_a)$. On the other hand, all following memory phases can contribute only to $I_k(J_b)$. □

Based on Lemma 8, each memory phase can contribute to at most one memory sequence on each core, i.e., m memory sequences in total. Therefore, we propose to construct memory sequences ρ_i out of memory phases μ_i as follows: $\rho_i = m \times \mu_i$. This guarantees that each memory phase appears at most m times in $I_k(\Gamma)$. Furthermore, following the proof of Lemma 7, by combining the largest memory sequences together, we maximize the the sum of the first j memory sequences as computed in Lemma 6; hence, the bound on $I_k(\Gamma)$ is also maximized.

5.3.4 Bounding the Interference on a Problem Job

In traditional global scheduling, the interference on the problem job J_k can be bounded by $I_k(\Gamma)/m$ assuming the scheduler is work-conserving. In our system however, J_k can be scheduled on a core with later time even though there is an earlier time available on another core, see how J_3 is scheduled in Figure 5.2. The following lemma further characterize the interference on J_k.

Lemma 9. *Let $J_a \rightarrow J_k$ where J_k is the problem job. In addition, let J_b be the last scheduled job on any core such that $proc(J_b) \neq proc(J_k)$. Then, it must hold that $t_s(J_a) \leq t_s(J_b)$.*

Proof. Let t be the time at which `DMA-Dispatcher` is invoked to schedule J_k. In order for J_k to be scheduled on $proc(J_a)$, $s_{proc(J_a)}$ has to be at time t the minimum (or at least equal since we assume that ties are broken arbitrarily) among all cores that have a free partition according to our scheduler rules. In other words, all other cores should have scheduled jobs with start times greater than or equal to $s_{proc(J_a)}$ before J_k can be scheduled on $proc(J_a)$. □

We let $I_k^{max}(I_k^{min})$ be an upper bound on the maximum (respectively, lower bound on the minimum) length of any scheduling intervals in the problem window of J_k, and $I_k^{diff} = I_k^{max} - I_k^{min}$. In Figure 5.6, we show the worst case pattern in which each core runs a sequence of scheduling intervals, and J_k is scheduled after the maximum scheduling interval. Based on Lemma 9, the maximum scheduling interval can finish at most I_k^{diff} time units after the earliest finishing time of last scheduling interval on any other core.

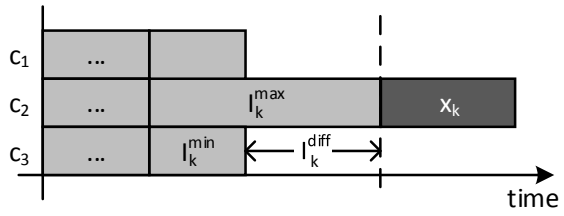


Figure 5.6: The computation phase of the problem job executes after I_k^{max}.

The sum of all scheduling intervals on all cores is upper bounded by $I_k(\Gamma)$, and a bound on the earliest finishing time of other cores can be derived as:

$$\frac{I_k(\Gamma) - I_k^{diff}}{m}. \tag{5.1}$$

By adding I_k^{diff} as in Figure 5.6 and rephrasing the terms, we obtain the upper bound on δ_k, the interference on J_k, as:

$$\frac{I_k(\Gamma)}{m} + (\frac{m-1}{m})I_k^{diff}. \tag{5.2}$$

We compute I_k^{max} as $\max(\rho_{max}, \theta_\alpha)$ and I_k^{min} as θ_1. We note that tasks execute non-preemptively on each core; hence, the length of each scheduling interval must be at least equal to the length of a computation phase, and taking the shortest computation phase within the problem window constitutes a safe lower bound on I_k^{min}. We note that decreasing I_k^{min} could render a task unschedulable. Hence, we can assume for *sustainability* tasks idle until their worst-case execution time; otherwise, we consider a lower bound of 0 on I_k^{min}. In the evaluation, we assumed the former case.

5.3.5 Schedulability Condition

Since the bound on $I_k(\Gamma)$ requires the knowledge of all interfering jobs, it follows that the bound on interference of the problem job derived in Equation 5.2 depends on the size of the problem window. We note that even though the extended problem window that starts at t_o has the advantage of limiting the amount of carry-in, finding t_o is of pseudo-polynomial time complexity, given that the total utilization is strictly less than m [30]. However, the authors of [68] observed that choosing a window of length L_k and starting at t_o is sufficient for the schedulability analysis. Since $t_o \leq r_k$, the length of $[t_o, t_l] \geq [r_k, t_l]$. Intuitively, computing bounds on the amount of work inside a small time interval is tighter than a

large interval as the amount of work gets amortized over larger intervals [30]. Based on such intuition, we prove our schedulability condition as follows.

Lemma 10. *If J_k misses its deadline, then $\delta_k \geq L_k$.*

Proof. Assume J_k misses its deadline and $\delta_k < L_k$. Since we assume a work-conserving scheduler, the time interval $[r_k, t_l]$ has to be busy in order for J_k to miss its deadline. Otherwise, J_k could have executed and finished before the deadline. In addition, $[t_o, r_k)$ has to be busy as per Definition 2. As a result, the time interval $[t_o, t_l]$ has to be busy for J_k to miss its deadline. Equation 5.2 gives a bound on δ_k, the interval of time where all cores are busy including both computation and holes. Since $\delta_k < L_k$, there must exist a time in $[t_o, t_l]$ which is not busy. This creates a contradiction; hence, the lemma follows. □

Theorem 2. *If $\forall \tau_k \in \Gamma : \delta_k < L_k$ then the system is schedulable.*

Proof. It follows from the contrapositive of Lemma 10 that J_k will meet its deadline if $\delta_k < L_k$. If this hold for all tasks in Γ, the system is schedulable. □

5.4 Evaluation

Due to the complexity in estimating the cache preemption and migration delay (CPMD) [43, 34], we compare our scheduling algorithm with contention-based global non-preemptive fixed-priority scheduling [69].

In the contention-based system, we assume that cores access main memory in round-robin order, with no overhead for arbitrating to access main memory. Therefore, if all cores simultaneously contend for access main memory, each core receives $\frac{1}{m}$ from the total bandwidth of the memory subsystem. In our system, we load tasks before execution. Therefore, tasks have no memory stalls during execution. For the sake of a fair comparison, we favor the contention-based system by assuming ideal caches that are managed to have no conflict or capacity misses. Furthermore, since we utilize a DMA component in our system which is an extra hardware resource, we compare against a contention-based system with the same number of cores and with one extra core. In this case, we favor the contention-based schedule as a DMA peripheral is much smaller than an extra core.

We first intuitively show how our system is able to utilize the CPU cores better than the contention-based system, then we compare the schedulability for both systems based on simulations. Assume a system loaded with a large number of tasks. Each task takes four

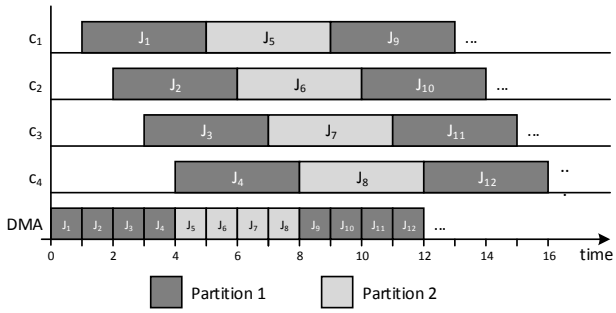


Figure 5.7: Our schedule on 4 cores showing 100% utilization.

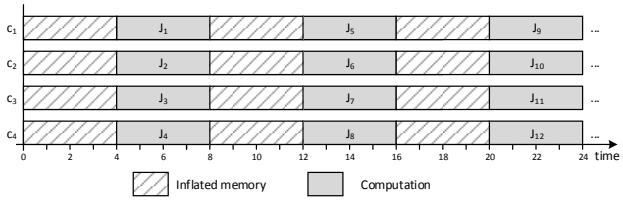


Figure 5.8: Contention-based schedule on 4 cores showing 50% utilization.

time units to compute on a core and requires one time unit for the DMA to unload/load into a local partition. Our approach is able to utilize the cores 100% by overlapping tasks' execution over the cores as long as the DMA operations are small compared to cores execution, this is shown in Figure 5.7. On the other hand, in the contention-based system, cores utilization is affected by the portion of time that a core needs to load a task. In the worst case, all cores could access main memory at the same time. In this case, each core takes $1 \times m$ time units to load the required data instead of one time unit. In particular, with $m = 4$, each core takes 1×4 time units to load a task. As shown in Figure 5.8, the cores utilization is $\frac{computation}{computation+memory} = \frac{4 \times m}{(4 \times m)+(4 \times m)} = 50\%$. Based on this intuition, our approach should perform significantly better. However, due to the requirement to schedule both memory and computation, our schedule exhibits more priority inversion than the contention-based system, which can worsen the Worst Case Response Time (WCRT) for high priority tasks.

The Evaluation of our system is performed on an FPGA platform similar to the one discussed in 4.1. Here, we report some of the important differences. The hardware platform is based on Altera's Cyclone II FPGA instead of Xilinx. The platform uses the Nios-II/f soft-core processor [3] with instruction and data SPMs of 16-KB each. The platform provides 64MB of off-chip SDRAM as main memory, running at 100 MHz, and a standard

DMA core. The operating frequency of the Nios-II processor was 100 MHz.

We evaluate our system on a set of real benchmarks that are executed and measured on the platform in order to obtain data towards our schedulability analysis. The selected real benchmarks are from existing embedded benchmark suites. The selection is aimed to represent several applications used in the embedded real-time domain. Memory intensive applications are chosen to better stress the platform's memory subsystem. Three benchmarks from the well-known automotive EEMBC benchmark suite [134] are selected, a2time (angle to time conversion), canrdr (response to remote CAN request), and rspeed (road speed calculation). Two benchmarks from the DIS (Data Intensive System) benchmark suite [118] are selected, transitive and corner-turn. We limited each benchmark to either run up to 1 ms, or by its size so that it does not consume more than half of a SPM size (the size of a partition). Table 4.4 reports, for each benchmark, the size of code and data in the local memory, the time taken to run out of the local memory (scratchpad) and the DMA load and unload times. All times are reported in cycles (cyc) and the sizes are reported in bytes (B).

Table 5.1: Benchmarks

Benchmark	**Code size(B)**	**Data size(B)**	**SPM(cyc)**	**load(cyc)**	**unload(cyc)**
a2time	3108	5420	100497	4560	2578
rspeed	1956	6864	55688	4635	2950
canrdr	2724	8030	47280	5134	3251
corner-turn	2032	8192	16728	4996	3292
transitive	2080	3024	102898	3677	1960

The applications in Table 5.1 are used to generate sets of random tasks. The task sets are generated similarly to Section 4.1.4 and Section 4.2.3. Note that the utilization in % reported in the figures below is the total CPU utilization normalized by the number of cores m. Finally, each schedulability graph in this section is made of 4,000 experiments.

Figures 5.9-5.11 show results in term of ratio of schedulable task sets. As shown in Figure 5.9, in the case of dual-core system, our bound is better than the contention-based bound with equivalent number of cores (m). However, the contention-based system with one extra core ($m + 1$) was able to schedule all generated task sets. The reason for this is that the generation of task sets targeted an m-core system and then scheduled on an $m + 1$-core system. In addition, when targeting a small number of cores, the generated tasks set is relatively small. As a result, tasks execution takes advantage of the extra core so they finish sooner.

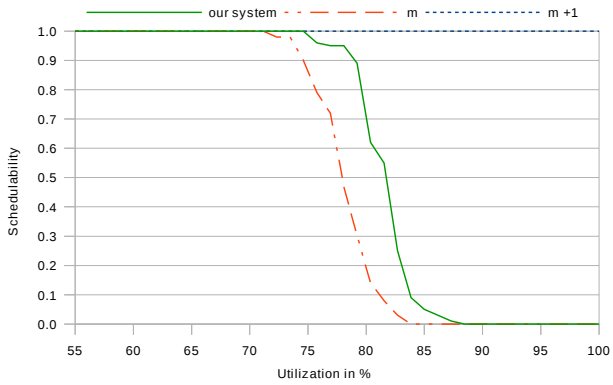


Figure 5.9: 2-cores schedulability comparison.

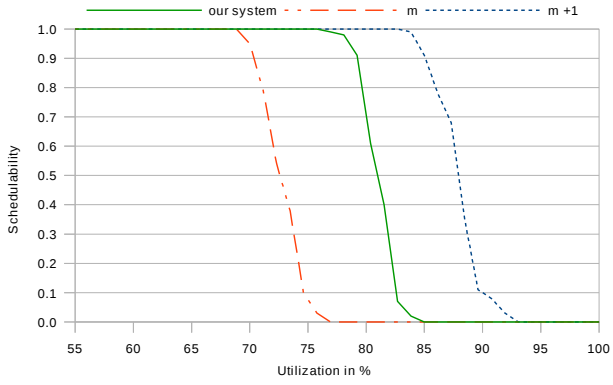


Figure 5.10: 4-cores schedulability comparison.

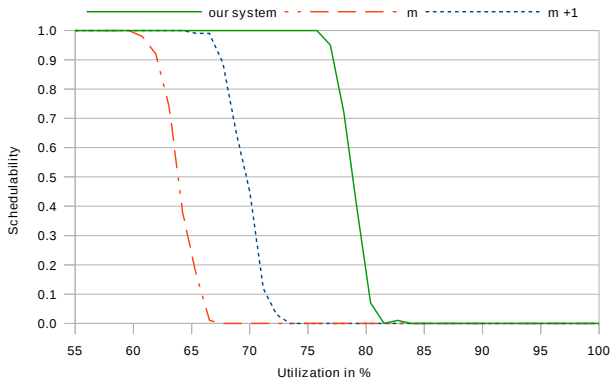


Figure 5.11: 8-cores schedulability comparison.

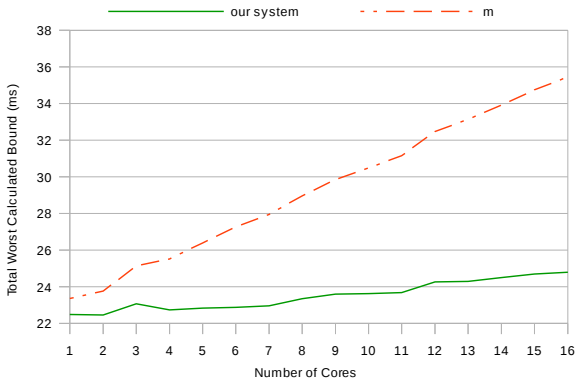


Figure 5.12: The scalability comparison of the WCRT bound.

In Figure 5.10, we compare the four-core case. As shown in the figure, we still perform worse than $m + 1$ cores. This is mainly due to the pessimistic bound in our approach as discussed in Section 7.2. However, the schedulability of the contention-based systems dropped significantly. For example, the $m + 1$ system dropped from 100% to around 89%.

As we increase the number of cores, as in Figure 5.11 for an eight-core system, the memory phases in the contention-based system becomes longer and affects the schedulability. In addition, the generated tasks set is larger than before. Therefore, the advantage of one extra core becomes less effective as access to main memory becomes a bottleneck. As you can notice, our approach is much less affected by the increase in number of cores due to advantage of overlapping DMA operations with execution on the cores.

Figure 5.12 compares the WCRT obtained through the analysis on both systems for the lowest-priority task as the number of cores increases. It is clear that the contention-based bound rises much quicker with the number of cores compared to our bound. This mainly because as the number of cores increases the contention on main memory, in the contention-base system, becomes a bottleneck. The figure also explains why we do even better than $m + 1$ cores with higher number of cores.

5.5 Summary

In this chapter, we extended the 3-phase task scheduling model introduced in Chapter 3 to handle global, rather than partitioned scheduling. We again consider a set of sporadic real-time tasks scheduled non-preemptively according to fixed-priorities, and we make similar hardware assumptions: each core is provided with a dual-ported SPM, divided in two partitions, so that we can execute a task out of one partition while reloading the other one.

Compared to the partitioned case, a key difference in the global case is that we assume knowledge of all tasks in the system. Hence, we do not need to enforce a fair DMA arbitration among cores either in software or hardware. Instead, we can directly schedule tasks' accesses to main memory through a global queue of DMA operations; effectively, this implies that core execution is driven by DMA scheduling, rather than vice-versa.

We then derive a schedulability analysis based on the well-known technique of the problem window. The key intuitions behind the analysis are: 1) we can still construct the interference on the task under analysis by considering the workload of tasks executing on the m cores in the system; however, to account for the overhead of DMA operations, we need to add extra workload in the form of scheduling holes, where the cores are stalled waiting for the completion of DMA operations; 2) in the worst case where holes are present, the DMA schedule is equivalent to a round-robin among the m cores. Hence, we can bound the length of holes by considering the maximum length of m consecutive task unloads/loads.

Finally, we compared our technique against standard global scheduling, where tasks are preemptive but contend for main memory, using a modified version of the platform introduced in Section 4.1. In particular, we show that as the number of cores increases, our solution provides higher schedulability even compared to a contention-based system with $m + 1$ cores (to account for the hardware overhead of the DMA engine).

Part II

Task Communication For Hard Real-time Applications

Chapter 6

Inter-Task Communication with 3-Phase Task Model

In this chapter, we discuss how the proposed 3-phase execution model can be extended to support asynchronous communication between tasks. In our model, the communication data is written to main memory during the DMA unload phase of a sender task, and then read back from main memory during the DMA load phase of a receiver task. Section 6.1 discusses the communication model in more details, while Section 6.2 presents the implementation requirements. Section 6.3 shows how to bound the communication latency for the partitioned scheduling scheme with fixed-size DMA operations presented in Section 3.3, and finally Section 6.4 presents evaluation results.

6.1 The Proposed Inter-Task Communication Model

We assume an asynchronous communication model that regulates data exchange between tasks, whether they run on the same core or on different cores. In this asynchronous model the previously produced data, by a sender task, can be overwritten if it was not read by the receiver task before. In detail, a sender task's communication data is written to main memory during its unload phase. Similarly, any communication data required by a receiver task is loaded into the SPM during its load phase. Therefore, this model does not require inter-core communication in our proposed SPM-centric system: the communication is performed via main memory during the load and unload phases of the tasks. This model of communication can be applied to both sequential and parallel tasks, since the

communication is scheduled and performed by the DMA during the load/unload of the tasks.

This asynchronous communication model is more suitable to state-based communication. Unlike the event-based communication in which the receiver task might receive a sequence of massages, in state-based communication the receiver task is only interested in the latest up-to-date data or state. Packets arriving on a network card is an example of event-based data, while state ports of a temperature sensor or proximity sensor are examples of state-based data.

We consider a DAG of communicating tasks, from which we extract every possible chain of communicating tasks. Each chain, λ_k is a set of n communicating tasks. Thus, we denote the chain λ_k as $\lambda_k = \{\tau_1, \tau_2, ..., \tau_n\}$, which we call a flow. A task τ_i in the flow λ_k receives data from τ_{i-1} and sends data to τ_{i+1}. In other words, the communication data in the flow passes through all the tasks in the flow in sequence; We assume that task-local data, code, and communication buffers can be accommodated within one partition of the scratchpad memory.

6.2 Implementation

To explain how the communication model works, consider for example two tasks, τ_1 and τ_2, that need to send data to a task τ_3. Based on the statically defined communication graph, we need to assign communication buffers to each communicating task. For instance, consider a communicating task τ_1. We need to allocate one send buffer and one receive buffer for each task τ_1 is sending to or receiving from respectively. In this example, for τ_1 and τ_2, we need to allocate one send buffer each, dedicated for communication with τ_3. On the other hand, τ_3 needs two receive buffers, one to store data from τ_1, and the other to store data from τ_2.

At task definition, τ_1 and τ_2 need to know the size of each message sent to τ_3. This size information is used to create the temporary send buffer inside scratchpad memory of the core where τ_1 and τ_2 will be loaded and executed. On the other hand, the defined receive buffers of τ_3 is in main memory, two in this example.

During execution of sender tasks from the SPM, data to be sent to the receiver tasks are written inside temporary buffers. These buffers are unloaded using the DMA from scratchpad to main memory, and specifically into the corresponding receive buffers of the receiver tasks. Finally, when τ_3 is activated, all of its local code, data, and communication

data received by the time it is scheduled to load, is loaded from main memory into the corresponding core's SPM.

6.3 Bounding Communication Latency

As discussed in the previous sections, tasks communicate asynchronously without any precedence constraint between them. In other words, when a job of a higher-priority task τ_1 is ready, it will be scheduled and start loading as soon as there is a free partition regardless of any other running task that sends data to it. Since the access to main memory is serialized using DMA, integrity of the communication data is assured as there will be no data race condition (lock-less data sharing). Suppose τ_1 is a receiver task for data sent by τ_2. Then τ_1 will access the previous (old) communication data from τ_2 if τ_1 is loaded while τ_2 is still running or not yet unloaded from the SPM partition to main memory.

From a schedulability point of view, this communication model does not affect task scheduling. However, we still need to bound the worst-case end-to-end communication latency. As an example, we show how to determine the communication latency under the same DMA scheduling and response time model used in Section 3.3; as a reminder, this means that we consider fixed-size DMA operations, and the computed response time of a task is the time that elapsed from when a task becomes ready (released) to the time it finishes and is fully unloaded. Therefore, communication data sent by a sender task τ_i will be available to a receiver task after R_i, which is the worst-case response time of τ_i, accounting for interference and overheads.

As mentioned earlier, we consider sets of communicating tasks; each set, which we call a flow λ_k, is a chain of n tasks: $\lambda_k = \{\tau_1, \tau_2, ..., \tau_n\}$, such that a task τ_i in the flow receives data from τ_{i-1} and sends data to τ_{i+1}. In other words, the communication data in the flow passes through all the tasks in the flow in sequence; consequently, the end-to-end communication latency (L^{λ_k}) is the time it takes for the data to be consumed (loaded) by the first task in the flow, i.e, τ_1, until the last task in the flow, i.e., τ_n, writes the data to main memory (unload). The end-to-end latency of a flow λ_k is computed as in Equation 6.1.

$$L^{\lambda_k} = \begin{cases} R_1 + \sum_{i=2}^{n}(T_i - 2 \cdot \sigma + R_i) & \text{Different Cores} \\ R_1 + \sum_{i=2}^{n}(T_i - (2 + m - 1) \cdot \sigma + R_i) & \text{Same Core} \end{cases} \tag{6.1}$$

Theorem 6.1. *The worst-case total end-to-end latency of a communication flow* λ_k *is* $L^{\lambda_k} = R_1 + \sum_{i=2}^{n}(T_i - 2 \cdot \sigma + R_i)$

Proof. As shown in Figure 6.1, the worst-case alignment (critical instant) between any two different jobs that communicate is when the receiving job (τ_2) starts loading right before the unload phase of the producer job (τ_1). This behavior prevents τ_2 from loading the fresh communication data until the next invocation. In addition, in the worst-case, τ_2 starts loading right after its release to maximize latency by minimizing the overlapping region (see Figure 6.1) between τ_2 and τ_1. Note that, in our system, this only happens if the two tasks are on different cores. If they are on the same core, that core has to wait for $(m-1)$ TDMA slots to perform the unload of τ_1, which will increase the overlapping region between the two tasks, thus reducing the latency.

Therefore, to compute the communication latency between τ_1 and τ_2, we add one period and the worst-case response time (as detailed in the previous section) of τ_2, and subtract the overlapping region between τ_1 and τ_2. In the worst-case, the minimum overlapping region is always made of the loading slot of τ_2 and the unloading slot of τ_1 which is $2 \cdot \sigma$. To compute the total end-to-end latency between a chain of tasks in a flow, we simply apply the same method between any two successive tasks in the flow. For the first task in the flow (τ_1), we only consider the response time. □

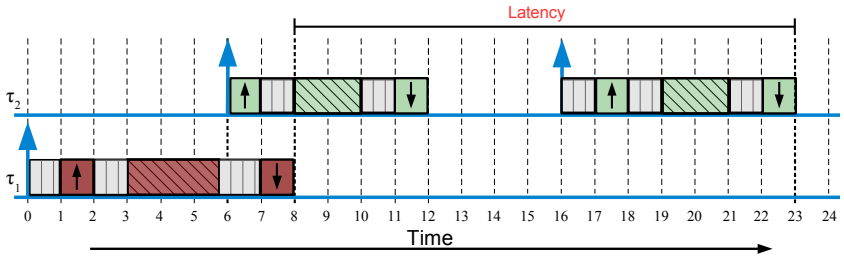


Figure 6.1: Worst-case communication latency between two tasks

.

6.4 Evaluation

We evaluate the communication latency by generating random set of tasks as discussed in Section 4.2.3. From a generated task set, we compute the response time of each task using

the analysis in Section 3.3. After that, we generate four random communication flows with 5, 10, 15, and 20 tasks and compute the communication latencies of each flow. Each synthetic evaluation is repeated 1,000 times and the average worst-case communication latency is reported. Figure 6.2 shows the estimated worst-case communication latencies for the generated flows comprised of a mix of applications provided in Table 4.7. As one can observe, there is a slight improvement when the communication tasks are scheduled on the same core. In this specific evaluation, the system comprises two application cores, as this is the setting of the implemented COTS platform in Section 4.2.

In addition, the yellow line in the figure represents the average communication bandwidth, that is, the total amount of data transferred divided by the end-to-end communication latency. For the sake of this evaluation, we assumed that each task in a flow will send data equal to the size of its data section as reported in Table 4.7; hence, the amount of data transferred between any two successive tasks in a flow might be different, which resemble real-life scenarios. Each point in the line is generated by randomly picking 5, 10, 15, or 20 tasks based on the number of tasks in the flow. Then, we compute the total amount of data transferred withing the end-to-end latency window. We repeat the test 1,000 times and take the average to capture all the application benchmark in Table 4.7.

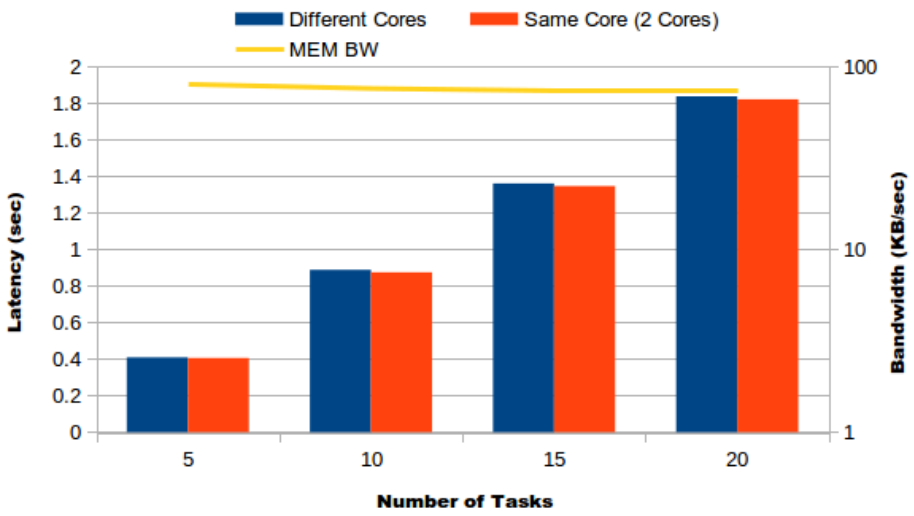


Figure 6.2: End-To-End communication Latency.

6.5 Summary

We showed how to incorporate inter-task communication in the proposed 3-phase task model. We consider an asynchronous communication model, where pairs of sender and receiver tasks exchange data. In particular, communication data is written by the sender task to main memory during its unload phase, and read from main memory by the receiver task during its load phase. Since there are no precedence constraints among communicating tasks, the schedulability analyses presented in previous chapters do not need to be changed.

We computed bounds on the end-to-end latency for a chain of task based on the analysis for partitioned systems in Section 3.3, and evaluated the obtain latency on the platform introduced in Section 4.2.

Chapter 7

Bundled Scheduling of Parallel Real-time Tasks

After discussing inter-task communication in Chapter 6, in this chapter and the next one we focus on intra-task communication for parallel real-time tasks. In particular, in this chapter we focus on how to schedule parallel tasks to simplify synchronization among parallel threads of the same task; while in Chapter 8 we introduce a predictable interconnection design and derive latency bounds for messages exchanged between communicating threads.

With the increased demand for high-performance applications such as autonomous driving and computer vision [158], parallel processing is becoming relevant to the real-time community. However, most related works on scheduling of real-time parallel tasks [142, 89, 92, 109] assume that application threads are scheduled independently. In practice, there is large evidence [123, 60, 77, 150] that parallel threads often needs to be executed concurrently. This is especially true for threads that are tightly synchronized through the use of either shared resources or message passing primitives. Some synchronization primitives, such as intra-threads locks, cannot be modelled by most related work [39, 109]. Many primitives can cause an unnecessary number of context-switches and increase thread execution time due to blocking when threads are not executed together. Furthermore, the need to account for such synchronization mechanisms greatly increases the complexity of the task model.

For instance, consider the synchronizing Thread#1 and Thread#2 in Figure 7.1. If the scheduling policy does not provide guarantees to schedule them in parallel at the same time, their WCET are prone to inflation. In a preemptive policy they might suffer excessive amount of preemptions, while in a non-preemptive policy a thread might spin-wait for

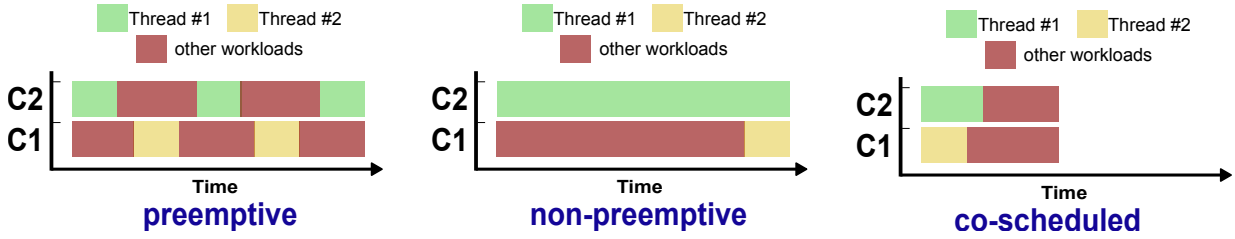


Figure 7.1: Illustrations of the negative impact on synchronized parallel threads if not co-scheduled at the same time

the other thread. Even in the best-case, depending on the synchronization primitives and the scheduling policy, the WCET might not be prone to inflation, but it still complicates accounting for their communication time. Therefore, we argue that parallel threads need to be co-scheduled to reduce the synchronization overheads and to simplify their communication analysis. As will be detailed in Section 8.6, we can compose the total WCET as the WCET on CPUs plus the worst-case communication time between the parallel threads. In particular, all parallel threads of a tasks are scheduled concurrently, hence their communication time can be simply accounted for as long as the inter-core interconnect provides provable real-time bounds.

To address this issue, gang scheduling [123] has been extensively studied in the HPC and general purpose domains [60, 77, 150]. Under gang scheduling, an application is scheduled only if there are sufficient cores to executed all the application's threads in parallel. Gang scheduling of real-time tasks has been investigated in [81, 55, 64, 38, 47]. In particular, [81, 55, 64] consider a *rigid* task model, where in the worst case the number of threads required by an application is assumed to remain constant over its entire execution time. While the rigid model has the benefit of simplicity, it can incur a significant loss of performance by overestimating the computational demand of an application: many parallel applications change their required number of threads during execution. For example, in the common fork-join model [1], the application progresses through a set of phases, where each phase can require a different number of threads. In the similarly common Directed Acyclic Graph (DAG) model, the application comprises a set of precedence-constrained subtasks, and the number of threads used by the application at any one time depends on the subtask scheduling.

Hence, in this chapter we introduce a novel task model, which we call the bundled model, which supports gang scheduling of parallel threads without incurring undue pessimism in

[1]Note that we use the term fork-join to refer to the model that is also known as a multi-threaded task in the literature.

modelling the application's demand. In this model, a real-time task is composed of a sequence of bundles, where each bundle is characterized by a known worst-case execution time (WCET) and number of required cores. All threads within a bundle are then gang scheduled. Our model thus represents a generalization of the traditional real-time gang model, in the sense that a traditional gang task is equivalent to a bundled task with a single bundle.

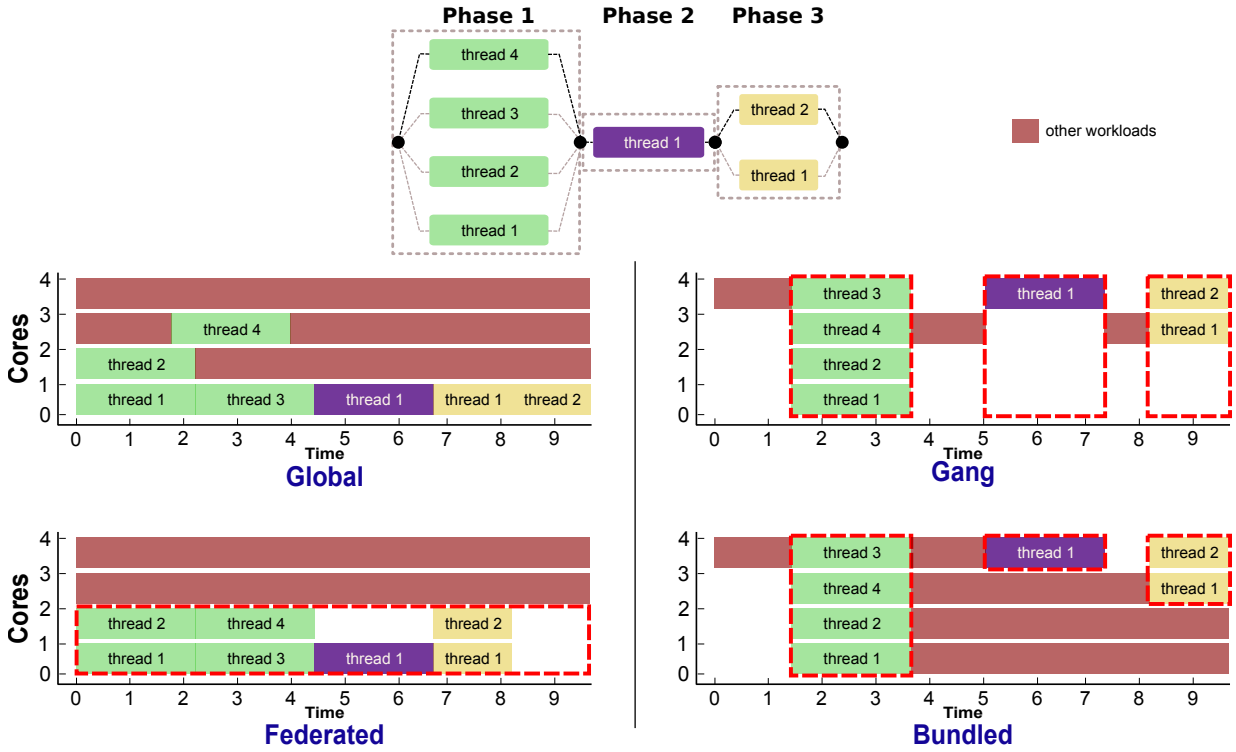


Figure 7.2: Illustrations of how a fork-join parallel task can be scheduled according to different scheduling strategies

Figure 7.2 shows an illustrative example of a parallel fork-join task and how parallel threads might be scheduled with different scheduling strategies. The important point to note here is that our objective is to guarantee that parallel threads are scheduled concurrently 1) to reduce the synchronization overheads and 2) to simplify accounting for their communication. On the left side of the figure, global thread scheduling [109] and federated scheduling [93] are shown. As can be seen, both scheduling schemes do not provide guarantee to schedule the parallel threads in the same phase concurrently. Consequently, these scheduling schemes do not meet our objectives. As illustrated in the figure, federated

scheduling allocates a dedicated cluster of cores for the parallel task based on a metric relevant to the parallel task utilization. However, federated scheduling might suffer from core overprovisioning compared to the greedy global thread scheduling.

On the right side of the figure, gang scheduling of the rigid model [81] and the proposed bundled scheduling are shown. Indeed, gang scheduling meets our objectives. However, as mentioned earlier, gang scheduling of the rigid model suffers significant loss of performance due to core overprovisioning. As illustrated in the figure of the gang case, four cores are reserved for the entire execution time of the task, even when there are times where only one or two cores are actually needed. In the proposed bundled scheduling, we segmented the shown fork-join task into three different bundles based on the number of required cores in each phase. All threads in each bundle are guaranteed to be scheduled in parallel, similar to the gang case. However, since bundles can be independently scheduled, we only reserve the required number of cores in each bundle, which leads to improved CPU utilization. Note that, we improve over the gang scheduling of the rigid model at the cost of a more complex schedulability analysis due to induced complexity to deal with the precedence constraints between bundles.

More in details, we provide the following contributions: (A) We introduce the bundled task model, and discuss how bundles are scheduled on an identical multiprocessor according to a preemptive, fixed priority gang scheme. (B) We show how applications coded according to different programming models can be executed within bundles. (C) We derive a sufficient schedulability analysis for sporadic bundled task sets. (D) We evaluate the performance of the derived analysis and compare it against the rigid gang model, as well as a state-of-the-art analysis for global (non-gang) scheduled parallel tasks [109]. (E) Finally, we discuss how to integrate bundled scheduling with the 3-phase task model to ensure predictable access to memory resources and reduce the overhead of preemptions.

The rest of the chapter is organized as follows. We first introduce the proposed system model in Section 7.1. In Section 7.2, we detail the schedulability analysis for the proposed system. The evaluation results are reported in Section 7.3. In Section 7.4, we discuss how bundled scheduling could be integrated with the 3-phase execution model. Finally, we conclude with some remarks in Section 7.5.

7.1 System Model

We consider an identical multiprocessor platform consisting of m processors; we use the terms processor and core interchangeably. We focus on scheduling n sporadic bundled

parallel tasks denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i comprises a sequence of b_i *bundles* [2], represented as $\tau_i = \{\tau_{i,1}, \tau_{i,2}, ..., \tau_{i,b_i}\}$. Each bundle $\tau_{i,j} \in \tau_i$ is characterized by $\langle h_{i,j}, l_{i,j} \rangle$, where $h_{i,j} \leq m$ is the number of cores required to execute the bundle, and $l_{i,j}$ is the WCET of the bundle. We call $h_{i,j}$ the height of the bundle and $l_{i,j}$ the length of the bundle; we also call $v_i = h_{i,j} \times l_{i,j}$ the volume of the bundle. We denote the total WCET of τ_i as $L_i = \sum_{j=1}^{b_i} l_{i,j}$ and the total volume of τ_i as $V_i = \sum_{j=1}^{b_i} v_{i,j}$. Each task τ_i is further characterized by a period T_i and a relative deadline D_i, with $D_i \leq T_i$ (constrained deadline). τ_i generates a (potentially) infinite sequence of jobs, with arrival times of successive jobs separated by at least T_i time units; when referring to a bundle $\tau_{i,j}$ of τ_i, we also use the term job to denote the instance of $\tau_{i,j}$ executing as part of a job of τ_i.

Tasks are scheduled according to a global, preemptive fixed-priority gang scheduling scheme, as detailed in Section 7.1.1. We say that task τ_i is active at time t if a job of τ_i has arrived before or at t but has not yet finished executing. Since bundles are always processed as a sequence, at any time t when the task is active, exactly one bundle of the task is active; bundle $\tau_{i,1}$ becomes active when a job of τ_i arrives, while each other bundle $\tau_{i,j+1}$ becomes active once the previous bundle $\tau_{i,j}$ finishes. The system scheduler selects which tasks to execute based on the currently active bundles. We say that a task/bundle is ready if it is active but not executing. We consider two priority schemes. In the fixed per-task scheme, each task τ_i is assigned a priority P_i, where higher number implies higher priority. In the fixed per-bundle case, each bundle of τ_i is assigned an individual priority $P_{i,j}$. Note that the former can be seen as a restriction of the latter, where $P_{i,j} = P_i$ for all bundles of τ_i. A task set if schedulable with respect to a priority assignment if all jobs are guaranteed to complete by their deadline. We assume that all time values and task parameters are natural numbers.

Similarly to related work on gang scheduling [81], throughout Sections 7.1-7.3 we do not directly model preemption overheads, but assume that they can be accounted for by extending the length of the task. In the case where concurrent execution of multiple tasks can cause interference on shared resources such as cache or main memory, we assume a resource sharing mechanism with provable delay bounds [106, 126]. Clearly, as discussed throughout this dissertation, such preemption overheads and delay bounds can be significant for memory-intensive tasks. Therefore, in Section 7.4 we remove such assumptions, and instead show how to integrate the proposed bundled scheduling with the 3-phase execution model, assuming the availability of a DMA engine and private per-core SPM.

[2] also referred to as *segments* or *phases* in related task models. We use the different term bundles to stress that they are gang scheduled.

7.1.1 The Scheduling Algorithm

The gang *system scheduler* is conceptually represented in Algorithm 7.1 [3]. The scheduler iterates over each active bundle $\tau_{i,j}$ in decreasing priority order, and executes the bundle if the number of remaining available cores is greater than or equal to its height $h_{i,j}$. Note that the scheduler is work-conserving, in the sense that it always schedules a bundle if there are enough available cores. The scheduler also always reserves $h_{i,j}$ cores for bundle $\tau_{i,j}$ while executing it. Since furthermore the priority of each bundle is constant, it follows that context-switches can only happen when a new bundle becomes active, and the system scheduler is invoked when a job arrives or finishes, and when a bundle finishes and the next one is activated.

We assume that an *application scheduler* uses the reserved $h_{i,j}$ cores to execute the functional components of the parallel application associated with task τ_i; an example of such application scheduler is the OpenMP runtime [49]. If all computation assigned to a bundle finishes early, then the application scheduler can yield to the system scheduler, causing the current bundle to finish; otherwise, the system scheduler executes an active bundle for its assigned duration $l_{i,j}$. In general, the way the application scheduler behaves depends on the programming model of the parallel application; hence, we briefly discuss how to handle different parallel programming models in the next section.

Algorithm 7.1 Bundled Fixed-Priority Scheduling Policy

Let Q_{active} be the set of active bundles.
Let Q_{exec} be the set of executing bundles.
Let h_{rem} be the number of remaining cores.

1: $Q_{exec} \leftarrow \emptyset$;
2: $h_{rem} \leftarrow m$;
3: **for** each $\tau_{i,j} \in Q_{active}$ in decreasing $P_{i,j}$ order **do**
4: **if** $h_{i,j} \leq h_{rem}$ **then**
5: $Q_{exec} \leftarrow Q_{exec} \cup \{\tau_{i,j}\}$;
6: $h_{rem} \leftarrow h_{rem} - h_{i,j}$;
7: execute the bundles in Q_{exec};

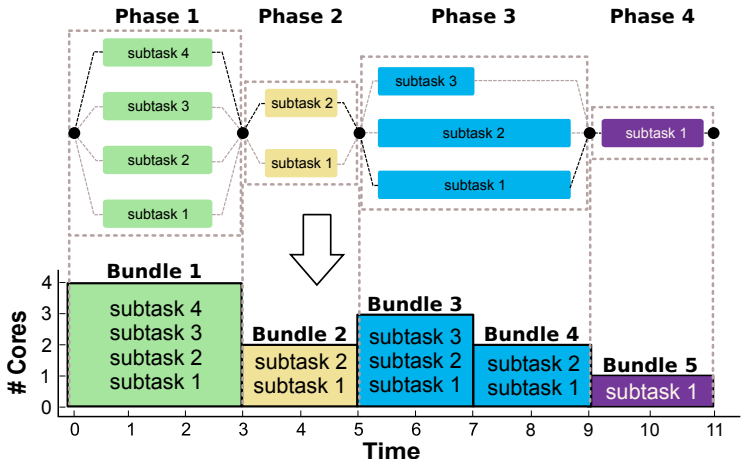


Figure 7.3: Fork-join application and resulting bundled task.

7.1.2 Programming Model

We next show how the proposed bundled model can be used to schedule parallel applications coded according to different programming models. In particular, bundled scheduling is a natural fit for the fork-join model. In the fork-join model, the program is made of a sequence of synchronous phases, as shown in Figure 7.3. Each phase is composed of one or more parallel threads, which are also known as subtasks. To schedule a fork-join application within a bundled task, we allocate one or more bundles per phase. If all subtasks within a phase have the same WCET, then we allocate a single bundle with a height equal to the number of subtasks in the phase, and a length equal to the WCET of the subtasks. If the subtasks have different WCET, then we allocate multiple bundles, one for each WCET value, as shown for Phase 3 in Figure 7.3: the first bundle has length equal to the shortest WCET and includes all subtasks, while successive bundles include only subtasks with higher WCET. In either case, at run-time the application scheduler can yield once all subtasks in the phase have finished, possibly before their WCETs. The key advantage of bundled scheduling the application, compared to independent thread scheduling, is that all subtasks are gang scheduled to minimize the synchronization overhead, thus potentially reducing their WCET estimate.

The bundled model can also be applied to the more general case where the internal structure of the program is unknown. Assume that the logic of the application scheduler

[3]Note that the algorithm could be optimized by using multiple priority queues for active bundles based on their heights.

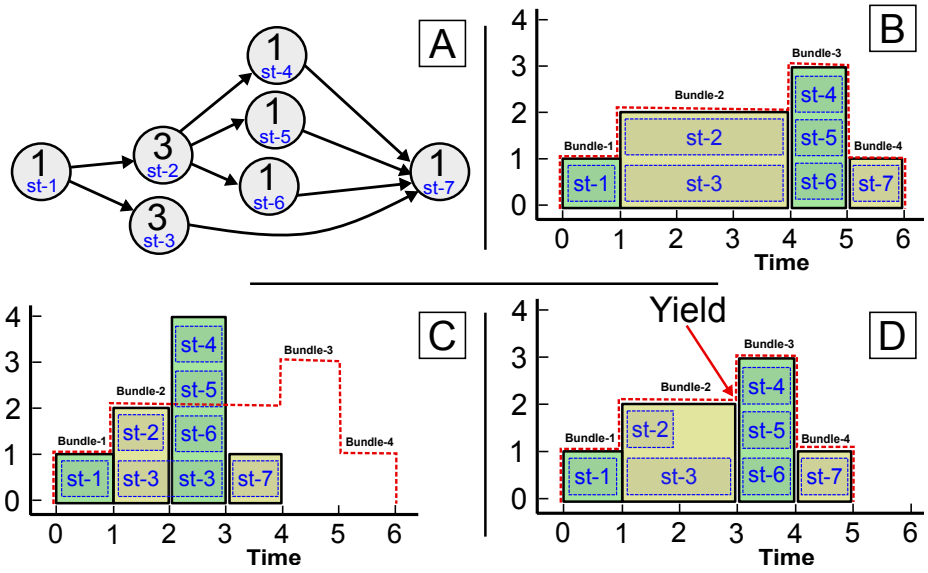


Figure 7.4: DAG application and possible schedules.

does not depend on when bundles are executed, which is decided by the system scheduler. Then we can guarantee that the application safely completes when assigned to a (schedulable) bundled task by following this procedure: by either measurement or analysis, we determine the *execution shape* of the application when running in isolation on m cores, that is, the number of cores used by the application at every point of time. If the execution shape of the application changes between different runs, for example based on its inputs, then for any point of time we consider the largest number of cores used by the application in any possible run (assuming complete coverage). Bundles can then be allocated based on the obtained shape; this guarantees that when the task is scheduled by the systems scheduler, the application scheduler is always provided with a number of cores equal to or higher than the number of cores it can use at that time. While this procedure can lead to overprovisioning CPU resources, it cannot result in a larger reservation compared to the rigid model.

If more information is known about the internal structure of the application, then we can avoid core overprovisioning. For example, consider a DAG application, where the WCET of each subtask is known, as shown in Figure 7.4(A). Figure 7.4(B) shows the execution shape generated by simulating the application on a system with at least 3 cores, based on any work-conserving application scheduler and where each subtask executes for its WCET. As shown in the figure, based on the obtained shape we can generate a task with

4 bundles. However, if subtasks do not execute for their WCET, then the resulting shape might not be safe, in the sense that the application might require more cores at some point in time. For example, in Figure 7.4(C) we show the case where subtask2 executes for 1 unit of time, and subtask3 executes for 2 units of time; this results in the application using 4 cores at time 2. We can solve the issue by employing a time-triggered, non-work conserving application scheduler that activates each subtask based on its start time with respect to a bundle in the original shape; the corresponding schedule is shown in Figure 7.4(D). Note that in this example, the scheduler yields at time 3 once all subtasks in Bundle-2 have finished; subtasks3-5-6 are activated once Bundle-3 starts. Finally, note that the scheduler implementation in [168] already relies on a static table of subtasks, so it could be easily extended to implement the described time-triggered scheme. As an added advantage, the described scheduler can guarantee that specific subtasks are executed concurrently, again minimizing synchronization overheads.

7.2 Schedulability Analysis

In this section, we derive a sufficient schedulability analysis for a bundled task set. We use a similar response time approach as the one in [39, 109], but we specialize it for gang scheduling based on fixed priorities. We derive a response time upper bound R_k for each task under analysis τ_k, and we deem the task set schedulable if for every task it holds $R_k \leq D_k$. For clarity of exposition, we begin by detailing the analysis for the special case of the rigid model, that is each task τ_i has only one bundle. Hence, without loss of generality we use $h_i = h_{i,1}$ to denote the height of the single task-bundle; also note that $l_{i,1} = L_i$. Then in Section 7.2.2 we extend the analysis to tasks with multiple bundles.

As in all related work on global scheduling, the analysis is based on the concept of interference, which is defined as the amount of time during which τ_k is active, but cannot execute because all cores are busy executing other tasks (i.e., the task remains in the ready queue).

Definition 7.1. *The interference I_k on task τ_k is the cumulative time during which some job of τ_k is ready (active but not executing).*

While the definition of interference remains unchanged, note an important difference compared to the case of sequential tasks: in the context of parallel tasks scheduled as a gang, an active task can remain ready even when some cores are idle, due to the number h_k of cores required to run the gang.

Theorem 7.1. *Let $\bar{I}_k(t)$ be a monotonic [4] upper bound on the interference on τ_k in any window of time of length $t \leq D_k$ starting with the arrival of a job of τ_k. Then for any work-conserving scheduling algorithm, an upper bound R_k to the response time of τ_k can be found with the following fixed-point iteration, starting with $R_k = L_k$:*

$$R_k \leftarrow L_k + \bar{I}_k(R_k). \tag{7.1}$$

Proof. Since $\bar{I}_k(t)$ is monotonic and due to the natural number time convention, it follows that the iteration either converges to a fixed point $R_k = L_k + \bar{I}_k(R_k) \leq D_k$ in a finite number of steps, or it becomes $R_k > D_k$ in a finite number of steps, in which case we fail to prove the task schedulable. It remains to show that if $R_k \leq D_k$ is a fixed point, then it is an upper bound to the response time of τ_k.

Consider a window of time of length R_k starting with the arrival of any job of τ_k. Since we assume constrained deadlines, it follows that only one job of τ_k can be active in the window, and that job remains active from the beginning of the window until it either finishes executing for L_k time or the window is over. Given that we know that the amount of time that τ_k is ready is no more than $\bar{I}_k(R_k)$, it follows that the amount of time that τ_k can execute in the window is at least $R_k - \bar{I}_k(R_k) = L_k$. Hence, since the schedule is work-conserving, the job must finish executing in the time window, meaning that R_k is indeed an upper bound to the response time of any job of τ_k.

□

Theorem 7.1 is the same as 6 in [39] and 5.1 in [109]. The monotonicity condition is required to ensure that the iteration terminates in a finite number of steps by either finding a response time upper bound or obtaining $R_k > D_k$, in which case we fail to prove the task schedulable.

The challenge is then to derive the bound on I_k for a window of length t based on the volume of interfering higher priority bundles. The height of the task under analysis determines the sensitivity to interference: since the task requires h_k available cores to run, it follows that for the task to suffer interference, at least $m - h_k + 1$ cores must be occupied by higher priority tasks. Similarly to [81], we can thus depict the interference as a rectangle in time × processor space, with a height of $m - h_k + 1$, which we call the *interference rectangle*. Figure 7.5 shows the worst case interference for three cases on a system of 4 cores. Note that, for simplicity of illustration all interfering tasks are lumped and drawn before the task

[4] For convenience and since we are interested in upper bounding the interference and response time, we use the term monotonic to denote a non-decreasing function.

under analysis. However, in preemptive systems, the interference can occur anytime while the task is active. Case (A) is similar to sequential tasks and the interference rectangle is of height $m - 1 + 1 = 4$, and the volume of interfering tasks is lumped together and divided over the number of cores. As the height of the task under analysis increases, the height of the interfering rectangle decreases. This can extend its horizontal length as the volume of interfering tasks is divided over less cores. However, note that interfering tasks with heights greater than the height of the interfering rectangle cannot contribute to the interference by more than the height of the rectangle itself. To formalize the derivation of the interfering rectangle length, we next introduce some definitions.

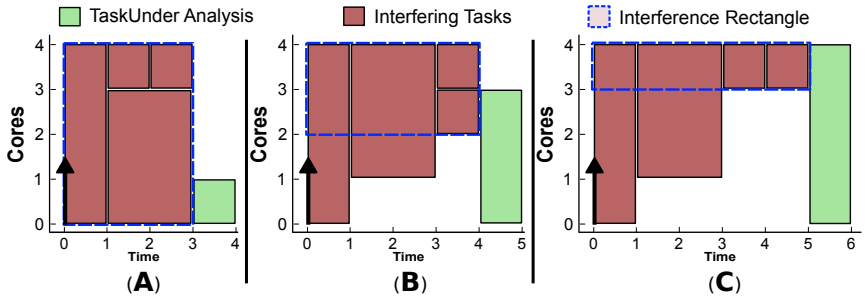


Figure 7.5: Interfering caused by higher priority tasks. The up arrow denotes the arrival time of the task under analysis.

Definition 7.2. *The contribution C_k^i of a task τ_i to the interference I_k, with $i \neq k$, is the cumulative amount of time that τ_i is executed while τ_k is active.*

Definition 7.3. *For a set of rigid tasks, the interference factor I_k^i of task τ_i on τ_k is defined as:*

$$I_k^i = \begin{cases} \frac{\min(h_i, m-h_k+1)}{m-h_k+1}, & \text{if } P_i \geq P_k; \\ 0, & \text{otherwise.} \end{cases} \tag{7.2}$$

Lemma 7.1. *For gang scheduling of rigid tasks with fixed priorities, the following is a bound on the interference on task τ_k:*

$$I_k \leq \lfloor \sum_{\forall i \neq k} (I_k^i \times C_k^i) \rfloor. \tag{7.3}$$

Proof. We seek to bound the cumulative amount of time I_k that τ_k is ready. Since we are using gang scheduling with fixed priorities, it follows that tasks with higher priority than τ_k must occupy at least $m - h_k + 1$ processors for I_k time. (A) As illustrated in Figure 7.5,

this implies that the cumulative volume of interfering higher priority tasks, bounded by a height of $m - h_k + 1$, must be equal to at least $I_k \times (m - h_k + 1)$. (B) Given contribution C_k^i, by definition the cumulative time that τ_i executes while τ_k is ready is no more than C_k^i, and thus the interfering volume of τ_i is at most $C_k^i \times \min(h_i, m - h_k + 1)$. Based on (A) and (B), we obtain:

$$\sum_{\forall i: i \neq k \wedge P_i \geq P_k} \left(C_k^i \times \min(h_i, m - h_k + 1)\right) \geq I_k \times (m - h_k + 1), \tag{7.4}$$

which is equivalent to:

$$\begin{aligned} I_k &\leq \sum_{\forall i: i \neq k \wedge P_i \geq P_k} \left(C_k^i(t) \times \frac{\min(h_i, m - h_k + 1)}{m - h_k + 1}\right) \\ &= \sum_{\forall i \neq k} \left(I_k^i \times C_k^i\right). \end{aligned} \tag{7.5}$$

Now due to the natural number time convention, $I_k \leq \sum_{\forall i \neq k}(I_k^i \times C_k^i)$ implies $I_k \leq \left\lfloor \sum_{\forall i \neq k}(I_k^i \times C_k^i) \right\rfloor$, completing the proof. □

7.2.1 Bounding the Contribution

It remains to compute an upper bound to the contribution C_k^i of τ_i on τ_k in a window of length t; we can then set $t = R_k$ and use the iteration in Equation 7.1 to compute the response time. We rely on the common concept of workload.

Definition 7.4. *The workload of task τ_i is the maximum cumulative time that τ_i is executed in a window of length t.*

Observation 7.1. *By definition, the contribution C_k^i of a bundle τ_i is upper bounded by its workload.*

As in the case of sequential tasks [39], the workload is maximized when the first job of τ_i starts executing as late as possible (with a starting time aligned with the beginning of the window) and later jobs are executed as soon as possible. The corresponding scenario is represented at the bottom of Figure 7.6, where the first job of τ_i starts executing $T_i - \Delta_i$ time units after its arrival, with $\Delta_i = T_i - R_i + L_i$. The corresponding workload bound $W^i(t)$ is represented at the top of Figure 7.6, and can be formally derived as:

$$W^i(t) = \begin{cases} \min(t, L_i), & \text{if } t \leq \Delta_i; \\ L_i + L_i \times \lfloor \frac{t - \Delta_i}{T_i} \rfloor + & \\ \min(L_i, (t - \Delta_i) \mod T_i), & \text{otherwise.} \end{cases} \tag{7.6}$$

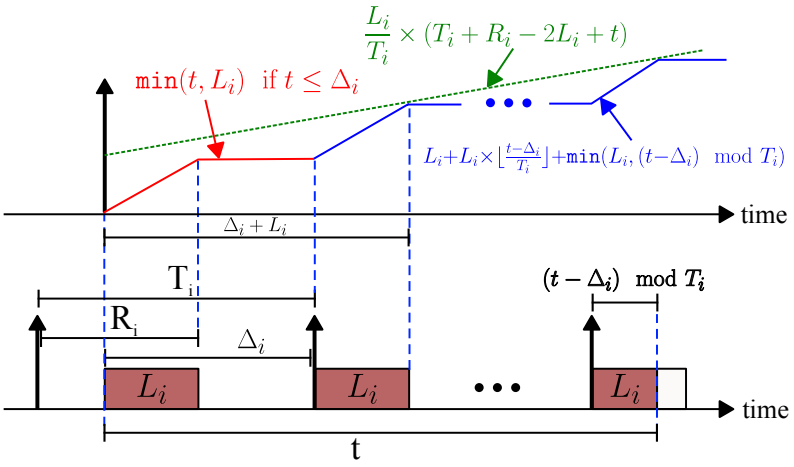


Figure 7.6: The maximum workload of task τ_i within a window of time t

Based on Observation 7.1 and Equation 7.6 we thus have:

$$C_k^i \leq W^i(t). \tag{7.7}$$

We are now ready to derive the response time R_k. Note that Equation 7.3 is monotonic in C_k^i, and Equation 7.6 is monotonic in t. Hence, based on Theorem 7.1, Lemma 7.1 and Equation 7.7, we can obtain the response time for gang scheduling with fixed priorities based on the following iteration:

$$R_k \leftarrow L_k + \left\lfloor \sum_{\forall i \neq k} \left(I_k^i \times W^i(R_k) \right) \right\rfloor. \tag{7.8}$$

7.2.2 Analysis for Multiple Bundles

We now consider the general case where tasks comprise multiple bundles. We employ the same analysis scheme: in particular, note that Definition 7.1 and Theorem 7.1 remain true, albeit the number of required available cores changes based on the height $h_{k,p}$ of the current active bundle. Similarly, the definition of contributions C_k^i and workload $W^i(t)$ remain valid, since they do not depend on the height of individual bundles. However, Definition 7.3 and Lemma 7.1, which allow us to bound the interference, cannot be used as they assume constant height. To bound the interference for the bundled case, we thus extend our definitions to account for the contribution and interference of individual bundles.

Definition 7.5. *The interference $I_{k,p}$ on bundle $\tau_{k,p}$ is the cumulative time during which $\tau_{k,p}$ is ready.*

Observation 7.2. *Since at most one bundle of τ_k can be ready at any time, by definition it holds: $I_k = \sum_{p=1}^{b_k} I_{k,p}$.*

Definition 7.6. *The contribution $C_k^{i,j}$ of a bundle $\tau_{i,j}$ to the interference I_k, with $i \neq k$, is the cumulative amount of time that $\tau_{i,j}$ is executed while τ_k is active. Similarly, the contribution $C_{k,p}^{i,j}$ of $\tau_{i,j}$ to the interference $I_{k,p}$, with $i \neq k$, is the cumulative amount of time that $\tau_{i,j}$ is executed while $\tau_{k,p}$ is active.*

Definition 7.7. *The interference factor $I_{k,p}^{i,j}$ of bundle $\tau_{i,j}$ on $\tau_{k,p}$ is defined as:*

$$I_{k,p}^{i,j} = \begin{cases} \frac{\min(h_{i,j}, m-h_{k,p}+1)}{m-h_{k,p}+1}, & \text{if } P_{i,j} \geq P_{k,p}; \\ 0, & \text{otherwise.} \end{cases} \tag{7.9}$$

Lemma 7.2. *For gang scheduling with fixed priorities, the following is a bound on the interference on bundle $\tau_{k,p}$:*

$$I_{k,p} \leq \Big\lfloor \sum_{\forall i \neq k} \sum_{j=1}^{b_i} (I_{k,p}^{i,j} \times C_{k,p}^{i,j}) \Big\rfloor. \tag{7.10}$$

Proof. We seek to bound the cumulative amount of time $I_{k,p}$ that $\tau_{k,p}$ is ready. Since we are using gang scheduling with fixed priorities, it follows that bundles with higher priority than $\tau_{k,p}$ must occupy at least $m-h_{k,p}+1$ processors for $I_{k,p}$ time. (A) As illustrated in Figure 7.5, this implies that the cumulative volume of interfering higher priority bundles, bounded by a height of $m-h_{k,p}+1$, must be equal to at least $I_k \times (m-h_{k,p}+1)$. (B) Given contribution $C_{k,p}^{i,j}$, by definition the cumulative time that $\tau_{i,j}$ executes while $\tau_{k,p}$ is ready is no more than $C_{k,p}^{i,j}$, and thus the interfering volume of $\tau_{i,j}$ is at most $C_{k,p}^{i,j} \times \min(h_{i,j}, m - h_{k,p} + 1)$. Based on (A) and (B), we obtain:

$$\sum_{\forall i: i \neq k \wedge P_{i,j} \geq P_{k,p}} \sum_{j=1}^{b_i} \big(C_{k,p}^{i,j} \times \min(h_{i,j}, m - h_{k,p} + 1)\big) \geq I_{k,p} \times (m - h_{k,p} + 1), \tag{7.11}$$

which is equivalent to:

$$I_{k,p} \leq \sum_{\forall i: i \neq k \wedge P_{i,j} \geq P_{k,p}} \sum_{j=1}^{b_i} \left(C_{k,p}^{i,j}(t) \times \frac{\min(h_{i,j}, m - h_{k,p} + 1)}{m - h_{k,p} + 1} \right)$$

$$= \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times C_{k,p}^{i,j} \right). \tag{7.12}$$

Now due to the natural number time convention, $I_{k,p} \leq \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times C_{k,p}^{i,j} \right)$ implies $I_{k,p} \leq \left\lfloor \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times C_{k,p}^{i,j} \right) \right\rfloor$, completing the proof. □

Next, we extend the definition of workload.

Definition 7.8. *The workload of bundle $\tau_{i,j}$ is the maximum cumulative time that $\tau_{i,j}$ is executed in a window of length t.*

Observation 7.3. *By definition, both the contribution $C_k^{i,j}$ and $C_{k,p}^{i,j}$ of a bundle $\tau_{i,j}$ is upper bounded by its workload.*

We can determine an upper bound $W^{i,j}(t)$ to the workload of $\tau_{i,j}$ using the same scenario as in Section 7.2.1: the first job of $\tau_{i,j}$ starts executing as late as possible and later jobs are executed as soon as possible. Since we make no assumption on the minimum execution time of bundles of τ_i executed before $\tau_{i,j}$, in the worst case $\tau_{i,j}$ can start executing immediately when the corresponding job of τ_i arrives. On the other hand, the following lemma bounds the latest time $T_i - \Delta_i^j$ that $\tau_{i,j}$ can start executing after the arrival of τ_i.

Lemma 7.3. *Let $R_i \leq D_i$ be an upper bound to the response time of task τ_i. Then each bundle $\tau_{i,j}$ must start executing no later than $R_i - \sum_{q=j}^{b_i} l_{i,q}$ time units after the arrival of the corresponding job of τ_i.*

Proof. Consider any window of time R_i starting with the arrival time of a job of τ_i; since $R_i \leq D_i$ and deadlines are constrained, only one job of τ_i can execute in the window. Since furthermore in the worst case the job must execute for L_i time units, the interference I_k in the time window cannot be greater than $R_i - L_i$. Now consider bundle $\tau_{i,j}$. Since the scheduler is work conserving and bundles are always executed in order, it follows that bundle $\tau_{i,j}$ must start executing no later than $I_k + \sum_{q=1}^{j-1} l_{i,j}$ from the beginning of the

window; note that $\sum_{q=1}^{j-1} l_{i,j}$ represents the worst case execution time of the bundles before $\tau_{i,j}$. This yields:

$$I_k + \sum_{q=1}^{j-1} l_{i,j} \leq R_i - L_i + \sum_{q=1}^{j-1} l_{i,j} = R_i - \sum_{q=j}^{b_j} l_{i,j}, \tag{7.13}$$

concluding the proof. □

Based on Lemma 7.3, we thus have $\Delta_i^j = T_i - R_i + \sum_{q=j}^{b_i} l_{i,q}$. Again referring to Figure 7.6, we obtain:

$$W^{i,j}(t) = \begin{cases} \min(t, l_{i,j}), & \text{if } t \leq \Delta_i^j; \\ l_{i,j} + l_{i,j} \times \lfloor \frac{t-\Delta_i^j}{T_i} \rfloor + & \\ \min(l_{i,j}, (t - \Delta_i^j) \mod T_i), & \text{otherwise.} \end{cases} \tag{7.14}$$

Finally, later in the section it will become useful to compute a linear approximation $\overline{W}^{i,j}(t)$ to the workload. Based on Figure 7.6, the resulting line has a slope of $\frac{l_{i,j}}{T_i}$ and has a value $\overline{W}^{i,j}(\Delta + l_{i,j}) = W^{i,j}(\Delta + l_{i,j}) = 2l_{i,j}$, which yields:

$$\overline{W}^{i,j}(t) = \frac{l_{i,j}}{T_i} \times (T_i + R_i - 2l_{i,j} + t). \tag{7.15}$$

Note that similarly to $W^{i,j}(t)$, $\overline{W}^{i,j}(t)$ is also monotonic in t. Furthermore, to tighten the linear bound, we also notice that by definition it must hold $C_{k,p}^{i,j} \leq t$ for any window of length t. In summary, all the following bounds (which will be later used as part of the proofs) hold for a window of length t:

$$C_k^{i,j} \leq W^{i,j}(t), \tag{7.16}$$

$$C_{k,p}^{i,j} \leq \overline{W}^{i,j}(t), \tag{7.17}$$

$$C_{k,p}^{i,j} \leq t. \tag{7.18}$$

Finally, using the same strategy as in Equation 7.8 and based on Observation 7.2, we could now determine $\bar{I}(R_k)$ by summing the interference bounds in Equation 7.10, thus obtaining:

$$R_k \leftarrow L_k + \sum_{p=1}^{b_k} \left\lfloor \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times W^{i,j}(R_k) \right) \right\rfloor. \tag{7.19}$$

7.2.3 Tightening the Analysis

Unfortunately, the bound resulting from Equation 7.19 is extremely pessimistic: when considering a higher priority bundle $\tau_{i,j}$, a single window of time of length R_k is used to determine its contribution on each bundle $\tau_{k,p}$ of the task under analysis. Hence, we are effectively "multiple counting" by having the same job of $\tau_{i,j}$ interfering with all the bundles of $\tau_{k,p}$ at the same time. However, in reality this is not possible as the following observation notices:

Observation 7.4. *Since at most one bundle of τ_k can be active at any time, by definition it holds: $C_k^{i,j} = \sum_{p=1}^{b_k} C_{k,p}^{i,j}$.*

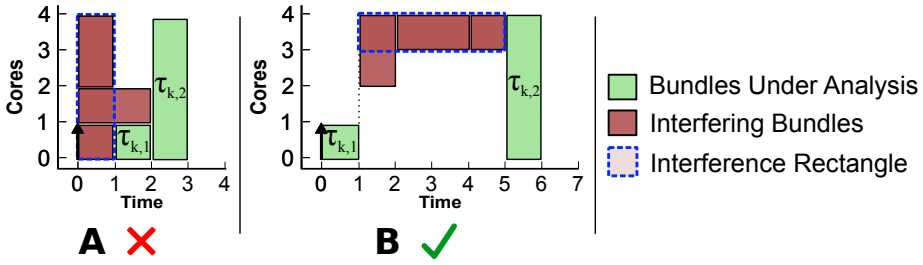


Figure 7.7: Worst-case contributions of interfering workloads. **A**: contributing to the first bundle. **B**: contributing to the second bundle.

To obtain a tighter analysis, we thus seek to determine the contributions $C_{k,p}^{i,j}$ on each bundle $\tau_{k,p}$ that maximize the total interference I_k. To give a concrete example, Figure 7.7 shows two cases of the same workloads. In case (A), the higher-priority bundles are assumed to contribute to the interference of the first bundle ($\tau_{k,1}$), whereas in case (B) they contribute to the second bundle ($\tau_{k,2}$). It is clear from the figure, that the contributions $C_{k,1}^{i,j}$ and $C_{k,2}^{i,j}$ are important to determine the total interference suffered by τ_k. In particular, in this specific example, the worst-case response time of τ_k is obtained by setting $\forall\ i,j : C_{k,1}^{i,j} = 0$ and $\forall\ i,j : C_{k,2}^{i,j} = l_{i,j}$, which conforms to case (B).

Listing 7.1 LP Problem formulation for Interference $\bar{I}_k(t)$

Input: `window length` t, `task parameters`

Output: $\bar{I}_k = \lfloor \sum_{p=1}^{b_k} \bar{I}_{k,p} \rfloor$

Free Variables: $\forall\ i \neq k, j, p \quad C_{k,p}^{i,j} \in \mathbb{R}_{\geq 0}$

Objective: `maximize` $\sum_{p=1}^{b_k} \bar{I}_{k,p}$

Subject to:

$1a.\ \forall\ i \neq k, j \quad \sum_{p=1}^{b_k} C_{k,p}^{i,j} \leq W^{i,j}(t)$

$1b.\ \forall\ i \neq k, j \quad \sum_{j=1}^{b_i} \sum_{p=1}^{b_k} C_{k,p}^{i,j} \leq W^{i}(t)$

$\uparrow$ *workload constraints on* τ_k

$2a.\ \forall\ i \neq k, j, p \quad C_{k,p}^{i,j} \leq \overline{W}^{i,j}(l_{k,p} + \bar{I}_{k,p})$

$2b.\ \forall\ i \neq k, j, p \quad C_{k,p}^{i,j} \leq l_{k,p} + \bar{I}_{k,p}$

$\uparrow$ *workload constraints on* $\tau_{k,p}$

$3.\ \forall\ i \neq k, j, p \quad C_{k,p}^{i,j} \leq l_{i,j}$ `if` $\exists q \neq j, SF_{k,p}^{i,q} =$ `true`

$\uparrow$ *bundle-priority constraints*

Where:

$$\bar{I}_{k,p} = \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times C_{k,p}^{i,j} \right)$$

$$SF_{k,p}^{i,q} = (P_{i,q} < P_{k,p}) \wedge (h_{i,q} + h_{k,p} > m) \wedge$$
$$\big((h_{i,q} + m - h_{k,p} + 1 > m) \vee$$
$$(\forall \tau_{y,z}, y \neq k \wedge P_{y,z} \geq P_{k,p} : h_{i,q} + h_{y,z} > m) \big)$$

We next introduce a linear programming formulation that computes the required upper bound $\bar{I}_k(t)$ by finding contributions $C_{k,p}^{i,j}$ that maximize the interference. The interference bound is obtained by summing individual interference upper bounds $\bar{I}_{k,p}$ for each bundle of $\tau_{k,p}$. The contributions are subject to five constraints. Constraint 1a bounds the overall contribution $C_k^{i,j} = \sum_{p=1}^{b_k} C_{k,p}^{i,j}$ on task τ_k using Equation 7.16, thus satisfying Observation 7.4. However, as noticed in Section 7.2.2, $W^{i,j}(t)$ is derived assuming that $\tau_{i,j}$ executed for its worst case time, while bundles of τ_i before $\tau_{i,j}$ executed for zero time. Clearly, this situation cannot happen for every bundle of τ_i; hence, when writing Constraint 1a over all j, we can effectively over-estimate the workload of task τ_i. We make one final observation:

Observation 7.5. *Since at most one bundle of τ_i can be active at any time, by definition it holds: $C_k^i = \sum_{j=1}^{b_i} C_k^{i,j}$.*

Therefore, based on Observations 7.4, 7.5 and Equation 7.7, in Constraint 1b we also bound the contribution of τ_i on τ_k such that:

$$C_k^i = \sum_{j=1}^{b_i} C_k^{i,j} = \sum_{j=1}^{b_i} \sum_{p=1}^{b_k} C_{k,p}^{i,j} \leq W^i(t). \tag{7.20}$$

Moving on, Constraints 2a, 2b bound the contributions on each bundle under analysis $\tau_{k,p}$, rather than on the entire task τ_k. Note that here, $l_{k,p} + \bar{I}_{k,p}$ represents a bound on the time that $\tau_{k,p}$ can be active, that is, the response time of the bundle. We use the linearized upper bound in Equations 7.17, 7.18 rather than the workload bound $W^{i,j}(t)$ to ensure that the algorithm formulation is a linear programming problem; note that for the same reason, we remove the floor from Equation 7.10 in the expression for the interference bound $\bar{I}_{k,p}$, and we relax the variables $C_{k,p}^{i,j}$ to be real rather than natural numbers.

Finally, Constraint 3 adds a restriction on the values of the contributions that allow us to tighten the analysis. The condition $SF_{k,p}^{i,q}$ evaluates to true if a bundle $\tau_{i,q}$ of τ_i cannot execute while $\tau_{k,p}$ is active. Consider another bundle $\tau_{i,j}$ of τ_i with higher priority than $\tau_{k,p}$. Since bundles are always executed in order, between the execution of any two jobs of $\tau_{i,j}$, $\tau_{i,q}$ must be executed once. But if $SF_{k,p}^{i,q} = \text{true}$ this is not possible while $\tau_{k,p}$ is active, meaning that no more than one job of $\tau_{i,j}$ can interfere with $\tau_{k,p}$; hence, we obtain $C_{k,p}^{i,j} \leq l_{i,j}$. To understand how the condition $SF_{k,p}^{i,q}$ is derived, refer to Figure 7.8, and consider a bundle $\tau_{i,q}$ with $h_{i,q} = 2$ and lower priority than $\tau_{k,p}$ ($P_{i,q} < P_{k,p}$). Note that in both Figure 7.8(A) and Figure 7.8(B), $\tau_{i,q}$ cannot execute while $\tau_{k,p}$ is also executing because there are not enough cores for both bundles ($h_{i,q} + h_{k,p} > m$). In Figure 7.8(A), $\tau_{i,q}$ also cannot execute

in parallel with the interference rectangle, since its height ($m - h_{k,p} + 1 = 2$) leaves only one core free, while $\tau_{i,q}$ requires two (that is, $h_{i,q} + m - h_{k,p} + 1 > m$). In Figure 7.8(B), the height of the interference rectangle is 1, but all bundles that have a priority higher than $\tau_{k,p}$ ($\tau_{y,z}$ with $y \neq k$ and $P_{y,z} \geq P_{k,p}$), and can thus interfere with it, have a height of at least 2, which again does not leave enough cores for $\tau_{i,q}$ ($h_{i,q} + h_{y,z} > m$). In either case, $\tau_{i,q}$ cannot execute in parallel with the interfering rectangle; and since it also cannot execute in parallel with $\tau_{k,p}$, it follows that it cannot run while $\tau_{k,p}$ is active. The next two lemmas formally prove the correctness of the constraint.

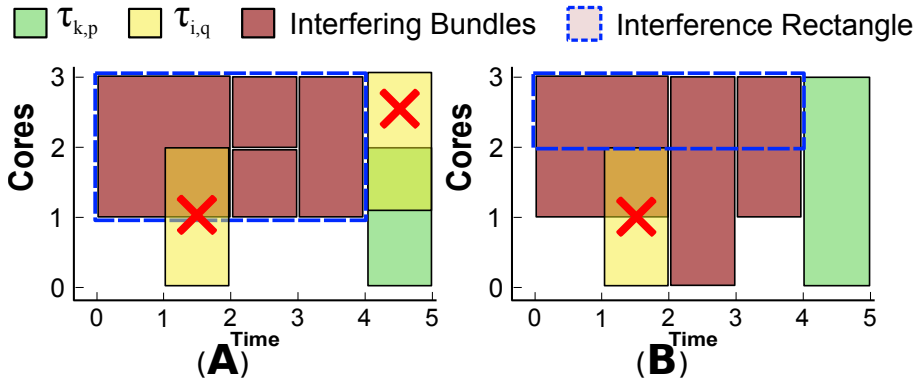


Figure 7.8: Examples of lower-priority bundles' ability to run

Lemma 7.4. *If $SF_{k,p}^{i,q}$ is true, then $\tau_{i,q}$ cannot execute while $\tau_{k,p}$ is active.*

Proof. Since $SF_{k,p}^{i,q}$ holds, we have $h_{i,q} + h_{k,p} > m$; hence, $\tau_{i,q}$ and $\tau_{k,p}$ cannot execute simultaneously on m cores.

Next, assume that $\tau_{k,p}$ is ready. Since it also holds $P_{i,q} < P_{k,p}$, other bundles with higher priority than both $\tau_{k,p}$ and $\tau_{i,q}$ must occupy at least $m - h_{k,p} + 1$ cores. Given $SF_{k,p}^{i,q}$ =true, we finally have $h_{i,q} + m - h_{k,p} + 1 > m$ or $\forall \tau_{y,z}, y \neq k \wedge P_{y,z} \geq P_{k,p} : h_{i,q} + h_{y,z} > m$. In either case, there are not enough cores available for $h_{i,q}$ to execute in parallel with the higher priority bundles.

In summary, we have shown that if $SF_{k,p}^{i,q}$ holds, then $\tau_{i,q}$ cannot execute while $\tau_{k,p}$ is either ready or executing. Hence, $\tau_{i,q}$ cannot execute while $\tau_{k,p}$ is active. □

Lemma 7.5. *In any window of time where at most one job of $\tau_{k,p}$ is active, if $SF_{k,p}^{i,q}$ is true for a bundle $\tau_{i,q}$ with $i \neq k$, then we have $C_{k,p}^{i,j} \leq l_{i,j}$ for all other bundles of τ_i.*

Proof. Let t_a be the earliest time in the window in which the only job of $\tau_{k,p}$ is active, and let t_b be the last such time (if there is no such job, then trivially $C_{k,p}^{i,j} = 0$ and the lemma holds). After being activated, a bundle remains active until it finishes; hence, $[t_a, t_b]$ is a continuous interval of time.

By contradiction, assume that $C_{k,p}^{i,j} > l_{i,j}$; then, at least two jobs of $\tau_{i,j}$ must execute within $[t_a, t_b]$. However, since bundles are always executed in order, at least one job of $\tau_{i,q}$ must also execute between the finishing time of the first job and the time the second job of $\tau_{i,j}$ becomes active. This implies that $\tau_{i,q}$ executes in $[t_a, t_b]$, which is impossible by Lemma 7.4. □

We can now show that Listing 7.1 computes a valid upper bound $\bar{I}_k(t)$ in Theorem 7.2. We start with a helper lemma.

Lemma 7.6. *Consider a window of time of length $t \leq D_k$ starting with the arrival of a job of τ_k. The contributions $C_{k,p}^{i,j}$ for such window must satisfy Constraints 1-3 in Listing 7.1.*

Proof. We show that the values of the contribution $C_{k,p}^{i,j}$ must satisfy all five constraints in Listing 7.1.

Constraint 1a: by Observation 7.4 and Equation 7.16 we have: $\sum_{p=1}^{b_k} C_{k,p}^{i,j} = C_k^{i,j} \leq W^{i,j}(t)$. Hence, the constraint must be satisfied.

Constraint 1b: similarly to the case of Constraint 1a, by Observations 7.4, 7.5 and Equation 7.7 we obtain $\sum_{j=1}^{b_i} \sum_{p=1}^{b_k} C_{k,p}^{i,j} \leq W^i(t)$.

Constraint 2a: note that the constraint is equivalent to $C_{k,p}^{i,j} \leq \overline{W}^{i,j}\Big(l_{k,p} + \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \big(I_{k,p}^{i,j} \times C_{k,p}^{i,j}\big)\Big)$. By contradiction, assume the constraint is violated. By monotonicity of $\overline{W}^{i,j}(t)$ and Lemma 7.2, this yields:

$$\begin{aligned} C_{k,p}^{i,j} &> \overline{W}^{i,j}\Big(l_{k,p} + \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \big(I_{k,p}^{i,j} \times C_{k,p}^{i,j}\big)\Big) \\ &\geq \overline{W}^{i,j}\Big(l_{k,p} + \lfloor \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \big(I_{k,p}^{i,j} \times C_{k,p}^{i,j}\big) \rfloor\Big) \\ &\geq \overline{W}^{i,j}(l_{k,p} + I_{k,p}). \end{aligned} \tag{7.21}$$

But since the schedule is work conserving, $\tau_{k,p}$ cannot be active for more than $l_{k,p} + I_{k,p}$ time units; hence based on Equation 7.17 applied to a window of length $t = l_{k,p} + I_{k,p}$ we must have $C_{k,p}^{i,j} \leq \overline{W}^{i,j}(l_{k,p} + I_{k,p})$, a contradiction.

Constraint 2b: similarly to the case of Constraint 2a, if the constraint is violated we obtain: $C_{k,p}^{i,j} > l_{k,p} + I_{k,p}$, which contradicts Equation 7.18.

Constraint 3: since we are assuming a window of length $t \leq D_k$ starting with the arrival of a job of τ_k, it follows that at most one job of $\tau_{k,p}$ can be active in the window; hence, Lemma 7.5 must hold and the constraint must be satisfied. □

Theorem 7.2. *The LP formulation in Listing 7.1 computes an upper bound to the interference I_k for a window of time of length $t \leq D_k$ starting with the arrival of a job of τ_k. Furthermore, the bound is monotonic in t.*

Proof. We first show that Listing 7.1 is indeed a Linear Programming problem. The free variables $C_{k,p}^{i,j}$ are (non-negative) real variables. Note that $SF_{k,p}^{i,j}$, $W^{i,j}(t)$ and $W^{i}(t)$ are constants for given input t. $\bar{I}_{k,p}$ is a linear expression of variables $C_{k,p}^{i,j}$, while the objective function and Constraints 2a, 2b are linear in $\bar{I}_{k,p}$; but since the composition of linear functions is linear, all expressions are indeed linear in the free variables.

We next show that the computed value for the objective function is monotonic in t. Note that in Listing 7.1, the value of t is only used to compute functions $W^{i,j}(t)$ and $W^{i}(t)$ in Constraints 1a and 1b. But since both functions are monotonic in t, it follows that increasing t leads to a relaxation of the constraints; hence, the objective function cannot decrease.

Finally, we show that Listing 7.1 computes a valid upper bound for I_k. First note that any assignment to variables $C_{k,p}^{i,j}$ that maximizes the objective function $\sum_{p=1}^{b_k} \bar{I}_{k,p}$ also maximizes the value of $\bar{I}_k = \lfloor \sum_{p=1}^{b_k} \bar{I}_{k,p} \rfloor$. Furthermore, based on Observation 7.2 and Lemma 7.2 it holds:

$$\bar{I}_k = \lfloor \sum_{p=1}^{b_k} \bar{I}_{k,p} \rfloor = \lfloor \sum_{p=1}^{b_k} \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times C_{k,p}^{i,j} \right) \rfloor \geq$$
$$\sum_{p=1}^{b_k} \lfloor \sum_{\forall i \neq k} \sum_{j=1}^{b_i} \left(I_{k,p}^{i,j} \times C_{k,p}^{i,j} \right) \rfloor \geq \sum_{p=1}^{b_k} I_{k,p} = I_k; \tag{7.22}$$

that is, for a given set of contributions $C_{k,p}^{i,j}$, $\bar{I}_k$ is a valid bound to the interference I_k. Now by Lemma 7.6, we know that any valid set of contributions in a a window of time

of length $t \leq D_k$ starting with the arrival of a job of τ_k must respect all constraints of the optimization problem; and since furthermore the optimization problem relaxes the value of the variables (from natural numbers to real), it follows that no valid values of the contributions in the studied window can produce a higher value of $\bar{I}_k$. This shows that $\bar{I}_k$ is indeed an upper bound to the interference for any possible valid values of the contributions, concluding the proof. □

Based on Theorem 7.2, the response time R_k of the task under analysis is then computed using Equation 7.1, where $\bar{I}_k(R_k)$ is obtained using Listing 7.1.

7.2.4 Priority Assignment

Based on the presented analysis, we now discuss how to assign priorities to tasks or bundles. For the case of per-task priorities, it is well-known that even for sequential tasks, neither Rate-Monotonic (RM) nor Deadline-Monotonic (DM) assignments are optimal in global scheduling [50]. Audsley's Optimal Priority Assignment (OPA) algorithm is optimal for some schedulability analyses [23], but the proof of optimality requires a key condition: the schedulability of a task under analysis should not depend on the relative ordering of higher priority tasks. Unfortunately, this is not true for either our approach or related work [109], since the workload upper bounds for higher priority task τ_i in Equations 7.6, 7.14 are functions of the response time R_i of τ_i, which itself depends on tasks with priority higher than τ_i.

Hence, in rest of this chapter, we consider the DM policy for per-task priority assignment. However, for the case of per-bundle priorities, we propose a heuristic to further optimize the schedulability of bundled task sets. As noted when discussing Figure 7.5, the height of the interference rectangle for "tall" bundles is smaller, which in turn can lead to higher interference for a given amount of contribution. For this reason, the LP problem will attempt to assign as much contribution as possible to the tallest bundle of the task under analysis. Consequently, we propose to raise the priorities of tall bundles; this reduces the contribution that the LP can assign to such bundles, at the cost of generating some amount of priority inversion. In particular, we adopt the following simple heuristic, which modifies the DM priorities $1/D_i$ based on the ratio $h_{i,j}/m$ between the height of a bundle and the number of cores in the system:

$$P_{i,j} = \frac{1}{D_i} \times \left(1 + \frac{h_{i,j}}{m}\right). \tag{7.23}$$

Note that we scale the impact of $h_{i,j}$ based on the deadline to avoid causing too much priority inversion. We evaluate the quality of the heuristic in Section 7.3 comparing it against the per-task DM assignment.

7.2.5 Discussion

Sustainability: as discussed in Section 7.1.1, an executing bundle always reserves the same number of cores $h_{i,j}$. However, the bundle could execute for less than its worst case length $l_{i,j}$. The analysis is sustainable with respect to changes in bundle length: in particular, note that Lemma 7.3 shows that the worst case scenario used to derive the workload $W^{i,j}(t)$ remains correct even if some bundles of τ_i execute for less than their worst case length.

Holistic Analysis: as noted in Section 7.2.4, Equations 7.6, 7.14 assume that a bound on the response time of interfering task R_i is known. When priorities are assigned per-task and are distinct, it is sufficient to compute the response time of each task according to Equation 7.1, from the highest priority to the lowest priority one. However, in the case of per-bundle priorities, this poses a problem: some bundles of τ_i could be higher priorities than other bundles of τ_k and vice-versa, creating a circular relation between R_i and R_k.

We can solve the issue by employing a standard holistic analysis framework, as shown in Algorithm 7.2. The algorithm starts by assigning a response time of L_i to each task τ_i. Then, at each iteration, the algorithm computes a new set of response times R'_k as the fixed points of Equation 7.1, applied to each task under analysis τ_k using the response times R_i of the previous iteration. The algorithm continues until the response times converge with $R'_i = R_i$ for all tasks, or we obtain $R_k > D_k$ for at least one task, in which case we fail to prove the task set schedulable.

It is immediate to see that if the algorithm converges, then we have $R_i \leq D_i$ for all tasks and the obtained R_i must indeed be valid upper bounds. Furthermore, note that $W^i(t), W^{i,j}(t)$ and $\overline{W}^{i,j}(t)$ are monotonic in R_i. Hence, if the response times R_i of interfering tasks increase, then the response time R'_k computed based on Equation 7.1 and Listing 7.1 cannot decrease. Due to the natural number time convention, this means at each iteration of the algorithm either all response times are the same as the previous iteration, in which case the algorithm terminates, or at least one response time must increase by a discrete amount. Hence, the algorithm terminates in a finite number of iterations.

Complexity and Precision: the complexity of each iteration of the fixed-point recursion in Equation 7.1 is dominated by the LP problem in Listing 7.1. Given $b_{\max} = \max_{\forall i}\{b_i\}$, both the number of variables and the number of equations in the problem

Algorithm 7.2 Holistic Analysis Iteration

1: $\forall i \quad : R'_i = L_i$
2: **repeat**
3: $\quad \forall i \quad : R_i = R'_i$
4: $\quad$ **for** each τ_k **do**
5: $\quad\quad$ Compute R'_k based on Equation 7.1 using
6: $\quad\quad\quad$ the $\{R_i\}$ values
7: $\quad\quad$ **if** $R'_k > D_k$ **return** unknown
8: **until** $\forall i \quad : R'_i = R_i$
9: **return** schedulable

are $\mathcal{O}(n \times b_{max}^2)$. Since an LP problem can be solved in polynomial time in the number of variables and constraints, the resulting schedulability analysis is of pseudo-polynomial complexity.

We make two observations on the precision of the analysis. First, as noticed in Section 7.2, some constraint relaxations were required to obtain a LP problem, chief among them the substitution of $W^{i,j}(t)$, which is neither linear nor convex/concave, with $\overline{W}^{i,j}(t)$. These relaxations have been employed to avoid the complexity of solving a non linear problem. Second, while we presented our analysis based on the response time framework introduced in [39], the same analysis strategy could also be applied for the problem window framework [30, 81] by invoking Listing 7.1 based on the length of the window. The trade-off is a worse bound on $W^{i,j}(t)$ for tasks that include carry-in (an interfering task has carry-in if its first job arrives before the start of the time window) for the ability to limit the number of tasks that have carry-in. However, limiting carry-in is challenging in gang-scheduling: as pointed out in Section 2.2.1, the solution adopted in [81] is incorrect. We thus reserve such investigation as future work.

7.3 Evaluation

We evaluate the effectiveness of the proposed approach based on synthetic task sets generation. A bundled task τ_i is generated by first uniformly selecting a total WCET L_i in the interval $[10, 150]$ and a number of bundles b_i in $[2, 5]$. The period T_i is then uniformly generated in the interval $[L_i, 10 \times L_i]$, resulting in a task utilization U_i between $V_i/(10 \times L_i)$ and V_i/L_i, and a minimum possible period of 10 and maximum possible period of 1500. Deadline is assigned equal to period (implicit deadline). To obtain a task set τ with a

given total utilization value, we continue generating tasks and adding them to the task set until we reach the desired utilization.

We consider three types of task sets, meant to represent different types of applications, based on which we alter the generation of the height $h_{i,j}$ and length $l_{i,j}$ of each bundle. For *lightly-parallel* task sets, each bundle is randomly categorized as either a *short* or *tall* bundle. Bundles in the short category are assigned a height in the interval $\left[1, \lceil 0.3 \times m \rceil\right]$, while bundles in the tall category are assigned a height in $\left[m - \lceil 0.3 \times m \rceil + 1, m\right]$. For what concerns the length, 80% of the WCET L_i is randomly distributed among the short bundles, while the remaining 20% is distributed among tall bundles. The generation process for *heavily-parallel* task sets is similar, except that 80% of the WCET is assigned to tall bundles and 20% to short ones. Finally, for *mixed* task sets, we do not divide the bundles into categories; instead, each bundle is assigned a height in $[1, m]$, and L_i is randomly distributed among all bundles.

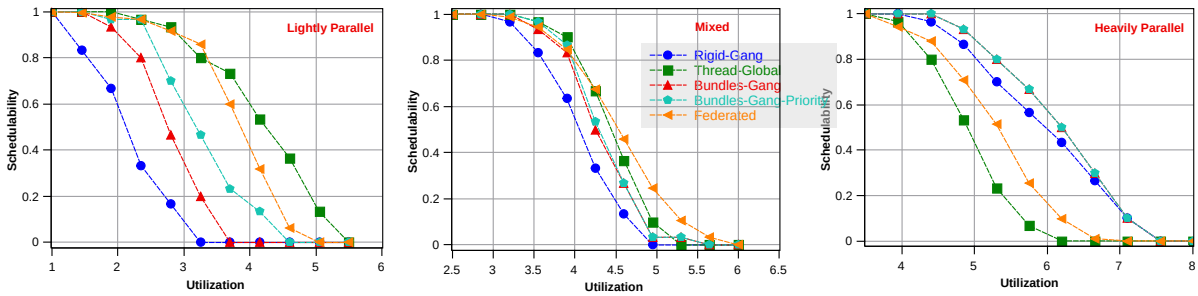


Figure 7.9: Schedulability test of the compared analyses on 8 cores with respect to task set types

Figure 7.9 shows results in terms of percentage of schedulable task sets based on total task set utilization, for each of the three task set types on a system with $m = 8$ cores. In Figure 7.10, we also show the case of a system with $m = 32$ cores; results are similar.

We show five curves in each graph. *bundles* represents our analysis in Section 7.2 using per-task priorities, while *bundles-priority* represents the same analysis using the per-bundle priority assignment introduced in Section 7.2.4. In the case of *rigid*, we again apply our analysis, except that we consider a rigid model where each task comprises a single bundle of length L_i and a height $h_i^{\max} = \max_{j=1}^{b_i}\{h_{i,j}\}$. The *thread* curve represents the state-of-the-art response time analysis for fixed-priority thread scheduling of DAG tasks in [109] [5]. Since the analysis in [109] relies on knowledge of the critical path length and

[5]While [109] discusses conditional DAG tasks, it can also be applied to traditional non-conditional DAG

volume of each DAG-task, for comparison with the bundled model we set the critical path length equal to L_i and the volume equal to V_i; indeed, note that when running in isolation on a system with m cores, as discussed in Section 7.1.2, the makespan of a DAG with maximum degree of parallelism m is equal to its critical path length. In practice, for real application we expect that the volume under thread scheduling would be inflated due to the synchronization overheads of independently scheduled threads. Finally, *federated* represents the schedulability test based on capacity argumentation bound provided in [93]. In federated, the task set is considered not schedulable if it requires more cores than the total number of cores provided by the system.

Based on the figure, it can be observed that *rigid* performs poorly in the case of lightly-parallel tasks. This is expected due to the pessimism in the rigid model, as the rigid box of a task ($L_i \times h_i^{max}$) overestimates its actual volume V_i. As tasks become more "packed", such that the difference between the rigid box and the actual volume is smaller, the performance of the rigid model becomes closer to the one of the bundle model. Clearly, in the extreme case of a task set where each task always requires $h_i^{\max}$ cores, there would be no difference between the models.

In addition, the suggested per-bundle priority heuristic is mostly useful for the case of lightly-parallel tasks. By raising the priority of tall bundles, the suggested heuristic causes some amount of priority inversion to higher priority tasks. In the case of lightly-parallel tasks, tall bundles have shorter length; hence, the amount of interference caused by priority inversion is smaller compared to the other cases.

Note that the global thread scheduling approach is performing comparatively poorly for highly-parallel task sets. The reason is that the analysis in [109] suffers from an extra self interference term $(V_i - L_i)/m$, which does not exist in gang scheduling. For more packed task sets, this term grows with respect to the WCET L_i. On the other hand, federated scheduling can suffer from cores overprovisioning in a different way to the gang rigid model. In federated scheduling, a parallel task is allocated the minimum number of cores that makes it barely schedulable. In the worst-case, one complete core might be wasted for each parallel task. Although the schedulability test in federated scheduling [93] does not suffer from self interference as in global thread scheduling, based on how much core overprovisioning is incurred, federated scheduling can do comparatively worse or better than global thread scheduling. Note that as the number of cores in the system increases, federated scheduling improves over global thread scheduling, as show in Figure 7.10, due to more available cores that can be dedicated for parallel tasks.

tasks, which are a special case of the former.

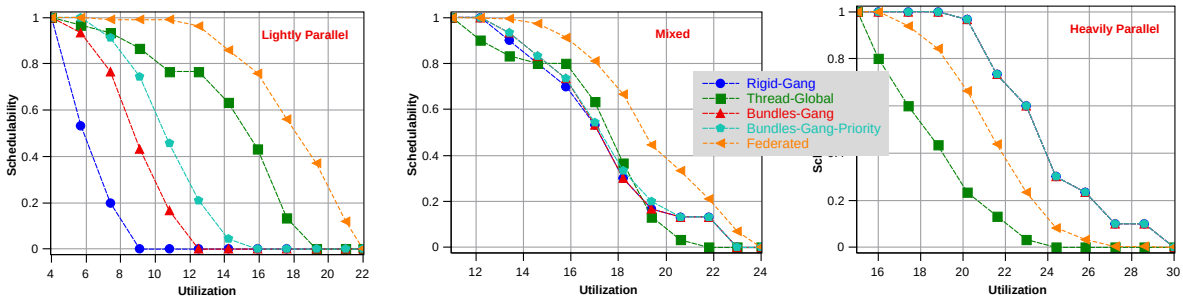


Figure 7.10: Schedulability test of the compared analyses on 32 cores with respect to task set types

7.4 Bundled Scheduling for the 3-Phase Task Model

The bundled scheduling model presented so far does not manage accesses to main memory. To enforce isolation among parallel tasks, we next discuss how to integrate the 3-phase task model with bundled scheduling. As discussed in Section 3.2, the 3-phase task model suffers more delay in preemptive policy than in non-preemptive policy, mainly due to excessive reloading wasting the memory transfer time. Hence, the main question we need to address is: can we extend the bundled model by considering a non-preemptive gang scheduling policy?

Unfortunately, it is easy to see that under a work-conserving, non-preemptive gang scheduling scheme, a task can suffer unbounded blocking due to lower-priority tasks. Consider the example in Figure 7.11. In the preemptive case, the scheduler preempts two lower-priority bundles on three cores to schedule the just arrived higher-priority bundle that requires 4 cores. On the other hand, the non-preemptive case fails to schedule the released higher-priority bundle. Specifically, at time zero, when the higher-priority bundle is released, one core is occupied by one bundle of height one. Since it is a non-preemptive scheduler, it waits for that bundle to finish. However, another lower-priority bundle arrives before the first lower-priority bundle finishes. As a work-conserving scheduler, it will schedule the newly arrived bundle as it only requires one core. By the time the first lower-priority bundle finishes, at time one, still not all four cores are free, hence, as a non-preemptive scheduler, cannot schedule the pending higher-priority bundle yet. This scenario can go on indefinitely.

To overcome the aforementioned issues, we propose to adopt a deferred preemption approach [51]. Specifically, a lower-priority bundle is allowed to run for σ time units before it can be preempted by a higher-priority bundle. Unfortunately, a naive application of

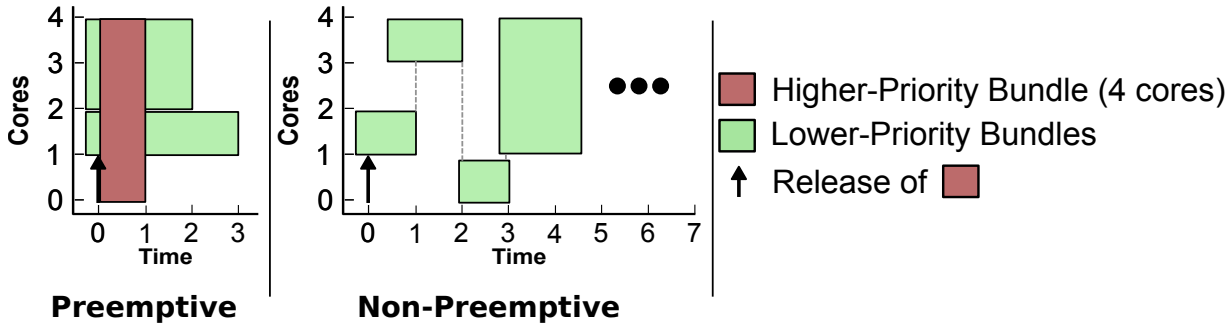


Figure 7.11: Preemptive versus non-preemptive gang scheduling

deferred preemption does not prevent unbounded blocking; the problem stems from the unsynchronized preemption points on different cores, causing scheduling decisions for each core to happen at different time. This timing behavior impedes the work-conserving scheduler from making system-wide scheduling decision for all cores and limits its decision to just few free cores. Figure 7.12 shows an example of the problem (**deferred preemption**) and the proposed solution (**synchronous deferred preemption**).

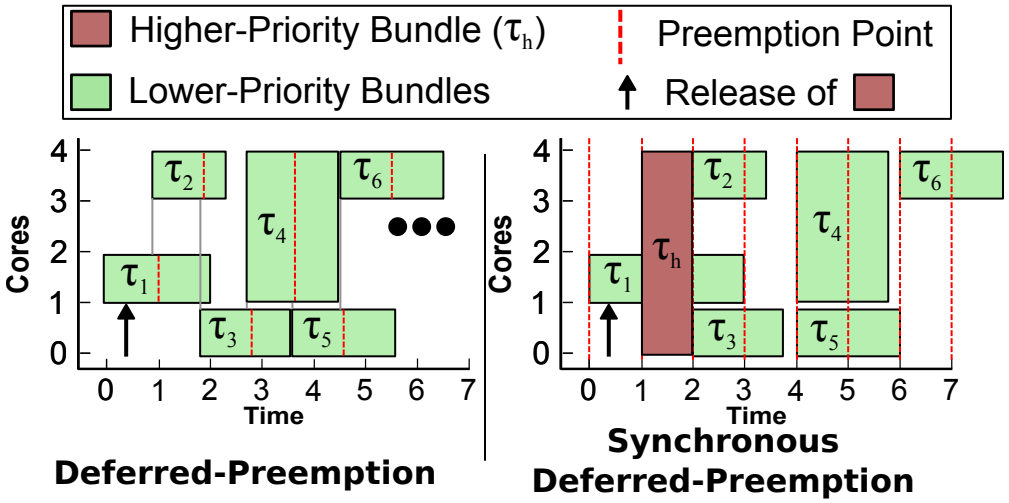


Figure 7.12: Deferred preemption versus synchronous deferred preemption in gang scheduling

In the deferred preemption case, a bundle is scheduled on the assigned set of cores as soon as it arrives if the cores are available. After that the bundle can run for σ time

before it can be preempted (red-dashed lines). Since different bundles can be activated and scheduled on different cores at different times, the respective preemption points of each bundle can also be at different times. As shown in the figure, it is possible that another bundle arrives just before the preemption point and prevents the higher priority bundle from being scheduled. Just like the non-preemptive case, this scenario can be repeated indefinitely. In a non-gang scheduling algorithm, this problem does not exist since scheduling decisions are local to cores, thus there will be no pending task as long as there is at least one idle core.

The proposed solution, synchronous deferred preemption, relies on synchronizing the context-switches on all cores according to a system-wide time-triggered event that fires at a fixed rate (every σ). As shown on the right side of Figure 7.12, scheduling decisions happen at the synchronous preemption points only. For example, unlike in the traditional deferred preemption case, although τ_2 arrives before the preemption point of τ_1, it cannot be scheduled right when it arrives. Instead, it has to wait until the next system-wide preemption point, which is strictly at time one in the example. Since τ_1 already executed for one time slot of σ and τ_h is pending, the scheduler will preempt τ_1, schedule τ_h, and keep the lower-priority τ_2 pending. The downside of this solution is the wasted time due to synchronization. Specifically, if a bundle finishes execution before the preemption point, the relevant cores stay idle until the next preemption point. This is depicted in the figure at time 4 and time 6. The wasted time is maximized when the bundle finishes ε time after the preemption point, thus wasting almost a complete slot.

To account for the wasted CPU time due to the synchronous preemption policy, we thus incorporate the wasted CPU time into the WCETs of the bundles, by considering an extended bundle length $\overline{l_{k,p}}$ that is a multiple of σ:

$$\overline{l_{k,p}} = \left\lceil \frac{l_{k,p}}{\sigma} \right\rceil \times \sigma. \tag{7.24}$$

According to our proposed execution model, a bundle needs to be loaded into the local memories of the targeted cores before it can run. Similar to Section 3.3, we assume a fixed-size time slot for DMA operations. In addition, we assign the slot size for DMA operations to be equal to the slot size between successive preemption points; effectively making the scheduling decisions for both cores and DMA at the preemption points. We further assume that the size of the allocated time slot σ is sufficient to load one partition in each of the m cores, including any required unload operation to free up the targeted partitions. The DMA operations are non-preemptive; thus, once a bundle starts loading, it is guaranteed to run on the CPU for at least σ time during the next time slot. Unlike the eager DMA

loading policy for the non-preemptive cases in Sections 3.2, 3.3 and in Chapter 5, we apply a lazy DMA loading mechanism, meaning that a bundle is only loaded one slot before it is executed, even if free partitions are available beforehand. This policy is to avoid unnecessary blocking in the case where a lower-priority bundle would be loaded, but not executed in the next time slot; nonetheless, the lower priority bundle would have to execute at some point for at least one slot since it has been loaded, hence introducing an extra slot of blocking.

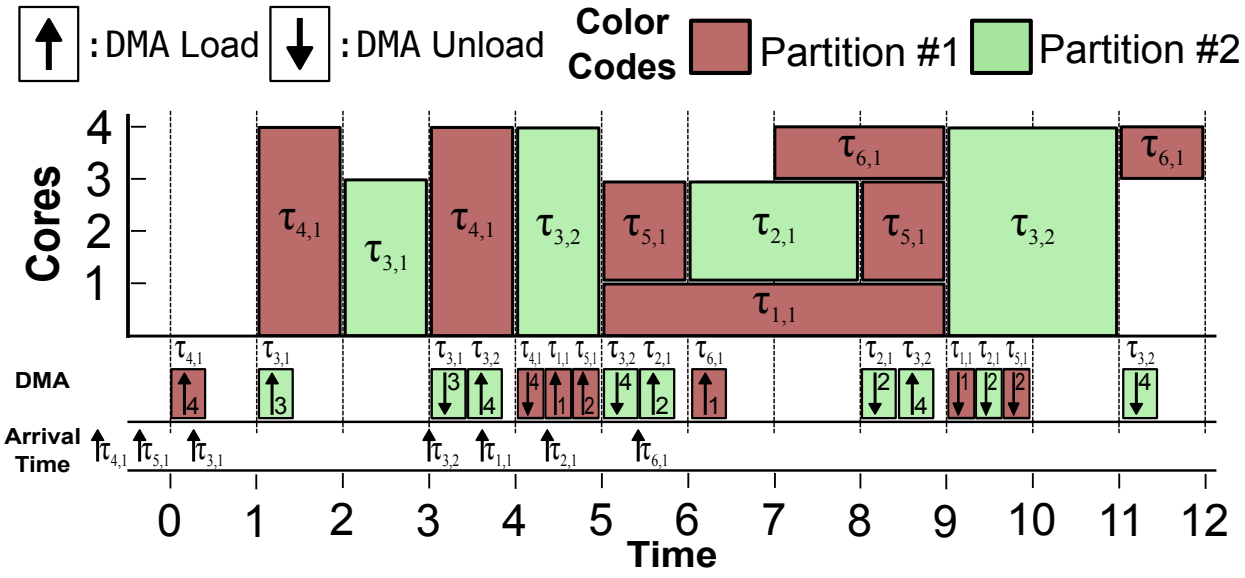


Figure 7.13: Example schedule of integrating bundles into the 3-phase execution model

Figure 7.13 shows an example of bundle scheduling integrated with the 3-phase execution model. In the example, there are six tasks with ordered priories: τ_6 is the lowest priority and τ_1 is the highest priority. τ_3 and τ_4 consist of two bundles each, whereas the other tasks consist of one bundle each. The scheduling decisions happen at the beginning of the time slots, times $0, 1, 2..., 12$ in the figure. Each DMA operation, in the figure, is annotated with the name of the bundle, and the number of partitions on different cores that it loads, which is equal to the height of the bundle. Since we consider a sporadic task system, the first bundle in a task can arrive at any time. Each subsequent bundle in the same task is activated after the previous bundle finishes, synchronously with the time slots. Due to the asynchronous nature of the arrival of the first bundle, it is possible that it arrives in the middle of a slot, in which case it cannot start loading until the next slot, even if there are free partitions. In the worst-case, a task might arrive ε time after the

beginning of a slot, effectively wasting it. Therefore, we account for an extra σ of blocking time for each task due to the asynchronous arrival of its first bundle. For example, consider the arrival of bundle $\tau_{4,1}$; this bundle arrives before time 0 when all cores are idle, but it has to wait until time 0 to start loading.

We also account for an extra delay of σ time units for each bundle to load before it can be executed, hence $\sigma \cdot b_k$ in total for a task τ_k with b_k bundles. This is necessary because bundles of the same task cannot overlap (load a bundle while the previous bundle of the same task is executing), as there might be data interdependence between them, similarly to the multi-segment tasks discussed in Section 3.2.2. This case is depicted in the figure with bundle $\tau_{3,2}$. Specifically, the release of $\tau_{3,2}$ is aligned with the finish time of the previous bundle $\tau_{3,1}$, which is in turn aligned with the time slot at time 3. Therefore, the scheduler selects the next bundle $\tau_{3,2}$ for loading immediately at time 3 to be able to execute it one slot later at time 4. Note that since no bundle of τ_3 can run in the interval $[3, 4]$, the scheduler can instead execute a lower priority bundle, $\tau_{4,1}$ in this case.

Note that, although $\tau_{5,1}$ arrives in the same time slot as $\tau_{4,1}$ and enough partitions are available for all of them, at time 0 the scheduler only loads the highest-priority bundle that can run in the next slot, $\tau_{4,1}$ in this case. Otherwise, $\tau_{5,1}$ could have blocked $\tau_{3,1}$ for one extra slot. Therefore, in this scheduling scheme, higher-priority bundles cannot suffer blocking from lower-priority bundles.

Obviously, a bundle under analysis $\tau_{k,p}$ still suffers interference due to preemption from higher-priority bundles. Note that after being preempted, the bundle under analysis might need to be reloaded. This happens when the bundle under analysis is evicted from the local memory partitions to schedule other higher-priority bundles or lower-priority bundles that can run in parallel with the higher-priority bundles; this is the case of bundle $\tau_{3,2}$ in the figure, which is unloaded once preempted at time 5, and then resumes execution at time 9. However, the system scheduler, as discussed in Section 7.1.1, is assumed to know the WCETs of all bundles. Consequently, we can hide the delay of reloading preempted tasks by reloading the preempted bundle at the beginning of the last time slot where interfering bundles run. This is possible since there must be at least m free memory partitions or occupied by lower-priority bundles during that time slot, as there will be no more pending higher-priority bundles that can interfere with $\tau_{k,p}$. If the higher-priority interfering bundles execute for less than their WCETs, it is still guaranteed that the preempted bundle can be reloaded in the expected time slot or before; hence, in either cases, the reload time is completely hidden. Observably, the lower-priority bundles cannot block the preempted bundle. The reason is that the system is occupied by higher-priority bundles and only lower-priority bundles of height less than $h_{k,p}$ can run during the preemption window. Once $h_{k,p}$ partitions become free from higher-priority bundles, $\tau_{k,p}$ is scheduled to reload.

For the example in Figure 7.13, the decision to reload $\tau_{3,2}$ is taken at time 8, since the scheduler knows that the interfering higher-priority bundle $\tau_{1,1}$ will finishes by time 9. Other cases of preemption that do not incur reloading are the preemptions of $\tau_{4,1}$ and $\tau_{5,1}$, as they are not evicted from their partitions.

Once a bundle finishes execution, it is guaranteed to unload in the next time slot. This is because we assume that the size of the time slot is enough to both unload and load m partitions. By employing eager DMA unload, meaning that any finished task is unloaded right away, there will be no more than m partition to unload at any giving time slot. Hence, any task suffers an additional σ delay to write back its data to main memory after finishing execution.

We can now show how to extend the response time analysis in Section 7.2 to account for the discussed synchronous deferred preemption model. First, we substitute the original bundle lengths $l_{k,p}$ with the extended lengths $\overline{l_{k,p}}$ defined in Equation 7.24. Then, Equation 7.25 determines the worst-case blocking time B_k a task under analysis τ_k can suffer due to asynchronous sporadic arrival, and DMA load times for each bundle:

$$B_k = \sigma + b_k \cdot \sigma. \tag{7.25}$$

Now we can construct an upper bound on the response time of the task under analysis τ_k (based on when it finishes executing) as in Equation 7.26, where $\bar{I}_k(R_k - \sigma)$ is the bounded interference computed by Listing 7.1, and $\overline{L_k} = \sum_{p=1}^{b_k} \overline{l_{k,p}}$:

$$R_k \leftarrow \overline{L_k} + \bar{I}_k(R_k - \sigma) + B_k. \tag{7.26}$$

Note that we subtract σ from the length of the interfering window $R_k - \sigma$ used by Listing 7.1 since the task under analysis cannot be preempted during its last execution slot; this is similar to the approach in Chapters 3 and 5, where we subtract the execution time of the task under analysis from the interference window as it runs non-preemptively. Finally, to obtain the response time of $\tau_{k,p}$ including the time to write back its data, we simply sum an extra term σ to the fixed-point R_k of Equation 7.26.

7.5 Summary

In this chapter, we introduced a novel task model for parallel real-time applications. Traditional real-time parallel models schedule threads of the same task independently, but this can greatly increase the synchronization and intra-task communication overhead. In contrast, we argue that parallel real-time tasks should be gang-scheduled, so that all threads of the same task are required to be executed concurrently.

While there exist previous work on gang scheduling of real-time tasks, the rigid gang model assumes that the number of threads remains constant for the entire task execution, which is not the case for many applications. Instead, our new bundled model allows to change the number of parallel threads of an application during its execution, which are gang scheduled on an identical multiprocessor. We derived a sufficient schedulability analysis for the proposed model, and demonstrated its applicability after assigning distinct priorities to each bundle based on its characteristics.

The proposed schedulability analysis assumes preemptive scheduling, similar to previous work. However, in practice the overheads of preemption and access to main memory can be significant. For this reason, we also showed how the model can be adapted to 3-phase tasks, based on a simple system model where the size of a DMA slot is sufficient to reload all cores.

We identify four main directions for future work. First, the model assumes a fixed sequence of bundles; it could be further generalized to conditional tasks [109] where multiple different sequences of bundles are possible at run-time. Second, we plan to study how to further optimize the priority assignment. Third, as discussed in Section 7.2.5, minimizing the workload of carry-in tasks is left for future investigation. Fourth, the proposed solution for 3-phase tasks does not scale for large number of cores, given the requirement of reloading all cores within the fixed time slot. Hence, we plan to study a more efficient scheme that schedules DMA operations of multiple cores, along the lines of the solution introduced in Chapter 5.

Chapter 8

Predictable Inter-core Communication

In this chapter, we present a predictable interconnect between cores to facilitate intra-task communication for parallel tasks. Aligning with our design philosophy, we are interested in reducing hardware complexity. In particular, we propose HopliteRT, a simple FPGA-based NoC architecture that inter-connects the private SPMs. Figure 8.1 shows a simplified illustration of the proposed system architecture. Note that we still use DMA to load tasks from main memory using the memory bus as discussed in the previous chapters. The proposed SPM inter-connect NoC is limited for inter-core communication only.

To simplify the hardware implementation and improve timing predictability, we propose to adopt a push-based mechanism for intra-task communication. To adapt the push-based mechanism to a shared-memory semantic, we program the SPM controllers to push any write operation to all relevant cores based on the execution context that we know from the scheduling level. Specifically, we assume a release memory model [113] to simplify the hardware design, Further, we assume that the hardware is weakly ordered in which it appears sequentially consistent only to a software that obeys the synchronization model. Therefore, a programmer has to use the concept of mutual exclusion (critical section) to ensure correct functionality, since the total order of memory operations is not guaranteed by the memory model adopted by the hardware. In particular, upon scheduling a parallel task on specific cores, the local SPM-controllers are programmed to push (forward) write operations to all sharers. Note that only write operations inside a critical section are pushed. Exiting the critical section waits for last write operation to reach to its destinations. Compared to a traditional pull-based mechanism using a snooping cache coherence interconnect, the proposed approach results in simpler hardware implementation and derivation of latency

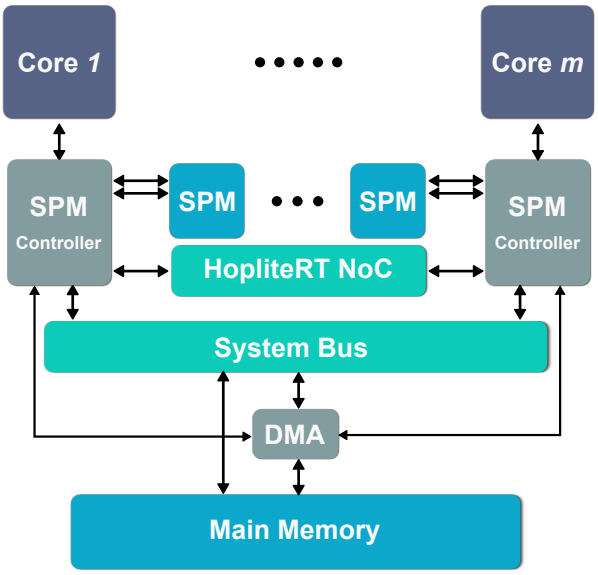


Figure 8.1: System architecture with the proposed predictable interconnect between private SPMs.

bound for communication among cores. However, as discussed in Section 2.2.2, it comes at the cost of a more restricted programming model.

It is worth mentioning that for the sake of bounding the communication time between parallel threads of a parallel task, it suffices to plug any real-time interconnect with provable latency bounds in place of HopliteRT, such as any real-time NoC. This would equivalently meet our objective in composing the worst-case computation time and the worst-case communication time to derive the total WCET of the parallel task. However, aligning with our design philosophy, we are interested in reducing hardware cost, especially since HopliteRT is only targeting inter-core communications, while communicating with main memory is actually done through the system bus as shown in Figure 8.1. Therefore, we are introducing HopliteRT a lean and efficient NoC targeting FPGA platforms.

Note that compared to the One-Way Shared Memory [144] reviewed in Chapter 2, HopliteRT uses a simpler hardware architecture. Specifically, [144] implements bidirectional torus to improve the total bandwidth at the cost of more complex switch architecture. While the analysis in [144] is not novel, the main contribution is at the Network Interface (NI) level that continuously synchronizes local memories with the network at the cost of less energy efficiency for data that are not actually shared, which also has a negative impact on memory space utilization. In additions, the addressing mechanism adopted by the

NI is not scalable with the local memory size. Specifically, the read and write addresses are implicitly embedded into the TDM slot number, which will lead to a very long hyper period for larger memory sizes impacting the latency bounds.

In this work, we present a comprehensive solution that includes task and execution models plus predictable inter-core communication for parallel tasks. Note that, our inter-core communication model is expected to be more energy efficient compared to [144], as the communication is only activated on demand. Moreover, we support arbitrary local memory size at the cost of larger network payload (packet size), which is more scalable than [144]. Unlike [144], in HopliteRT, we bound the communication latency based on aggregated traffic curves.

Similar to Section 4.1, we consider FPGA as a viable prototyping platform to quickly assess and alter our design parameters. In addition, FPGAs are actually a suitable platform for deploying real-time systems. In detail, FPGA provides several interesting properties. First, its ability to be reprogrammed with new configurations makes it a viable hardware development platform to reduce the development-evaluation cycle of hardware designs. Second, the reconfigurability property of FPGA made it well suited for the modern heterogeneous embedded systems. For example, the FPGA can be used as hardware accelerators for speeding up specific functions. In addition, different hardware functions can be configured on demand leading to a better hardware utilization. In real-time domain, the dynamic management of FPGA functions, at both the application and OS levels, has received good attentions [130, 42, 154]. Moreover, for system deployments, FPGA is considered as low-cost alternative to Application Specific Integrated Circuit (ASIC), especially for low deployment volumes. Actually, NASA utilizes FPGAs to deploy certain real-time applications in their space programs [82].

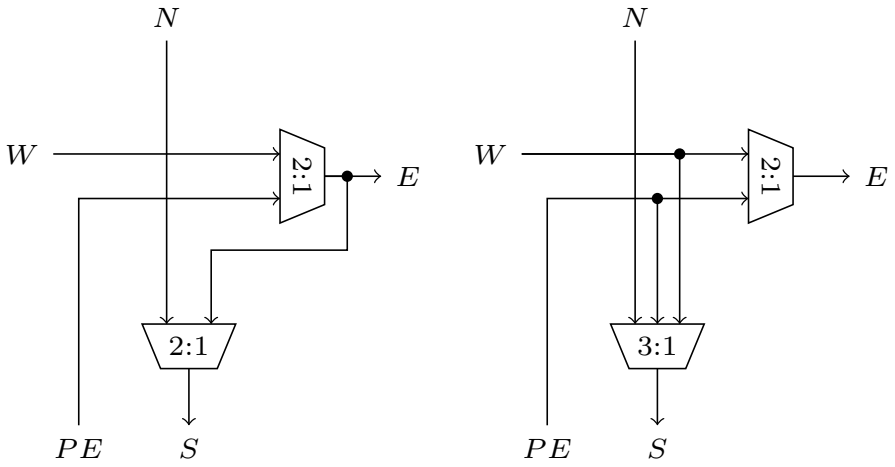
The rest of the chapter is organized as follows. We first discuss HopliteRT generally, assuming a known set of communication flows between source and destination clients (cores). In particular, Sections 8.1 and 8.2 provide background on FPGA overlay NoCs and the existing Hoplite switching architecture; Sections 8.3 and 8.4 discuss the new HopliteRT design for real-time systems and provide bounds on maximum in-flight and injection latency; and Section 8.5 evaluates the derived bounds under various workloads. Then, in Section 8.6 we discuss how to employ HopliteRT for intra-task communication of bundled task, according to the model described in Chapter 7. Finally, Section 8.7 provides concluding remarks.

8.1 The Case for FPGA Overlay NoCs

FPGA overlay NoCs (network-on-chip) such as Hoplite [79] provide a low-cost, high-throughput implementation of communication networks on the FPGA chip. NoCs allow designers to compose large-scale multi-processor, or multi-IP digital systems while providing a standard communication interface for interaction. This trend is true in the embedded, real-time computing domain with multi-core chips supported by message-passing NoCs [122, 35]. Modern real-time applications [87, 74] require many communicating processing elements to cooperate on executing the task at hand. In contrast, with shared memory systems that rely on cache coherency, explicit message passing on a suitably-designed NoC allows real-time applications to have (1) deterministic bounds on memory access, and (2) energy-efficient transport of data within the chip. FPGAs should be a particularly attractive target for the real-time computing market due to the small shipment volumes of real-time products, and the ability to deliver precise timing guarantees that are desirable for certification and correct, safe operation of real-time systems.

Until recently, FPGA-based NoCs were inefficient and bloated due to the implementation cost of switching multiplexers and FIFO queues. Hoplite [79] uses deflection routing to (1) reduce the LUT mapping cost of the switching crossbar, and (2) remove queues from the router that are particularly expensive on FPGAs. When compared to other FPGA NoCs such as CMU Connect, and Penn Split-Merge routers, Hoplite is lean, fast, and scalable to 1000s of routers on the same FPGA chip. Deflection routing resolves conflicts in the network by intentionally mis-routing one of the contending packets. In baseline Hoplite, these deflections can go on forever resulting in livelocks. This makes it unsuitable to use this NoC in a real-time environment where bounds on routing time must be computable to meet strict timing deadlines imposed by the application. An age-based routing scheme for resolving livelocks in deflection-routed networks does exist, but this requires extra wiring cost for transporting age bits and still does not deliver strict upper bounds that we are able to show in this work.

In this chapter, we answer the following question: **Can we modify Hoplite for real-time applications to deliver strong, deterministic upper bounds on worst-case routing latency including waiting time at the client?** We introduce Hoplite-RT (Hoplite with Real-Time extensions) that requires no extra LUT resources in the router and only two cheap counters in the client/processing element. These modification implement the new routing and packet regulation policy in the NoC system. The router modification has a zero cost overhead over baseline Hoplite due to a suitable encoding of multiplexer select signals that drive the switching crossbars. With the counters at the client injection port, we can enforce a token bucket injection policy that controls the burstiness and



(a) Shared East exit (b) Without sharing

Figure 8.2: Implementation choices for the Hoplite FPGA NoC Router. A LUT-economical version (left) is able to exploit fracturable Xilinx 6-LUTs to fit both 2:1 muxes into a single 6-LUT. The larger, higher-bandwidth version (right) needs 2 6-LUTs instead as the number of common inputs is lower than required to allow fracturing.

throughput of the various packet flows in the system. With these hardware modifications, we compute bounds on two components of the packet latency (1) in-flight packet routing time on the NoC, and (2) waiting time at the client or PE ports. In the next sections, we first recap the architecture of Hoplite, and then discuss our modifications and implementation on a Xilinx Virtex-7 FPGA.

8.2 Background: Hoplite NoC Architecture

In this section, we introduce the Hoplite router switching architecture and identify why the worst-case deflection and source queueing latencies can be unbounded.

Hoplite routes single-flit packets over a switched communication network using Dimension Ordered Routing (DOR). DOR policy makes packets traverse in the X-ring (horizontal) first followed by the Y-ring (vertical). Hoplite uses bufferless deflection routing and a unidirectional torus topology to save on FPGA implementation cost. While DOR is not strictly required for deflection-routed switches, Hoplite includes this feature to reduce switching cost by eliminating certain turns in the router. The internal microarchitecture

of the router is shown in Figure 8.2 with three inputs N (North), W (West) and PE (processing element or client, used interchangeably in the text) and two outputs S (South + processor exit) and E (East). When packets contend for the same exit port, one of them is intentionally mis-routed along an undesirable direction to avoid the need for buffering. The fractured implementation (Figure 8.2-a) serializes the multiplexing decisions to enable a compact single Xilinx 6-LUT realization of the switching crossbar per bit. However, this sacrifices the routing freedom to achieve this low cost. A larger design (Figure 8.2-b) that needs 2 6-LUTs per bit (2× more cost) permits a greater bandwidth through the switches. The larger cost is due to the inability to share enough inputs that would have allowed fracturing the Xilinx 6-LUT into dual 5-LUTs. This larger design will become important later in Section 8.3 when considering the microarchitecture of HopliteRT.

When considering the routing function capabilities of the Hoplite router, we make the following observations:

- The PE port has lower priority than the network ports N and W resulting in waiting time for packet injections. A client can get dominated by traffic from other clients potentially blocking it forever. This is the source of unbounded waiting time at the client injection and called **source queueing latency**.
- The N packet only travels S and no path to E is permitted under DOR routing rules. Thus N packets can never be mis-routed (or deflected) and are guaranteed unimpeded delivery to their destination.
- As a consequence, conflicts and deflections may only happen on a $W \rightarrow S$ packet which is attempting to turn. A deflected packet must route around the entire ring in the network before attempting a turn again.
- Due to the static priority for $N \rightarrow S$ packets, an unlucky $W \rightarrow S$ packet may deflect endlessly in the ring and livelock and result in unbounded NoC **in-flight routing time**. This scenario is shown in Figure 8.3.
- Deflections also steal bandwidth away from PEs in the ring, and add more waiting time penalty for packets wishing to use the PE exit.

To summarize, the communication latency between two PEs at (X_1, Y_1) and (X_2, Y_2) on the zero-load network is $T^s + (\Delta X + 1) + (\Delta Y + 1)$ due to DOR policy. Here, T^s is the waiting time at PE level, $(\Delta X + 1) + (\Delta Y + 1)$ are the number of steps in the X-ring and the Y-ring respectively. **The worst-case in-flight routing and source queueing latencies in Hoplite are both** ∞. This is problematic for real-time applications that need guaranteed bounds on application execution. If such an application wishes to use the baseline Hoplite NoC for interacting with other components, one of the packets may get victimized and deflect endlessly or a client port may get blocked forever. As a result, the real-time application will miss its deadline and violate the system design requirements. For

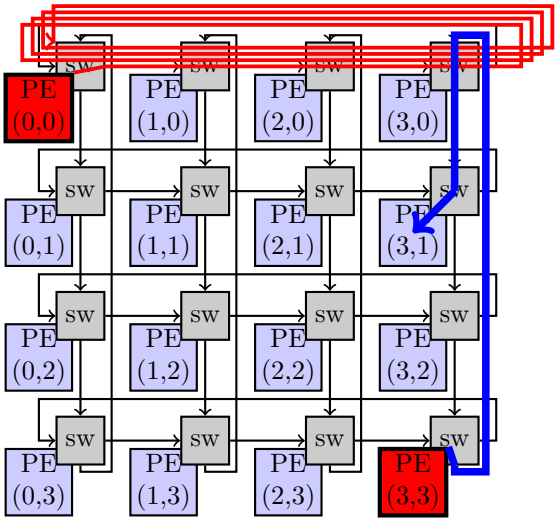


Figure 8.3: Endless deflection scenario where red packets from (0,0) → (3,3) are perpetually deflected by blue packets from (3,3) → (3,1). The red spaghetti is the flight path of one packet that gets trapped in the top-most ring of the NoC and never gets a chance to exit due to the bossy blue packets.

safety critical applications (hard real-time applications) the deadlines are hard constraints that cannot be violated. For NoCs in general, it is possible to provide statistical guarantees on packet delivery times but these are still not strong enough for hard real-time problems.

The next two sections proposes minimal modifications to Hoplite and Client to provide exact upper bounds of the worst-case latency of packet routing, and the worst-case waiting time at the source. These bounds can then be used by the real-time application developer to satisfy the hard deadline constraints imposed by the system.

8.3 Managing In-Flight Deflections

In this section, we describe the modifications to the Hoplite router to support bounded deflections in the network. We explain the resulting routing table modifications that are required and explain the operation of the NoC with an example.

In Figure 8.4, we show the proposed microarchitecture of HopliteRT. The key insight here is the need to strategically introduce deflection freedom by making the switching

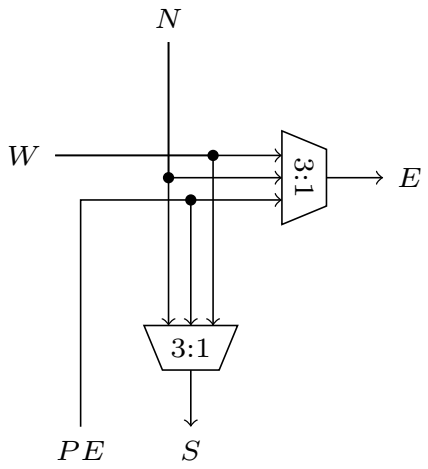


Figure 8.4: Proposed routing function arrangement for bounded in-flight latency. Despite splitting the logic into 2× 5-LUTs (3:1 muxes), the same multiplexer select signals (with different interpretation) drive both multiplexers. This allows a compact 6-LUT implementation per bit.

crossbar more capable without sacrificing LUTs in return. The fractured LUT dual 2:1 LUT implementation in Figure 8.2-a is too restrictive to permit any adaptations to the routing policy, and hence we consider a more capable dual 3:1 mux microarchitecture instead that may need the more expensive two 6-LUT implementations. This 3-input, 2-output crossbar arrangement permits any input to be routed to either output as desired by the routing policy. Thus, unlike the original Hoplite designs in Figure 8.2, we are able to use a $N \rightarrow E$ turn now to support our goals.

With this rich switching crossbar, we must now choose our routing policy. An age-based routing prioritization (Oldest-First [114]) can be implemented that prefers older packets over newer packets when in conflict. This is a good use of the $N \rightarrow E$ path which is exercised if $W \rightarrow S$ packet is older in age than the $N \rightarrow S$ packet. In this conflict situation the N packet can be deflected E. The original ∞ bound will be reduced to a different bound that depends on the network size $m \times m$ and congestion or load in the system. However, this policy is unfair as it is biased towards traffic travelling from distant nodes as traffic from nearby closer nodes is always victimized. A variation of the policy that increments age only on deflections may be slightly fairer but the resulting bound is still dependent on network congestion. Furthermore, we need to transmit extra bits in each packet to record age of the packet which is wasteful of precious interconnect resources.

It turns out that we can limit the number of deflections without carrying any extra information in the packet. The key idea is to invert the priorities of the router to always prefer W traffic over N traffic. This is the exact opposite of the original Hoplite DOR policy that always prioritizes N traffic (due to the absence of the $N \rightarrow E$ link). This modified policy allows traffic on the X-ring to be conflict free, even when making the turn to the Y-ring. On the other hand, it is the Y-ring traffic (North) that can suffer deflections. Once the Y-ring traffic has been deflected onto the X-ring, it will have a higher priority over any other Y-ring traffic it will encounter next. Thus, unlike the original design where packets deflects multiple times on the same row without making progress, now the deflected packet is guaranteed to make progress toward its destination. A packet might deflect only once on each row (X-ring).

The routing table for this new HopliteRT router is shown in Table 8.1. The added benefit of this routing policy is the ability to use the exact same select bits for both 3:1 multiplexers. So not only do we bound deflections in the NoC, but we also ensure we can retain fracturability of the 6-LUT by supplying identical five inputs to the 3:1 mux. Each mux interprets the same two select bits differently to implement the proper routing decision. In this arrangement, the PE→E with W→S turns happening in the same cycle are not supported even though the mux bandwidth is rich enough to support this condition. This is because we want to avoid creating a third mux select signal that would prevent the fractured 6-LUT mapping. If the developer can afford to double the cost of their NoC, then this extra function can be supported without affecting the in-flight NoC worst-case routing time bounds computed in this paper. It is important to note that the bandwidth capability of the HopliteRT switch is in-between Figure 8.2-a and Figure 8.2-b. When compared to Figure 8.2-a, HopliteRT supports an extra routing condition PE→S + W→E which is otherwise blocked due to cascading of the muxes. However, unlike Figure 8.2-b, HopliteRT does not support PE→E + W→S condition. Thus, HopliteRT is strictly the same size as a less-capable switch in Figure 8.2-a while offering extra bandwidth and latency guarantees.

In Figure 8.5, we show the path taken by the packet from (0,0)→(3,3) as the example earlier in Figure 8.3. In this scenario, we assume there are $W \rightarrow S$ flows in X-ring 0, 1, and 2 that will interfere with the red packets in each ring. These flows will have priority over the $N \rightarrow S$ red packet and will deflect those red packets in each ring. The blue packet from (3,3)→(3,1) had the right of way in the original Hoplite NoC as shown in Figure 8.3. In HopliteRT, it will be deflected once in the topmost ring, and then descend downwards to exit at its destination.

Analytic In-Flight Latency Bound: Under the proposed HopliteRT policy, the in-flight latency between nodes (X_1, Y_1) and (X_2, Y_2) on an $m \times m$ NoC is as follows:

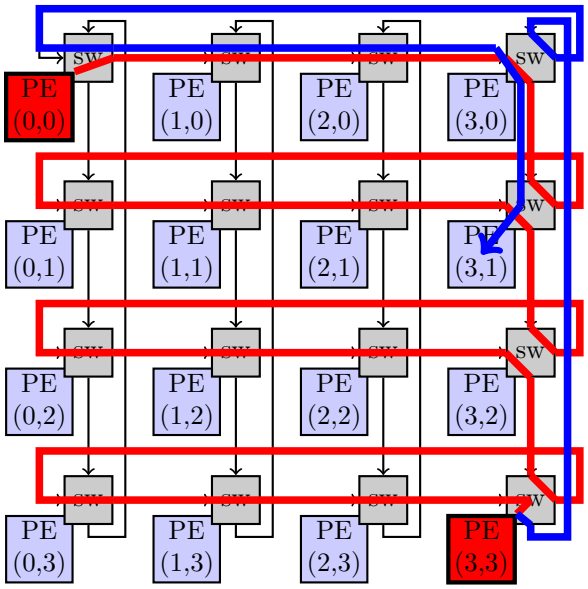


Figure 8.5: Worst-Case path on Hoplite-RT for packet traversing from top-left PE (0,0) to bottom-right PE (3,3). The red packets will deflect N→E in each ring due to a conflicting flow (not shown). The blue packets previously had priority are now deflected in the top-most ring before delivery.

Table 8.1: Routing Function Table to support Real-Time extensions to Hoplite. PE injection has lowest priority and will stall on conflict. PE→E + W→S is not supported to avoid an extra select signal driving the multiplexers and doubling LUT cost by preventing fracturing a 6-LUT into 2×5-LUTs.

Mux select		**Routes**	**Explanation**
sel1	**sel0**		
0	0	W→E + N→S	Non-interfering
0	1	W→S + N→E	Conflict over S
1	0	PE→E + N→S	No W packet
1	1	PE→S + W→E	No N packet

$$\textit{zero load: } T^f = \Delta X + \Delta Y + 2 \tag{8.1}$$

$$\textit{worst case: } T^f = \Delta X + \Delta Y + (\Delta Y \times m) + 2 \tag{8.2}$$

Here, $\Delta X = (X_2 - X_1 + m)\%m$ and $\Delta Y = (Y_2 - Y_1 + m)\%m$ are based on traversal distances of the packet on the torus irrespective of relative order of the two nodes along the directional topology. The zero load latency on the NoC is the same as original Hoplite.

Theorem 8.1. *The in-flight latency of the HopliteRT is upper bounded by Equation 8.2*

Proof. As described in Section 8.3, HopliteRT is engineered with a policy to prioritize the traffic turning from the X-ring to the Y-ring. This policy provides guarantees that a packet will be able to progress down on the Y-ring and cannot deflect more than once at every row. In the worst case, a packet can be deflected on every router during its journey on the Y-ring to the destination which is captured by Equation 8.2. □

Finally, in Table 8.2, we show the effect of compiling Hoplite and HopliteRT to the Xilinx Virtex-7 485T FPGA and observe a minor 4% reduction in LUT costs as (1) the design is able to exploit the fracturable dual 5-LUTs per bit of the switching crossbar, and (2) the DOR routing function is marginally simpler.

Table 8.2: FPGA costs for 64b router (4×4 NoC) with Vivado 2016.4 (Default settings) + Virtex-7 485T FPGA

Router	LUTs	FFs	Period (ns)
Hoplite	89	149	1.29 ns
HopliteRT	86	146	1.22 ns

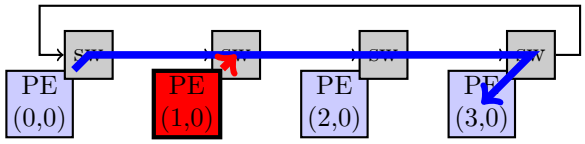


Figure 8.6: Unlucky client at (1,0) is swamped by client at (0,0) that has flooded the NoC with packets at full link bandwidth (one packet per cycle). Red packets from (0,0)→(x,y) are perpetually blocking the client exit at (1,0). This results in a waiting time of ∞ for packets at (1,0)

8.4 Regulating Traffic Injection

While the HopliteRT modification to the original Hoplite router was able to bound in-flight NoC latency, source queueing delays can still be unbounded. Source queueing delays are attributed to the least priority assigned to client injection port by the Hoplite router. This is unavoidable in a bufferless setup where we do not have any place to store a packet that may get displaced if the client port was prioritized. Furthermore, when using deflection routing, there is no mechanism to distribute congestion information to upstream clients to throttle their injection. Hence, a client may wait arbitrarily long if it is swamped by another upstream client that has decided to flood the NoC with packets; a simple scenario is depicted in Figure 8.6. Here, we have two flows: the blue flow from (0,0)→(3,0), and a red flow from (1,0)→(x,0). If the rate of the blue flow is 1 (one packet per cycle), the red flow will never get an opportunity to get onto the NoC. This is the cause of the ∞ waiting time at a client port.

In this section, we discuss a discipline for regulating traffic injection that ensures bounded source queueing times in the bufferless, deflection-routed NoC such as Hoplite. The underlying idea of regulation has been explored before in Kalray MPPA [52] but that relies on source routing, statically determining NoC packet paths, and queueing in the NoC and Client to deliver these deadlines.

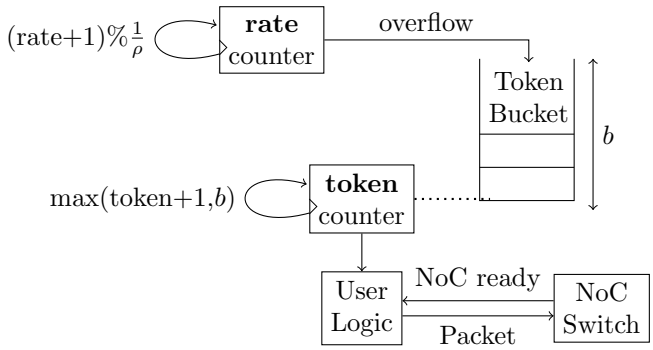


Figure 8.7: Conceptual view of the Token Bucket regulator at the injection port of each NoC client. FPGA implementation cost is two cascaded counters (no actual memory is needed to store any tokens). The client can inject a packet into the NoC only when the NoC is ready and there is at least one token available in the Token Bucket.

8.4.1 Token Bucket Regulator

In Figure 8.7, we show a high-level conceptual view of a Token Bucket regulator [36] inserted on the client→NoC interface. The regulator shapes the traffic entering the network and bounds the maximum amount of time the client (user logic) will have to wait to send a packet over the network. The regulator is defined by parameters ρ (rate of packet injection <1 packet/cycle) and b (maximum burst of consecutive packets ≥1). While conventional token bucket implementations in large-scale network switches (*i.e.*internet) need to queue or drop packets, we assume that we are able to directly stall or throttle the source (client).

The regulator is implemented using two counters that are cheap to realize on an FPGA. A free-running **rate counter** is programmed to overflow after the regulation period $\frac{1}{\rho}$. On each overflow a token is added to an abstract bucket. The bucket will fill until it reaches its capacity b where is saturates. This is tracked by the second counter **token counter**. Whenever a client wants to inject traffic into the NoC, the client is stalled until the bucket is not empty and the NoC is ready (that is, no other NoC packet is blocking the client). The client then withdraws one token from the bucket and sends the packet.

As an example, if a client has a rate $\rho = \frac{1}{10}$, it can inject one packet every 10 cycles. The burstiness parameter b determines the maximum number of packets that the client can send consecutively, assuming that the NoC is ready. If the client has a burst length $b = 5$, and it did not send any packets for 50 cycles, it will accumulate five tokens. As a result, it can send 5 packets in a burst before the token bucket is emptied.

It is worth mentioning that, in the worst case, if the traffic in the network is not

regulated (unknown traffic) the entire network will be considered as one ring. Therefore, all the clients will have to share the bandwidth utilization $\sum \rho = 1$. In this case the aggregated injection rates for each client needs to be $1/m^2$ to guarantee injection for everybody; otherwise, a client will be starved by the upstream traffic indefinitely, and the bound on the waiting time will be unknown.

Based on the described regulation mechanism, in the next section 8.4.2 we compute the maximum time that the client requires to send a sequence of K packets, with $K \leq b$. We first bound the amount of network traffic that interferes with the injection of packets on either the East or South port. Then, we express a tight bound on the maximum amount of interference to ensure that the client is not starved. Assuming that the bound is met, we finally determine the maximum network delay for the injection of the first packet, and of any successive packet in the sequence.

8.4.2 Analysis

We consider an $m \times m$ matrix of clients (x, y). We generalize the earlier discussion in Section 8.4.1 by assuming that (x, y) can send packets to multiple other clients in the network, using a different token bucket regulator for each such client. Hence, for any client (x, y), we can define a set of flows:

$$F_{x,y} = \{(x_1, y_1, \rho_1, b_1), (x_2, y_2, \rho_2, b_2), ..., (x_n, y_n, \rho_n, b_n)\}, \tag{8.3}$$

where for flow $f_i = (x_i, y_i, \rho_i, b_i)$, the destination client is (x_i, y_i) and the token bucket parameters are ρ_i, b_i. For a flow $f \in F_{x,y}$, we use the notation $f.x, f.y, f.\rho, f.b$ to denote the horizontal, vertical coordinates and regulator parameters of the flow, respectively. We further use the notation $\lambda(t)$ to denote a **traffic curve**, that is the maximum number of packets transmitted on a port in any window of time of length t cycles. By definition of the token bucket regulator, the maximum number of packets injected by (x, y) with destination (x_i, y_i) is then:

$$\lambda_{x,y}^{b_i,\ \rho_i}(t) = \min\left(t,\ b_i + \lfloor \rho_i \cdot (t-1) \rfloor\right). \tag{8.4}$$

To determine the total traffic that could pass through a client and possibly affect its injection rate, we need to know the aggregated traffic of the different sources that can reach to the client. Consider two traffic curves λ^{b_1,ρ_1} and λ^{b_2,ρ_2}, bounding the traffic on two input ports of a node and directed to the same output port. Lemma 8.1 defines an operator $\oplus$ that combines the two traffic curves to compute a tight bound on the resulting aggregated traffic.

Lemma 8.1. *Let λ^{b_1,ρ_1} and λ^{b_2,ρ_2} bound the traffic on input ports (West, North or PE) directed to the same output port (East or South). Then the traffic on the output port is bounded by the following curve:*

$$(\lambda^{b_1,\rho_1} \oplus \lambda^{b_2,\rho_2})(t) = \min\left(t, b_1 + b_2 + \lfloor \rho_1 \cdot (t-1) \rfloor + \lfloor \rho_2 \cdot (t-1) \rfloor\right). \tag{8.5}$$

Proof. In any time window of length t, the number of packets transmitted on an output port cannot be greater than the traffic produced by the input ports; hence it holds:

$$(\lambda^{b_1,\rho_1} \oplus \lambda^{b_2,\rho_2})(t) \leq b_1 + b_2 + \lfloor \rho_1 \cdot (t-1) \rfloor + \lfloor \rho_2 \cdot (t-1) \rfloor.$$

Furthermore, since the number of packets cannot be larger than t, it also holds $(\lambda^{b_1,\rho_1} \oplus \lambda^{b_2,\rho_2})(t) \leq t$. Equation 8.5 then immediately follows. □

Note that the definition of traffic curve for a token bucket regulator, as well as the curve composition in Lemma 8.1, directly follows from the theory of network calculus [36], and Lemma 8.1 is presented here for completeness. However, the bounds on injection latency derived later in this section, and in particular Theorem 8.2, cannot be obtained using network calculus because our regulator does not buffer packets.

Now, based on Equation 8.5, the operator $\oplus$ is both commutative and associative and we are essentially combining traffic flows. Hence, for any set $\mathcal{A}$ of traffic curves $\lambda^{b,\rho}$, we write $\oplus\mathcal{A}$ to denote the aggregation of all curves in $\mathcal{A}$ based on the operator. We also write $b(\mathcal{A})$ and $\rho(\mathcal{A})$ to denote the sum of the burstiness and rate parameters, respectively, for all traffic curves in $\mathcal{A}$. Note that based on Equation 8.5, this implies:

$$\oplus\mathcal{A}(t) = \min\left(t, b(\mathcal{A}) + \sum_{\forall \lambda^{b,\rho} \in \mathcal{A}} \lfloor \rho \cdot (t-1) \rfloor\right).$$

Deriving Conflicting Flows Γ_f^C: Having defined how to aggregate traffic curves for multiple flows, the next step is to define the set of **conflicting flows**, that is, those flows that block the injection of packets at the analyzed client. In particular, we consider a given flow $f \in F_{x,y}$ for the client at (x, y), and define a set Γ_f^C of traffic curves of other flows that conflict with f.

Due to the complexity of the formal derivation of Γ_f^C, we first provide intuition on how to derive such conflicting set. We show the set of interfering flows used to compute Γ_f^C in Figure 8.8. After that, we formally the derive Γ_f^C.

- First, we do not make any assumption on the arbitration used by client (x, y) to decide which flow to serve. Hence, f can suffer self conflicts from any other flow in $F_{x,y}$ that is injected on the same port of (x, y). For example, it may conflict on the South port if $f.x = x$, that is the destination of f is on the same Y-ring as client (x, y), or East otherwise.
- Second, we need to add to Γ_f^C all flows generated by other clients that conflict with f. According to Table 8.1, if f injects packets to the East port, then it suffers conflicts from any flow ($W \rightarrow E$ or $W \rightarrow S$) arriving on the West port of (x, y). If, instead, f injects to the South port, it suffers conflicts from flows turning $W \rightarrow S$ at (x, y), as well as $N \rightarrow S$ flows arriving on the North port of (x, y) directed South.
- The set of flows going $N \rightarrow S$ is easy to determine, because flows are never deflected on a different Y-ring (it may deflect $N \rightarrow E$ but will never switch Y-rings). However, determining the set of flows on the West port is more involved, because traffic arriving on the North port of any node in the y-th X-ring can be deflected N→E on the ring itself if there is any traffic simultaneously turning W→S at the same node. Finally, if the optimization discussed in Section 8.3 is employed, where the router is modified to support PE→E + W→S routing at the cost of doubling the LUTs, the conflicting set Γ_f^C can be modified to exclude W→S traffic in the case of East port injection.

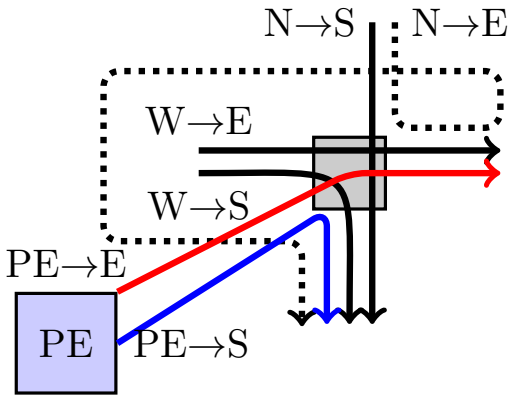


Figure 8.8: Understanding interfering traffic flows at a client for determining the set of conflicting flows Γ_f^C. Dotted $N \rightarrow E$ is a deflected flow that will wrap around the X-ring and return at the W port. The $PE \rightarrow E$ (red flow) will interfere with $W \rightarrow E$, and also $W \rightarrow S$ flow due to HopliteRT router limits. And, the $PE \rightarrow S$ flow (blue flow) will interfere with $N \rightarrow S$, $W \rightarrow S$, and the deflected $N \rightarrow E$ flows.

Now we focus on formally deriving the set of conflicting traffic curves Γ_f^C for a flow f of client (x, y), We first consider the set of conflicting flows of other clients, and then

determine the set of conflicting flows of the same client (x, y).

According to Table 8.1, in order for a client to inject on the East port there must be no traffic on the West port. Similarly, to inject on the the South port, there must be no North traffic and no West traffic turning South. Note that all North traffic goes South and does not turn East unless forcefully deflected due to a conflict on the South port. To know if a client is able to inject a packet, we need to know the total incoming traffic on each port. Equations (8.6 - 8.8) define the sets $\Gamma^{N2S}_{x,y}$, $\Gamma^{W2E}_{x,y}$, and $\Gamma^{W2S}_{x,y}$ of all traffic curves for source clients that have traffic on the North port of (x, y) aiming South, on the West port aiming East, and on the West port aiming South respectively. $\Delta Y(f.y, j) \geq \Delta Y(y, j)$ and $\Delta X(f.x, i) > \Delta X(x, i)$ mean that the traffic passes through the router (X, Y) on the Y-ring or on the X-ring respectively.

$$\Gamma^{N2S}_{x,y} = \{\lambda^{f.b,\ f.\rho}_{i,j} \mid \exists\ f \in F_{i,j}, \forall\ i, j \in [0, m) : (j \neq y) \\ \wedge (f.x = x) \wedge \Delta Y(f.y, j) \geq \Delta Y(y, j)\} \tag{8.6}$$

$$\Gamma^{W2E}_{x,y} = \{\lambda^{f.b,\ f.\rho}_{i,y} \mid \exists\ f \in F_{i,y}, \forall\ i \in [0, m) : (i \neq x) \\ \wedge \Delta X(f.x, i) > \Delta X(x, i)\} \tag{8.7}$$

$$\Gamma^{W2S}_{x,y} = \{\lambda^{f.b,\ f.\rho}_{i,y} \mid \exists\ f \in F_{i,y}, \forall\ i \in [0, m) : (i \neq x) \\ \wedge (f.x = x)\} \tag{8.8}$$

Note that the presented sets do not consider the deflection effect. In the case of a conflict on the South port, the deflected traffic on the East port will cause extra pressure on all the clients in the X-ring. Since deflection does not increase the mount of traffic going South, the traffic on the North port of (x, y) is simply $\lambda^{N2S}_{x,y} = \oplus\Gamma^{N2S}_{x,y}$. However, the same is not true for the traffic $\lambda^{W2E}_{x,y}$ and $\lambda^{W2S}_{x,y}$ on the West port of (x, y). Consider a client (i, y) on the same X-ring as (x, y): when $\lambda^{W2S}_{i,y}$ and $\lambda^{N2S}_{i,y}$ conflict on the South port of (i, y), some amount of traffic in $\lambda^{N2S}_{i,y}$ is deflected East and affects all the m clients on the y-th X-ring. If the client under analysis (x, y) wants to inject packets to the East, the deflected traffic at (i, y) thus need to be added to the set $\Gamma^{W2E}_{x,y}$ of conflicting injected traffic flows.

It remains to determine the amount of traffic of $\lambda^{N2S}_{i,y}$ that can be deflected in the general case. Clearly, the amount of deflected packets in a window of length t cannot be greater than $\lambda^{N2S}_{i,y}(t)$. Also, if there is no traffic turning West to South at (i, y), that is

if $\Gamma_{i,y}^{W2S} = \emptyset$, then no traffic can be deflected. In all other situations, we conservatively assume that all traffic produced by $\Gamma_{i,y}^{N2S}$ is deflected; even if the amount of traffic produced by $\Gamma_{i,y}^{W2S}$ is smaller than $\lambda_{i,y}^{N2S}$, deflected packets of $\lambda_{i,y}^{N2S}$ can travel around the X-ring and deflect further packets of $\lambda_{i,y}^{N2S}$ when turning W→S. Hence in the worst case, all N→S traffic will be deflected. Since the deflected traffic affects every client on the X-ring, we can thus define the set of aggregated deflected traffic Γ_y^{dtot} that goes around in the X-ring as the union of the sets $\Gamma_{i,y}^{N2S}$ for any client (i, y) such that there is traffic going W→S at that client:

$$\Gamma_y^{dtot}(t) = \cup_{\forall i \in [0,m)} \{\Gamma_{i,y}^{N2S} \mid \Gamma_{i,y}^{W2S} \neq \emptyset\}. \tag{8.9}$$

We can now compute the set of traffic curves $\Gamma_{x,y}^{CE}$ of other clients that conflict with injections on the East port at (x, y), and the set of traffic curves $\Gamma_{x,y}^{CS}$ of other clients that conflict with injections on the South port at (x, y). For $\Gamma_{x,y}^{CE}$, we combine the original traffic generated from other clients in the same X-ring, $\Gamma_{x,y}^{W2E}$ that pass to the East port and $\Gamma_{x,y}^{W2S}$ that turns south, plus the total deflected traffic on the X-ring:

$$\Gamma_{x,y}^{CE} = \Gamma_{x,y}^{W2E} \cup \Gamma_{x,y}^{W2S} \cup \Gamma_y^{dtot}. \tag{8.10}$$

For $\Gamma_{x,y}^{CS}$, we combine the original traffic generated from other clients in the same X-ring, $\Gamma_{x,y}^{W2S}$ that turns South, plus the traffic that arrives from the North port headed South, $\Gamma_{x,y}^{N2S}$:

$$\Gamma_{x,y}^{CS} = \Gamma_{x,y}^{W2S} \cup \Gamma_{x,y}^{N2S}. \tag{8.11}$$

Note that we do not consider the deflected traffic $\Gamma_y^{dtot}(t)$ as part of $\Gamma_{x,y}^{CS}(t)$, since among the deflected traffic, only the flows in $\Gamma_{x,y}^{N2S}(t)$ turn south at (x, y). If the optimization discussed in Section 8.3 is employed, where the router is modified to support PE→E + W→S routing at the cost of doubling the LUTs, then the bounds can be improved. Since the W→S traffic does not affect injection on the East port anymore, the set of conflicting traffic on the East port is now equal to:

$$\Gamma_{x,y}^{CE} = \Gamma_y^{W2E} \cup (\Gamma_y^{dtot} \setminus \Gamma_{x,y}^{N2S}). \tag{8.12}$$

Finally, we determine the set of conflicting flows of the same client (x, y). We make no assumption on the arbitration used by the client to decide which flow to serve. Hence, we consider a worst case where f is the lowest priority flow: that is, if there is any other flow at (x, y) ready to be injected to the same port (East or South) as f, then f is blocked. Based on this assumption, we simply consider all other flows in $F_{x,y}$ as conflicting flows,

similarly to traffic produced by other clients in the network. More in details, if $f.x = x$, then flow f injects to the South port at (x, y) and the set of conflicting traffic curves is:

$$\Gamma^C_f = \Gamma^{CS}_{x,y} \cup \{\lambda^{f.b,f.\rho}_{x,y} \mid \exists p \in F_{x,y}, p \neq f : p.x = x\}; \tag{8.13}$$

if instead $f.x \neq x$, then flow f injects to the East port and the set of conflicting traffic curves is:

$$\Gamma^C_f = \Gamma^{CE}_{x,y} \cup \{\lambda^{f.b,f.\rho}_{x,y} \mid \exists p \in F_{x,y}, p \neq f : p.x \neq x\}. \tag{8.14}$$

Computing Bounds based on Γ^C_f: Once the set of conflicting flows Γ^C_f has been derived, we can now study the maximum delay suffered by client (x, y) to inject a sequence of K packets of flow f, under the assumption that $K \leq f.b$. First, in the worst case, the sequence of packets can arrive at the client when the bucket has just been emptied; hence, in the worst case it will take a delay of $\lceil 1/f.\rho \rceil - 1$ before the bucket has one token. After such initial time, the packets can be further delayed by conflicting traffic in the network, which is bounded by $\oplus\Gamma^C_f$. Based on the properties of traffic curves, we first prove that the flow cannot be starved as long as the condition $\rho(\Gamma^C_f) < 1$ is satisfied. Intuitively, this means that the cumulative rate of conflicting flows is less than 1 packet/cycle; hence, eventually there will be available clock cycles when the flow can be injected on the NoC. The available rate of injection is thus $1 - \rho(\Gamma^C_f)$.

Assuming that the condition $\rho(\Gamma^C_f) < 1$ is satisfied, we then show that:

- The first packet in the sequence can suffer a delay of at most $\lceil 1/f.\rho \rceil - 1 + T^s$ cycles, where:

$$T^s = \left\lceil \frac{b(\Gamma^C_f)}{1 - \rho(\Gamma^C_f)} \right\rceil. \tag{8.15}$$

  Here, T^s represents the delay caused by the burstiness of conflicting flows; it is proportional to the cumulative burst length of conflicting flows, and inversely proportional to the available injection rate $1 - \rho(\Gamma^C_f)$.

- For each successive packet in the sequence, the client suffers an addition delay of either $1/f.\rho$ or $1/\big(1 - \rho(\Gamma^C_f)\big)$ cycles, whichever is higher. Here, the $1/f.\rho$ term represents the case where further packets are delayed by regulation, hence they are sent at the regulator rate $f.\rho$. The term $1/\big(1 - \rho(\Gamma^C_f)\big)$ represents the case where packets are delayed by conflicting flows, hence they are sent at the available injection rate of $1 - \rho(\Gamma^C_f)$. In essence, we can prove that further packets in the sequence are delayed by either regulation or conflicting traffic, but not both.

We now focus on formally deriving delay bounds for a sequence of $K \leq f.b$ packets of flow f injected by client (x, y), as intuitively described above. In any window of time of length t, by definition there must be at least $t - \oplus\Gamma_f^C(t)$ free clock cycles, that is, clock cycles where the flow is not delayed by conflicting flows. Therefore, if the flow has sufficient tokens, it can inject up to $t - \oplus\Gamma_f^C(t)$ packets. The rest of the analysis proceeds as follows. First, in Lemma 8.2 we derive a condition under which the flow is not starved, that is, it will eventually receive free cycles. Assuming that such condition holds, in Lemma 8.3 we then show that:

$$t - \oplus\Gamma_f^C(t) \geq \max\left(0, \left\lfloor \left(t - (T^s + 1)\right) \cdot \left(1 - \rho(\Gamma_f^C)\right)\right\rfloor + 1\right), \tag{8.16}$$

where T^s is defined as in Equation 8.15. This implies that the flow might receive no free cycles for T^s clock cycles (it receives one for a window of length $T^s + 1$), but is then guaranteed to receive slots at a rate of $1 - \rho(\Gamma_f^C)$. Finally, note that the flow also cannot inject packets at a rate higher than the one of its regulator, $f.\rho$. In summary, as proven in Theorem 8.2, the first packet in the sequence waits for at most $\lceil 1/f.\rho \rceil - 1 + T^s$ cycles; successive packets are sent either every $1/f.\rho$ or every $1/\left(1 - \rho(\Gamma_f^C)\right)$ cycles, whichever is higher.

Lemma 8.2. *Flow f cannot suffer starvation if $\rho(\Gamma_f^C) < 1$.*

Proof. By expanding the expression for the guaranteed number of free injection cycles $t - \oplus\Gamma_f^C(t)$ we obtain:

$$\begin{aligned} lCt - \oplus\Gamma_f^C(t) &= t - \min\left(t, b(\Gamma_f^C) + \sum_{\forall\lambda^{b,\rho}\in\Gamma_f^C} \lfloor \rho \cdot (t-1) \rfloor\right) \\ &= \max\left(0, t - b(\Gamma_f^C) - \sum_{\forall\lambda^{b,\rho}\in\Gamma_f^C} \lfloor \rho \cdot (t-1) \rfloor\right) \\ &\geq \max\left(0, t - b(\Gamma_f^C) - \rho(\Gamma_f^C) \cdot (t-1)\right) \\ &= \max\left(0, t \cdot \left(1 - \rho(\Gamma_f^C)\right) - \left(b(\Gamma_f^C) - \rho(\Gamma_f^C)\right)\right). \end{aligned} \tag{8.17}$$

Now note that $\rho(\Gamma_f^C) < 1$ implies $1 - \rho(\Gamma_f^C) > 0$; hence, the number of guaranteed free slots increases with t, meaning that the flow cannot be starved. □

Lemma 8.3. *If $\rho(\Gamma_f^C) < 1$, then Equation 8.16 holds.*

Proof. The lemma follows directly by algebraic manipulation, where the last inequality is based on Equation 8.17.

$$
\begin{aligned}
& lC \max\left(0, \left\lfloor \left(t - (T^s + 1)\right) \cdot \left(1 - \rho(\Gamma_f^C)\right) \right\rfloor + 1\right) \\
& \leq \max\left(0, \left(t - (T^s + 1)\right) \cdot \left(1 - \rho(\Gamma_f^C)\right) + 1\right) \\
& = \max\left(0, t \cdot \left(1 - \rho(\Gamma_f^C)\right) - \left((T^s + 1) \cdot \left(1 - \rho(\Gamma_f^C)\right) - 1\right)\right) \\
& = \max\left(0, t \cdot \left(1 - \rho(\Gamma_f^C)\right) - \left(\left(\left\lceil \frac{b(\Gamma_f^C)}{1 - \rho(\Gamma_f^C)} \right\rceil + 1\right)\right.\right. \qquad (8.18) \\
& \qquad \left.\left. \cdot \left(1 - \rho(\Gamma_f^C)\right) - 1\right)\right) \\
& \leq \max\left(0, t \cdot \left(1 - \rho(\Gamma_f^C)\right) - \left(\left(\frac{b(\Gamma_f^C)}{1 - \rho(\Gamma_f^C)} + 1\right)\right.\right. \qquad (8.19) \\
& \qquad \left.\left. \cdot \left(1 - \rho(\Gamma_f^C)\right) - 1\right)\right) \\
& = \max\left(0, t \cdot \left(1 - \rho(\Gamma_f^C)\right) - \left(b(\Gamma_f^C) - \rho(\Gamma_f^C)\right)\right) \leq t - \oplus\Gamma_f^C(t).
\end{aligned}
$$

□

Theorem 8.2. *Assume $\rho(\Gamma_f^C) < 1$ and the client wishes to inject a sequence of $K \leq f.b$ packets for flow f. Then the delay to inject all packets in the sequence is upper bounded by:*

$$
\lceil 1/f.\rho \rceil - 1 + T^s + \left\lceil (K - 1) \cdot \max\left(\frac{1}{f.\rho}, \frac{1}{1 - \rho(\Gamma_f^C)}\right) \right\rceil . \qquad (8.20)
$$

Proof. In the worst case, the token bucket for f can be initially empty for at most $\lceil 1/f.\rho \rceil - 1$ clock cycles. Afterwards, a new token is added to the bucket every $1/f.\rho$ cycles, at which point the next packet in the sequence becomes ready to be injected once the NoC port is free. Note that since $K \leq f.b$, the times at which the first K tokens are added, and thus the packets in the sequence become ready at the regulator, do not depend on the time at which the packets themselves are sent; this is because the bucket does not become full until the K-th token is added.

Now consider the effect of conflicting NoC traffic. Let $\lambda^{\text{free}}(t) = \max\left(0, \left\lfloor \left(t - (T^s + 1)\right) \cdot \left(1 - \rho(\Gamma_f^C)\right) \right\rfloor + 1\right)$, and consider any subsequence of i packets out of the sequence of K packets under analysis which are being delayed by NoC traffic. Since the time at which the packets become ready is fixed, the delay suffered by the last packet in the subsequence cannot be larger than both $\lceil (i - 1) \cdot (1/f.\rho) \rceil$ and $\bar{t}$, where $\bar{t}$ is the minimum window length for which $\lambda^{\text{free}}(\bar{t}) = i$ (that is, the time that it takes for the NoC to have i free

cycles based on Lemma 8.3). Based on the expression for λ^{free}, it is then trivial to see that if $1/f.\rho \geq 1/\big(1-\rho(\Gamma_f^C)\big)$, the worst case delay for the sequence is found when the first $(K-1)$ packets are sent as soon as they become ready at the regulator, while the last packet suffers NoC delay of T^s; while if $1/f.\rho \leq 1/\big(1-\rho(\Gamma_f^C)\big)$, the worst case is found where all K packets are delayed by NoC traffic rather than regulation. Combining the two cases yields Equation 8.20. □

This result, which is formally expressed by Theorem 8.2, only holds for sequences of packets of length at most equal to the burst length $f.b$ of the flow. The key intuition is that the burst length must be sufficient to allow the token bucket for f to "fill up" while the flow is being blocked by conflicting NoC traffic. In the extreme case in which $f.b = 1$, every packet in the sequence could suffer $\lceil 1/f.\rho \rceil - 1 + T^s$ delay, that is, it suffers delay due to both regulation and conflicting traffic. This reveals a fundamental trade-off for the burst length b assigned to each flow: if b is too small, then consecutive packets can be unduly delayed. However, a larger b increases the delay T^s suffered by other clients. Finally, if $\rho(\Gamma_f^C) \geq 1$, then no delay bound can be produced as the flow might be starved indefinitely by conflicting traffic. If burst lengths can be chosen freely, an assignment of burst lengths could be computed with a distributed optimization problem to minimize the worst case latencies. Formulating and solving this optimization approach is left as future work.

Revisiting the in-flight latency bound Equation 8.2 captures the analytical worst-case bound of the in-flight latency with no assumption about the traffic, e.g., the communication patterns and rates are unknown. The bound can be improved farther by leveraging the knowledge about the traffic. In particular, we can reduce the number of deflection points on the Y-ring if we know that on specific X-rings there will be no conflicting traffic. To do this we include only the X-rings that can cause conflict. Since the set of conflicting flows are now formally defined, we can update Equation 8.2, to be as in Equation 8.21. We mainly optimize the term $(\Delta Y \times m)$ to be $(V \times m)$, where $V \leq \Delta Y$ is the number of conflicting rows (x-rings) which is defined in Equation 8.22.

$$T^f = \Delta X + \Delta Y + (V \times m) + 2 \tag{8.21}$$

$$V_{x_2,y_2}^{x_1,y_1} = \sum_{j=(y_1+1)\%m}^{(y_1+1+DY)\%m} (1) \mid \Gamma_{x_2,j}^{W2S} \neq \emptyset \tag{8.22}$$

8.5 Evaluation

In this section, we show experimental validation of our bounds under various workloads, and packet flow configuration. We vary injection rates (in %), system sizes, traffic patterns, and real-time feature support and measure in-flight NoC latency and source queueing delays in the clients across our NoC. We compare the results to a baseline Hoplite NoC without real-time support. For in-flight latency tests, we evaluate synthetic traffic with 2K packets/client that exercises worst-case paths. We consider `LOCAL`, `RANDOM`, `TORNADO`, `TRANSPOSE`, and `ALLTO1` (everyone sends a packet to (0,0) client) traffic generators. For source queueing tests, we provide a tool to evaluate a user-supplied set of flows for feasibility and provide a bound where one can be proved.

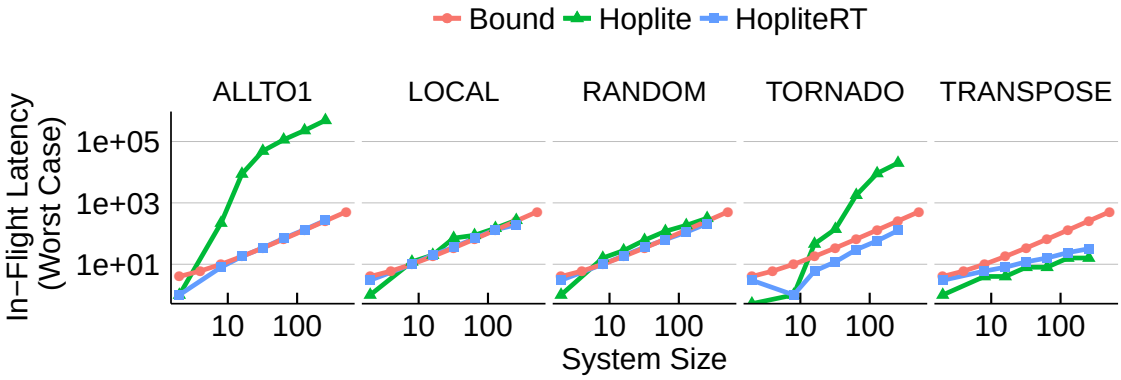


Figure 8.9: Effect of Traffic Patterns on Worst-Case In-Flight Latency of the Workload at 100% injection rate. Worst-case analytical bounds (red) are easily violated by baseline Hoplite. With HopliteRT we are always within the bound, and deliver superior worst-case latency for `ALLTO1`, `TORNADO`, `RANDOM`, and `LOCAL` patterns. For `TRANSPOSE`, the persistent victimization of $N \rightarrow S$ packets causes a slightly longer worst-case latency.

In-Flight NoC Latency Bounds: In Figure 8.9, we show the effect of using HopliteRT over the baseline design when counting the worst-case latency suffered by the packet in the NoC. It is clear that for `ALLTO1`, `TORNADO`, `RANDOM`, and to some extend for `LOCAL` patterns, the worst-case latency with our extensions has been significantly improved. This is particularly true in the adversarial case where each client sends data to a single destination. This pattern is representative of situations where a limited resource like a DRAM interface must be shared across all clients in the system. Without HopliteRT, some client requests to a DRAM interface may never route to the DRAM interface unless

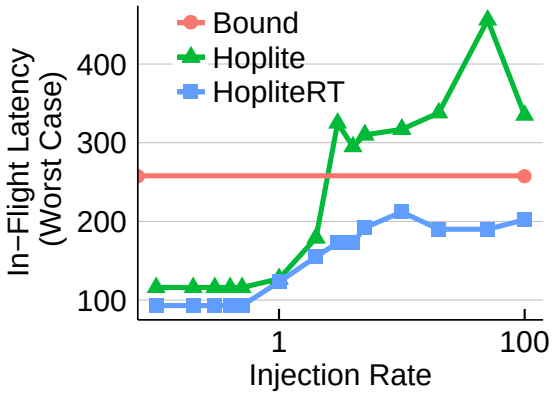


Figure 8.10: Effect of Injection Rate on Worst-Case In-Flight Latency of the `RANDOM` Workload for 256 clients. At low injection rates, the NoC routing latencies are not very different, but as the NoC gets congested, HopliteRT starts to deliver improvements.

other clients complete their requests[1]. For other traffic patterns, the benefit is less pronounced, and it gets slightly worse for `TRANSPOSE`. This is because W→S static priority victimizes $N \rightarrow S$ packets each time that abundantly occur in other workloads. It is possible to improve fairness by using an extra priority bit and history information, but that would increase the cost of the NoC router. Finally, we show that our router never violates the predicted bounds that are calculated based on our analysis and these are better than the worst-case latencies observed in baseline Hoplite in most cases. Apart from the proofs, this experimental validation supports a real-time developer in safely using these bounds during system design. In Figure 8.10, we take a closer look at a 256-client simulation and vary the injection rates from 0.1% to 100% for `RANDOM` workload. As expected, we observe that at low injection rates <10%, both networks deliver packets better than the bound. However, as network gets congested, the baseline Hoplite design delivers packets with increasing worst-case latency that exceeds the bounds. The HopliteRT design is always better than the bound at all injection rates. Our observed latencies are within 20% of the computed bound. The computed bound is consider tight, as it can be reached in some cases as in `ALLTO1` and `LOCAL`.

The improve bound on the in-flight latency is depicted in Figure 8.11. The figure shows the bound T_f based on Equation 8.2 which computes the latency statically without

[1]This behavior was demonstrated at the FCCM 2017 Demo Night where Jan Gray's GRVI-Phalanx [66] engine with 100s of RISC-V processors interconnected with Hoplite. Clients closer to the DRAM interface were effectively starved and never got service in a DRAM interface test.

considering the conflicting traffic. Whereas, $T_f.opt$ represents the improved bound based on Equation 8.21 that considers the conflicting traffic. The results show that the improved bound is especially beneficial in large networks. As shown in Figure 8.11, the optimized bound improves over the basic bound about 25% in the case of 16X16 NoC.

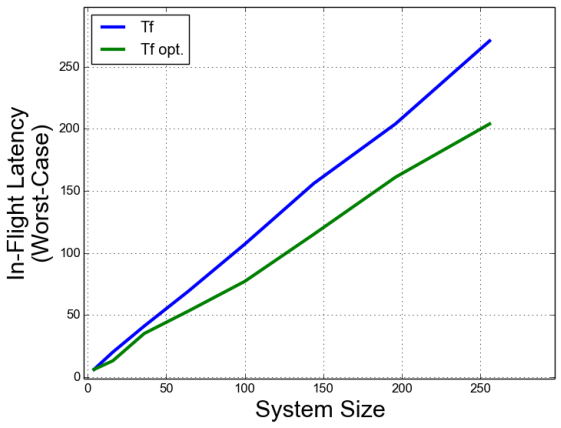


Figure 8.11: The optimized bound versus the basic one on Worst-Case In-Flight Latency of `RANDOM` workload at 100% injection rate of 256 clients

Source-Queueing Bounds: We now show the benefit of client injection regulation on source queueing delay. In these experiments, we use HopliteRT router in all comparisons and selectively enable regulation. This shows the need for regulation in addition to modification to the Hoplite router for delivering predictable, bounded routing latencies in the network. In this experiment, we set the offered rate ρ=$\frac{1}{m^2}$ ($m \times m$=size of NoC) and a burst b=1. The traffic rate is scaled to $\frac{1}{m^2}$ to ensure feasible flows to the single destination client.

In Figure 8.12, we show the effect of using our regulator on source queueing delay for traffic with the `ALLTO1` pattern. From this experiment, it is clear that simply using the HopliteRT router is insufficient and the source queueing times are large. When we add regulator hardware to the client, the waiting times drops dramatically by over four orders of magnitude. Our analytical bound tracks our observed behavior but there is a gap as it must assume pessimistic behavior from interfering clients in the calculations. We also consider `RANDOM` traffic pattern in Figure 8.13. Again, we observe better behavior with regulated traffic injection but the latencies are lower than the `ALLTO1` case. While `RANDOM` traffic is less adversarial than the `ALLTO1` pattern, our bound still holds, and regulation

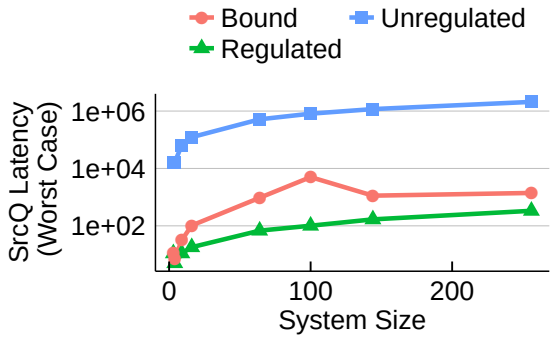


Figure 8.12: Comparing source queueing times for regulated vs. unregulated HopliteRT NoCs as a function of system size for the `ALLTO1` traffic pattern. Regulated traffic offers much improved waiting times at the clients.

is still up to two orders of magnitude lower latency. You may also note that the bound calculation is now significantly tighter than the `ALLTO1` scenario as the traffic flows now interfere is less-adversarial manner.

Finally, we show a breakdown of the bounds for the PE South and East ports in Figure 8.14. The difference in bounds is because the E port can accept more packets due to the DOR routing policy, and furthermore due to the limits of the routing configurations shown in Table 8.1. Recall, we want to avoid doubling the LUT cost of the HopliteRT router, and intentionally disallow $PE \rightarrow E$ and $W \rightarrow S$ packets to route simultaneously. These have the effect of creating a difference in bounds for traffic injection along S and E ports as shown in the figure. The E port suffers a higher waiting time as injection rate is increased as we disallow E traffic in our LUT-constrained router.

8.6 Integrating Communication Time into Execution Time of Bundled Tasks

In this section, we discuss how to bound the intra-task communication latency using HopliteRT for a parallel task scheduling according to the bundled model discussed in Chapter 7. As mentioned earlier in this chapter, to bound injection latency we rely on knowing the set of interfering flows. Once a packet is injected into the network, its routing (in-flight) latency is bounded statically regardless of interference. Although an optimized bound on

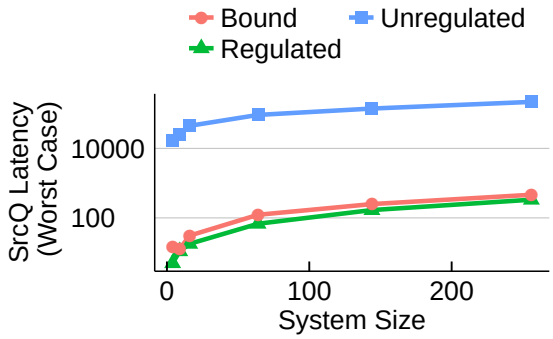


Figure 8.13: Comparing source queueing times for regulated vs. unregulated HopliteRT NoCs as a function of system size for the `RANDOM` traffic pattern. Again, regulated traffic offers better latency behavior, but bounds are much lower than the `ALLTO1` pattern.

routing (in-flight) latency is also provided based on knowing the interfering flows (Equation 8.21) , we limit our discussion in this section to the basic static bound (Equation 8.2).

In our bundled scheduling scheme, we have no assumption on which cores are assigned to a bundle, *i.e.*, no core pinning; this applies to the bundle under analysis and to the interfering bundles. This property eliminates the knowledge about the interfering flows, because each time a bundle is scheduled it can be on different set of cores. Therefore, we cannot directly apply the injection latency analysis based on conflicting flows as detailed in Section 8.4.2. Note that other scheduling schemes, such as federated scheduling [93], that promote task pinning can utilize the optimized latency analysis based on the knowledge of the conflicting flows.

To compute the injection latency according to Equation 8.20, we maximize the interfering flows based on assigned bandwidth to each application. Specifically, we assume that all nodes that are not part of a bundle under analysis have an interfering flow to some node assigned to the bundle under analysis. In other words, all other nodes are considered upstream. To ensure that the no-starvation condition $\rho(\Gamma_f^C) < 1$ holds, we can use the following bandwidth assignment: we assign a bandwidth utilization equal to $\frac{h_{k,p}}{m}$ to any currently executing bundle $\tau_{k,p}$. It is then up to the application to distribute the allocated bandwidth internally among flows originating from nodes assigned to the bundle. It remains to account for the burstiness of interfering flows, which impacts the value of the initial latency T^s in Equation 8.15.

We assume that the inter-core (inter-thread) communications within the application

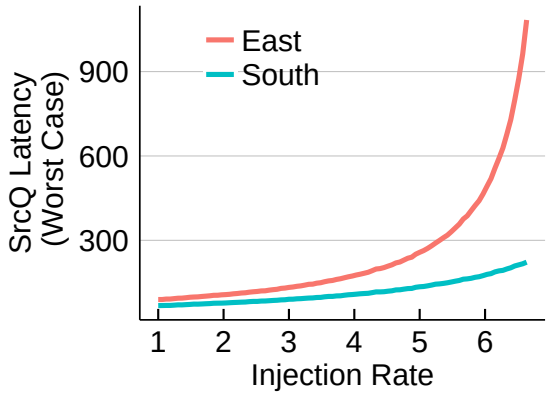


Figure 8.14: Effect of Injection Rate on Worst-Case Source-Queueing Latency of the `ALLTO1` workload for 16 clients. The E port can accept more packets due to the DOR routing policy; and furthermore be blocked by our LUT-constrained HopliteRT router. Above a certain injection rate, no bounds can be computed due to infeasible flow rates in the network.

are known. From scheduling perspective, while $\tau_{k,p}$ is running on $h_{k,p}$ cores for $l_{k,p}$ time units, other interfering workload can run in parallel on the remaining $m - h_{k,p}$ cores. Since we do not know which other bundles run in parallel at any given point in time, we determine the set of bundles with the highest burstiness according to a simple greedy algorithm. Specifically, all other interfering bundles are sorted according to their flows' total burstiness. Then, we pick one interfering bundle at a time, starting from the one with the highest burstiness, and assign it to the $m - h_{k,p}$ available cores, until there are no more core remaining. Note that in this greedy algorithm, the last assigned bundle might require a number of cores higher than the number of remaining cores; hence, the resulting assignment can be pessimistic, but it is easy to see that it upper bounds the total burstiness. In alternative, an exact algorithm could be used, but it would have a complexity equivalent to bin-packing.

Note that if an application tries to send a chunk of packets larger than its assigned burstiness b, the regulator will break it up into smaller chunks. By knowing the number of chunks an application sends during its execution, we can bound the total latency by summing the latency of each chunk of packets. Finally, we add the total communication latency to the worst case execution time (length) of the bundle under analysis.

Note that, even without knowing the set of conflicting flows, there are few special cases where the bundle under analysis will not have any external conflicting flows, hence, will

not suffer interference from other running bundles. Also, note that bundles of height one do not generate communication traffic, thus does not contribute to interference. Consider Figure 8.15, the first case **A** is when the bundle under analysis is of height m. Since no other bundle can run in parallel, there will be no external interference. The second case **B**, when the bundle under analysis is of height $m-1$. Since only single-core bundles can run in parallel, which do not generate communication traffic, there will be no external interference similar to the first case. The third case **C**, single-core bundles also do not receive communication interference as they do not use the network. In Case **A**, the worst-case communication delay can be bounded in isolation by fixing threads to specific cores and embed the pinning information into the application's scheduler. We can apply the same trick to Case **C** if we support task migration. Given the above observations, it is worth mentioning that in small network sizes, *e.g.*, ≤ 4, communication interference can be avoided totally. However, the NoC is actually meant for larger network sizes, *e.g.*, ≥ 16.

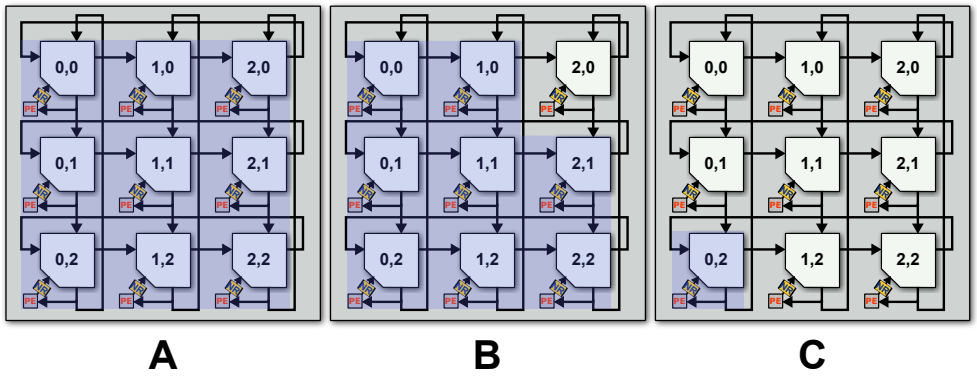


Figure 8.15: Example of three different cases of bundles that does not suffer communication interference

As discussed above, bounding the communication delay of the proposed HopliteRT NoC is highly affected by the decisions at the scheduling level. This aligns well with our design philosophy where we simplify the hardware and the analysis at the cost of more complex scheduling policies. Developing a scheduling policy with the objective to improve the bound of communication delay is in our interest and will be considered in future work. For example, to improve the communication delay we can develop a scheduling policy to assign rows (X-rings) to applications, as shown in Figure 8.16-A. This improves the delay bound, as all the traffic stays within the row and does not affect other applications on other rows, as shown in the upper row of the figure. In the case of assigning multiple rows to a bundle, as shown in the bottom two rows in the figure, all traffic in the same row or

from the upper rows to the lower rows stays withing the boundaries of the rows and does not affect other rows. However, traffic from lower rows to upper rows wraps around the Y-ring and can affect other applications on other rows.

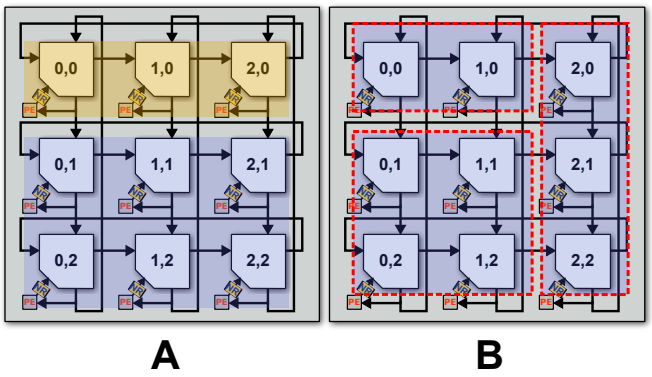


Figure 8.16: HopliteRT node scheduling (A) and network partitioning (B)

To mitigate the traffic leaking effect between unrelated applications, we suggest to add a new feature to HopliteRT router to support dynamic network partitioning. Similar approach for deflection torus NoC is proposed in [46]. The intuition is to partition the network into sub isolated networks as shown in Figure 8.16-B. We propose to group the related nodes in isolated partition where traffic does not travel outside the partition. Then, the same analysis mentioned above can be applied, but on the granularity of the partition. This optimization improve both the worst-case routing latency as the network width becomes smaller, and the injection rate (bandwidth) as the number of active nodes on any path is fewer.

Figure 8.17 shows the required extra multiplexers to implement the dynamic partitioning. In this case, the 4 to 1 mux is implemented in one 6-LUT. This configuration supports partitions as small as (1 X 2) and as large as (4 X 4). To support larger partitions, we need larger muxes that requires more LUTs to implement.

8.7 Summary

In this chapter, we presented HopliteRT, an FPGA-based NoC that provides predictable latency bounds for inter-core communication. A 64b HopliteRT router implementation delivers approximately identical LUT-FF cost (2% less) compared to the original Hoplite

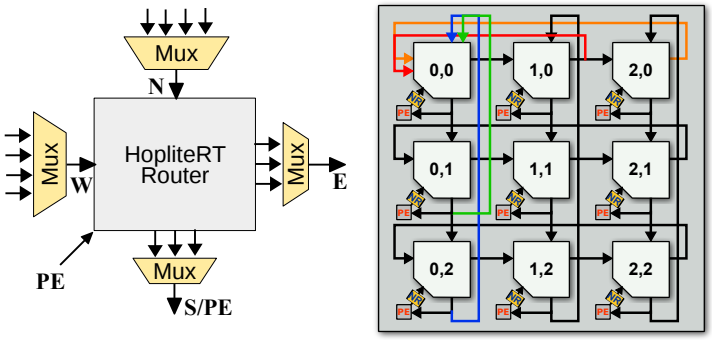


Figure 8.17: Suggested modification to the router to support dynamic partitioning

router. We also add two counters to the client interface to provide a Token Bucket regulator for controlling packet injection in a manner that bounds source queueing delay. We show the in-flight NoC routing bound to be $(\Delta X + \Delta Y + (\Delta Y \times m) + 2)$, and the source-queueing bound to be $(\lceil \frac{1}{\rho_i} \rceil - 1) + T^s : T^s = \lceil \frac{b(\Gamma_f^C)}{1-\rho(\Gamma_f^C)} \rceil$. We test HopliteRT across various statistical datasets and show that (1) our analytical bounds are relatively tight for `RANDOM` workloads, and (2) our solution provides significantly better latency behavior for `ALLTO1` workload that models shared DRAM interface access.

We then discuss how to use HopliteRT to determine latency bounds for intra-task communication based on a push mechanism. We first derive bounds based on the bundle scheduling model for parallel tasks detailed in Chapter 7. We then discuss a set of additional scheduling restrictions that would allow the derivation of better bounds. We reserve to study how to integrate such restrictions in the bundled scheduling model, in particular in conjunction with the 3-phase execution scheme in Section 7.4, as part of our future work.

Finally, note that the analysis tools for this NoC are freely available to download at [2].

[2] https://git.uwaterloo.ca/watcag-public/hoplitert-bounds.

Chapter 9

Conclusions

Scratchpad memory has long been introduced in embedded real-time platforms for its energy efficiency and predictable performance. Many works have utilized SPM for performance enhancement using static and dynamic SPM allocation at the applications level. In this thesis, we improved the usability of scratchpad memory for hard real-time systems. We introduced a system architecture and an execution model that exploit the predictability of the SPM and hide access latency to shared resources, such as main memory. An important property of the proposed solution is the dynamic management of the different components in the system at the OS level. This property allows real-time applications to be ported easily, as the complexity of the system is completely transparent to the applications. Basically, the proposed solution incorporates the SPM in a multi-tasking system without reliance on the number of tasks. This property stems from limiting the SPM partitions to two tasks only. In addition, the 3-phase execution mechanism made it easy to support inter-tasks asynchronous communication.

We augmented the proposed execution model with scheduling schemes that cover single-core, partitioned, and global multicore systems. In addition, we considered scheduling DMA access among cores in software, when the hardware does not provide predictable arbitration between cores. As the results show in our evaluation, the proposed solution significantly improves system schedulability compared to other cache and SPM-based approaches. As an outcome of this research, we argue that the use of SPM memory is a viable alternative to caches for real-time systems, providing predictable performance and simpler WCET analysis, with reasonable ease of use.

In this research, we addressed the gap between the need for predictable data sharing of parallel tasks, and the existing real-time scheduling models that assume independent

threads. We introduced bundled scheduling model that extends the classical gang scheduling to improve system schedulability. We showed how bundled scheduling can be applied to several programming models, such as fork-join and DAG-based tasks. We incorporated parallel tasks into the proposed execution model to obtain a comprehensive solution that covers both sequential and parallel tasks. To make that happen, we augmented our scheduling model of parallel tasks with predictable inter-core NoC to facilitate data sharing between parallel threads.

9.1 Limitations

In this section, we highlight a few limitations and the scope of applicability of the presented solutions. At the task execution level, the 3-phase execution model requires the task to be loaded into the local memory first before it can run from the local memory. After the task finishes its execution phase, it can be downloaded back to main memory. In comparison to the conventional hardware-managed cache-based execution model, the 3-phase execution model needs explicit software-based memory management to govern the execution of tasks. This introduces two main challenges. First the software complexity to manage local memory allocations and tasks execution. In this work, we showed how to hide this complexity by managing the entire process at the OS level and make it transparent to applications.

Second, applications need to be factored in a way compatible with the 3-phase model. Although this can be done manually for some small and simple programs, it becomes tedious in most cases. In addition, the proposed solution imposes size restriction on tasks memory footprint. Specifically, a task needs to fit entirely in the local SPM partition. Although many hard real-time tasks might fit in the more spacious local memory of modern chips, there are still applications that need to be split into smaller segments as discussed in Chapter 3.

For these reasons, researchers has proposed tools [105] that use profiling techniques for automatic refactoring of programs into 3-phase compatible format. However, there are still few challenges. For instance, some dynamic memory accesses might be missed during the profiling process leading to a need for accessing main memory during the execution phase. This behavior will introduce memory access contention between cores. To account for this contention at the analysis level, we can incorporate extra overheads in the WCETs by inflating them by a certain percentage. The inflation can be based on worst-case expectations. Similarly, based on the code and dynamic data structures, it might lead to load more data than what is actually needed in the current job execution.

On the other hand, Matejka et al. [108] proposed compiler-based technique, based on LLVM [90], to automatically transforms legacy code into 3-phase compatible code. Although, the static program analysis employed in [108] produces more efficient memory allocations than [105], it restricts the use of dynamic data that cannot be determined by the static program analysis. In general, such restrictions are in line with the requirements of typical coding standards adopted in the automotive real-time domain, such as the MISRA guidelines [22]. In addition, HePREM [62] extends the techniques in [108] to split the GPU tasks into 3-phase compatible segments.

Note that, although known methodologies exist [105, 108, 128, 145, 94, 152, 135, 100, 29] to split a large application into smaller segments that are individually compliant with the imposed constraint, not all types of applications can be efficiently segmented. Those types of applications might suffer excessive blocking due to data dependency between segments, which reduces the benefits of the proposed mechanism to hide memory access latency. For example, dynamic-data-intensive applications such database applications would have some challenges to be efficiently segmented. However, this dissertation is not focused on program analysis. Instead, we mainly focus on scheduling 3-phase tasks to provide timing isolation while hiding memory access latency. Through our work on this research, we note that porting applications to this platform can be as simple as adding few lines to the linker script to cluster tasks into consecutive blocks, as discussed in Section 4.1, to a more complex process that requires compiler support.

At the schedulability level, it is worth mentioning that the proposed scheduling scheme works best with tasks of similar sizes (execution times). This stems from the proposed non-preemptive policy, as in the worst case, one very large task can induce undue blocking time to other tasks, which reduces the schedulability of the system significantly. This effect is another reason why a task might need to be split in to segments. Specifically, a long executing task might be split into smaller execution segments to reduce the impact of blocking. However, in our experiments with real-time benchmarks most applications were configured to have relatively comparable execution times to resemble actual automotive applications with execution time less than 1 ms. For some type of applications, such as streaming applications, it is common to have similar execution times.

The standard schedulability evaluation graphs provided in this dissertation are based on simulation experiments. Although we do not claim full coverage of all possible task sets, we are confident that the presented results align with the projections of the mathematical modeling and the analytical intuitions.

Note that the evaluation of the 3-phase task execution of bundled scheduling with inter-core communication is not presented in this dissertation. Specifically, the evaluation would

require implementation of the extended platform, both on the hardware and the OS levels. Therefore, due to time limitations, it is left for future work. Furthermore, in regards to parallel tasks, the assumption that the largest bundle in the system can be loaded and unloaded in one time slot needs to be relaxed; otherwise, there will be wasted time in the slot that will manifest as unnecessarily blocking time for smaller bundles. Furthermore, the current heuristic-based priority assignment in bundled scheduling needs further study.

9.2 Future Directions

We expect to continue investigating this line of research in the future, and extend it in several directions. A key observation is that modern embedded platforms incorporate more heterogeneous components. Heterogeneous multi-core platforms embed a number of different types of processing elements, each specialized to optimize a set of functions, or providing a different trade-off between performance and energy efficiency. As such, each core may feature different ISAs, clock speeds and capabilities in terms of interaction with the rest of the system. Extending our co-scheduling and memory management scheme to such different components will impose additional challenges on both the scheduling level and analysis level.

The increased need for embedded computer vision leaded to include GPUs in modern embedded platform. Many relevant applications are memory intensive, and we anticipate that the proposed execution model can improve hiding the memory latency for such applications. In the same direction, it is interesting to note that some many-core architectures, such as Kalray[1] and Parallella [2], provide cores that are already equipped with private SPMs. In Kalary, cores are clustered in groups. The proposed mechanism, as the results show, is able to utilize the memory bandwidth efficiently; hence, it can scale to more number of cores on a single channel memory better than other compared approaches. However, to apply our approach to many-core platforms (> 64 cores), we need to investigate how to distribute or partition tasks over different memory channels to avoid saturating memory bandwidth. Otherwise, according to our scheduling scheme, all scheduling intervals will be dominated by DMA-bound intervals, and prevent the ability to hide memory access latency.

In practice, without handling of I/O data, any proposed approach will be of limited utility. While we showed how I/O data can be handled in our proposed solution, incorporating full I/O analysis for particular embedded I/O devices with details latency analysis

[1] http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf
[2] https://www.parallella.org/

is considered as a future work. We are also interested in incorporating more advanced compiler techniques to automatically convert parallel programs into bundles and generate application-level schedules accordingly. In particular, we plan to look at existing open programming standards for parallel applications, such as OpenMP, which has recently received significant attention in the real-time community.

In regards to HopliteRT for inter-core communication, an extension as suggested in Section 8.6 is beneficial. This is to simplify clustering relevant nodes that belong to the same bundle together, hence, improving worst-case communication latency.

Finally, we believe that this thesis has shown that there is ample space to improve the execution of real-time applications, given enough control over the behavior of the hardware platform. As such, we think it is crucial for embedded platforms' vendors to incorporate more fine-grain control over the hardware and allow software management of hardware resources. This will encourage researchers to investigate ways to improve the worst-case performance, enhancing the execution predictability beyond what the stock hardware-control can do.

References

[1] Cache Architecture, ECE 3055: Computer Architecture and Operating Systems. http://users.ece.gatech.edu/~dblough/3055/.

[2] MERASA Project, http://ginkgo.informatik.uni-augsburg.de/merasa-web/. http://ginkgo.informatik.uni-augsburg.de/merasa-web/.

[3] NiosII Embedded Processor. http://www.altera.com/devices/processor/nios2/ni2-index.html.

[4] parMERASA. http://www.parmerasa.eu/.

[5] Processors FAA Position Paper on Multi-ore. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/castâĂŘ32.pdf.

[6] Zynq-7000 SoC ZC706 Evaluation Kit. http://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.htm.

[7] P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. www.frescale.com, 2012.

[8] FreeRTOS. http://www.freertos.org/, 2013.

[9] RapiTime. https://www.rapitasystems.com/products/rapitime, 2013.

[10] MicroBlaze Processor Reference Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug984-vivado-microblaze-ref.pdf, 2014.

[11] ERIKA Enterprise. http://erika.tuxfamily.org/, 2016.

[12] Freescale MPC5777M. http://cache.freescale.com/files/32bit/doc/fact_sheet/mpc5777mfs.pdf, 2016.

[13] Ben Abdallah. *Multicore Systems On-Chip: Practical Software/Hardware Design*. Springer, 2013.

[14] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer (Long. Beach. Calif).*, 1996.

[15] Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proc. 14th Int. Conf. Embed. Softw. - EMSOFT '14*. ACM Press, 2014.

[16] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multi-threaded applications on multicore systems. In *Des. Autom. Test Eur. Conf. Exhib. (DATE), 2014*. IEEE Conference Publications, 2014.

[17] Ahmed Alhammad and Rodolfo Pellizzoni. Trading Cores for Memory Bandwidth in Real-Time Systems. In *2016 IEEE Real-Time Embed. Technol. Appl. Symp.* IEEE, 2016.

[18] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time Embed. Technol. Appl. Symp.* IEEE, 2015.

[19] Yannick Allard, Geoffrey Nelissen, Joel Goossens, and Dragomir Milojevic. A context aware cache controller to bridge the gap between theory and practice in real-time systems. In *2014 IEEE 20th Int. Conf. Embed. Real-Time Comput. Syst. Appl.* IEEE, 2014.

[20] Sidharta Andalam, Partha Roop, Alain Girault, and Claus Traulsen. PRET-C: A new language for programming precision timed architectures. Technical report, 2009.

[21] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 1986.

[22] Motor Industry Software Reliability Association et al. *MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013*. Motor Industry Research Association, 2013.

[23] Neil C Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer, 1991.

[24] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, and Others. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 2014.

[25] Ke Bai, Jing Lu, Aviral Shrivastava, and Bryce Holton. CMSM: an efficient and effective code management for software managed multicores. In *Proc. Ninth IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synth.*, 2013.

[26] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. *2012 IEEE Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, 2012.

[27] Theodore P Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *RTSS*, 2003.

[28] Rajeshwari Banakar, Stefan Steinke, BS Bo-Sik S Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proc. ....* ACM Press, 2002.

[29] S Bandyopadhyay, F Huining, H Patel, and E Lee. A scratchpad memory allocation scheme for dataflow models. Technical report, 2008.

[30] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, 2007.

[31] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Real-Time Syst. (ECRTS), 2015 27th Euromicro Conf.*, 2015.

[32] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A Generalized Parallel Task Model for Recurrent Real-time Processes. In *2012 IEEE 33rd Real-Time Syst. Symp.* IEEE, 2012.

[33] Sanjoy K Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Syst.*, 2006.

[34] Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proc. OSPERT*, 2010.

[35] M. Becker, B. Nikolic, D. Dasari, B. Akesson, V. Nelis, M. Behnam, and T. Nolte. Partitioning and Analysis of the Network-on-Chip on a COTS Many-Core Platform. In *proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[36] Van Bemten. Network Calculus: A Comprehensive Guide. Technical report, 2016.

[37] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 1992.

[38] Vandy Berten, Pierre Courbin, and Joël Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *5th Jr. Res. Work. Real-Time Comput.*, 2011.

[39] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Syst. Symp. 2007. RTSS 2007. 28th IEEE Int.*, 2007.

[40] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *ECRTS*, 2005.

[41] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-Time I/O Management System with COTS Peripherals. *Computers, IEEE Transactions on*, 2013.

[42] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *2016 IEEE Real-Time Syst. Symp.*, pages 1–12. IEEE, nov 2016.

[43] Björn B Brandenburg, John M Calandrino, and James H Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *RTSS*, 2008.

[44] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011.

[45] Paul Caspi and Oded Maler. From control loops to real-time programs. In *Handb. networked Embed. Control Syst.* Springer, 2005.

[46] Kumar H B Chethan, Shubham Agarwal, and Nachiket Kapre. Deflection routing for multi-level FPGA overlay NoCs. In *2016 Int. Conf. Field-Programmable Technol.* IEEE, 2016.

[47] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Inf. Process. Lett.*, 2008.

[48] Pierre Courbin, Irina Lupu, and Joël Goossens. Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks. *Real-time Syst.*, 2013.

[49] Leonardo Dagum and Rameshm Enon. OpenMP: an industry standard API for shared-memory programming. *Comput. Sci. Eng. IEEE*, 1998.

[50] Robert I Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 2011.

[51] Robert I Davis, Alan Burns, José Marinho, Vincent Nelis, Stefan M Petters, and Marko Bertogna. Global fixed priority scheduling with deferred pre-emption. In *RTCSA*, 2013.

[52] Benoˆ Dupont de Dinechin and Amaury Graillat. Network-on-chip Service Guarantees on the Kalray MPPA-256 Bostan Processor. In *Proceedings of the 2Nd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*. ACM, 2017.

[53] J.-F. Deverge and I Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *Real-Time Syst. 2007. ECRTS '07. 19th Euromicro Conf.*, 2007.

[54] J-F Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, 2007.

[55] Zheng Dong and Cong Liu. Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems. In *2017 IEEE Real-Time Syst. Symp.* IEEE, 2017.

[56] Stephen A Edwards and Edward A Lee. The case for the precision timed (PRET) machine. In *Proc. 44th Annu. Conf. Des. Autom. - DAC '07*. ACM Press, 2007.

[57] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *Proc. 6th ACM IEEE Int. Conf. Embed. Softw.* ACM, 2006.

[58] H. Falk and J. C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, 2009.

[59] Chris Fallin, Chris Craik, and Onur Mutlu. CHIPPER: A Low-complexity Bufferless Deflection Router. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 2011.

[60] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel Distrib. Comput.*, 1992.

[61] Shanna-Shaye Forbes, Hiren D. Patel, Edward A. Lee, and Hugo A. Andrade. An Automated Mapping of Timed Functional Specification to a Precision Timed Architecture. In *2008 12th IEEE/ACM Int. Symp. Distrib. Simul. Real-Time Appl.* IEEE, 2008.

[62] Bjorn Forsberg, Luca Benini, and Andrea Marongiu. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *2018 Des. Autom. Test Eur. Conf. Exhib.*, pages 539–544. IEEE, 2018.

[63] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proc. 41st Annu. Des. Autom. Conf.* ACM, 2004.

[64] Joël Goossens and Pascal Richard. Optimal scheduling of periodic gang tasks. *Leibniz Trans. Embed. Syst.*, 2016.

[65] K. Goossens, J. Dielissen, and A. Radulescu. A Ethereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Des. Test Comput.*, 2005.

[66] Jan Gray. GRVI-Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator. In *Proc. 24th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2016.

[67] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proc. seventh ACM Int. Conf. Embed. Softw. - EMSOFT '09*. ACM Press, 2009.

[68] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *J. Syst. Archit.*, 2011.

[69] Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. New Schedulability Test Conditions for Non-preemptive Scheduling on Multiprocessor Platforms. *2008 Real-Time Syst. Symp.*, 2008.

[70] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *2009 30th IEEE Real-Time Syst. Symp.* IEEE, 2009.

[71] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level set-associative instruction caches. *J. Syst. Archit.*, 2008.

[72] Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable Cache Coherence for Multi-Core Real-Time Systems. In *RTAS*, 2017.

[73] M. Y. Hsiao. A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of R and D*, 1970.

[74] Huang-Ming Huang, Terry Tidwell, Christopher Gill, Chenyang Lu, Xiuyu Gao, and Shirley Dyke. Cyber-physical systems for real-time hybrid structural testing: a case study. In *Proc. 1st ACM/IEEE Int. Conf. cyber-physical Syst.*, 2010.

[75] Yijie Huangfu and Wei Zhang. PEG-C: Performance Enhancement Guaranteed Cache for Hard Real-Time Systems. *IEEE Embed. Syst. Lett.*, 2014.

[76] Anantha Chandrakasan Jan M. Rabaey. *Digital Integrated Circuits*. Prentice-Hall, 2002.

[77] Morris A Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Supercomput. ACM/IEEE 1997 Conf.*, 1997.

[78] N. Kapre. Marathon: Statically-Scheduled Conflict-Free Routing on FPGA Overlay NoCs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[79] N. Kapre and J. Gray. Hoplite: Building austere overlay NoCs for FPGAs. In *Field Programmable Logic and Applications*, 2015.

[80] Hany Kashif and Hiren Patel. Buffer Space Allocation for Real-Time Priority-Aware Networks. In *proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[81] Shinpei Kato and Yutaka Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *2009 30th IEEE Real-Time Syst. Symp.* IEEE, 2009.

[82] Rich Katz Kenneth A. LaBel. NASA FPGA Needs and Activities. https://radhome.gsfc.nasa.gov/radhome/papers/label_fpga04.pdf, 2004.

[83] Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastaval. WCET-aware dynamic code management on scratchpads for Software-Managed Multicores. In *2014 IEEE 19th Real-Time Embed. Technol. Appl. Symp.* IEEE, 2014.

[84] Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. A Cache Design for Probabilistically Analysable Real-time Systems. In *Proc. Conf. Des. Autom. Test Eur.* EDA Consortium, 2013.

[85] Abderahman Kriouile and Wendelin Serwe. Formal Analysis of the ACE Specification for Cache Coherent Systems-on-Chip. In *Form. Methods Ind. Crit. Syst.* Springer, 2013.

[86] Matthew Kuo, Partha Roop, Sidharta Andalam, and Nitish Patel. Precision Timed Embedded Systems Using TickPAD Memory. In *2013 13th Int. Conf. Appl. Concurr. to Syst. Des.* IEEE, 2013.

[87] A Kurdila, M Nechyba, R Prazenica, W Dahmen, P Binev, R DeVore, and R Sharpley. Vision-based control of micro-air-vehicles: Progress and problems in estimation. In *Decis. Control. 2004. CDC. 43rd IEEE Conf.*, 2004.

[88] K Lakshmanan, S Kato, and R Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *2010 31st IEEE Real-Time Syst. Symp.* IEEE, 2010.

[89] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Real-Time Syst. Symp. (RTSS), 2010 IEEE 31st*, 2010.

[90] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, IEEE Computer Society, 2004.

[91] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[92] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of Global EDF for Parallel Tasks. In *2013 25th Euromicro Conf. Real-Time Syst.* IEEE, 2013.

[93] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *Proc. - Euromicro Conf. Real-Time Syst.*, 2014.

[94] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, 2005.

[95] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Syst.*, 2012.

[96] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D Patel, Stephen A Edwards, and Edward A Lee. Predictable programming on a precision timed architecture. In *Proc. 2008 Int. Conf. Compil. Archit. Synth. Embed. Syst.* ACM, 2008.

[97] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. Third IEEE Real-Time Technol. Appl. Symp.* IEEE Comput. Soc, 1997.

[98] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[99] Jane W S W Liu. *Real-Time Systems*. Prentice Hall PTR, 2000.

[100] Yu Liu and Wei Zhang. Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors. *J. Comput. Sci. Eng.*, 2015.

[101] Jing Lu, Ke Bai, and Aviral Shrivastava. SSDM: smart stack data management for software managed multicores (SMMs). In *Proc. 50th Annu. Des. Autom. Conf.*, 2013.

[102] Lars Lundberg. Multiprocessor scheduling of age constraint processes. In *RTCSA*, 1998.

[103] T R Maeurer and D Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 2005.

[104] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time Embed. Technol. Appl. Symp.* IEEE, 2013.

[105] Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *2014 IEEE 20th Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, pages 1–10. IEEE, aug 2014.

[106] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. WCET(m) Estimation in Multi-core Systems Using Single Core Equivalence. In *Real-Time Syst. (ECRTS), 2015 27th Euromicro Conf.*, 2015.

[107] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 2012.

[108] Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution. In *Proc. 9th Int. Work. Program. Model. Appl. Multicores Manycores - PMAM'18*. ACM Press, 2018.

[109] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Schedulability Analysis of Conditional Parallel Task Graphs in Multicore Systems. *IEEE Trans. Comput.*, 2016.

[110] S Metzlaff, I Guliashvili, S Uhrig, and T Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Proc. 24th Int. Conf. Archit. Comput. Syst. (ARCS)*, 2011.

[111] Jörg Mische, Christian Mellwig, Alexander Stegmeier, Martin Frieb, and Theo Ungerer. Minimally buffered deflection routing with in-order delivery in a torus. *Proc. Elev. IEEE/ACM Int. Symp. Networks-on-Chip - NOCS '17*, 2017.

[112] Jörg Mische and Theo Ungerer. Guaranteed Service Independent of the Task Placement in NoCs with Torus Topology. In *Proc. 22Nd Int. Conf. Real-Time Networks Syst.* ACM, 2014.

[113] David Mosberger. Memory consistency models. *ACM SIGOPS Oper. Syst. Rev.*, 1993.

[114] Thomas Moscibroda and Onur Mutlu. A Case for Bufferless Routing in On-chip Networks. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, 2009.

[115] Bryon Moyer. *Real World Multicore Embedded Systems*. Newnes, 2013.

[116] Tiago Rogerio Muck and Antonio Augusto Frohlich. Run-time scratch-pad memory management for embedded systems. In *IECON 2011 - 37th Annu. Conf. IEEE Ind. Electron. Soc.* IEEE, 2011.

[117] Frank Mueller. Compiler support for software-based cache partitioning. *ACM SIGPLAN Not.*, 1995.

[118] J Musmanno. Data Intensive Systems (DIS) Benchmark Performance Summary. Technical report, 2003.

[119] R. Naseer and J. Draper. Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs. In *Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European*, 2008.

[120] Geoffrey Nelissen, Vandy Berten, Joel Goossens, and Dragomir Milojevic. Techniques Optimizing the Number of Processors to Schedule Multi-threaded Tasks. In *2012 24th Euromicro Conf. Real-Time Syst.* IEEE, 2012.

[121] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Comput. Conf. (EDCC), 2012 Ninth Eur.*, 2012.

[122] Andreas Olofsson, Tomas Nordström, and Zain ul Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. *CoRR*, 2014.

[123] John K Ousterhout. Scheduling Techniques for Concurrebt Systems. In *ICDCS*, 1982.

[124] M. Panic, C. Hernandez, E. Quinones, J. Abella, and F. J. Cazorla. Modeling High-Performance Wormhole NoCs for Critical Real-Time Embedded Systems. In *proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[125] Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla. A hard real-time capable multi-core SMT processor. *ACM Trans. Embed. Comput. Syst.*, 2013.

[126] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. *ACM SIGARCH Comput. Archit. News*, 2009.

[127] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A Novel Technique to Use Scratch-pad Memory for Stack Management. In *Des. Autom. Test Eur. Conf. Exhib. 2007. DATE '07*, 2007.

[128] 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 *2011 17th IEEE Real-Time Embed. Technol. Appl. Symp.*, 2011.

[129] 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 *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2011.

[130] Rodolfo Pellizzoni and Marco Caccamo. Real-Time Management of Hardware and Software Tasks for FPGA-based Embedded Systems. *IEEE Trans. Comput.*, 56(12):1666–1680, dec 2007.

[131] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Des. Autom. Test Eur. Conf. Exhib. (DATE 2010)*. IEEE, 2010.

[132] Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *Real-Time Syst. (ECRTS), 2014 26th Euromicro Conf.*, 2014.

[133] Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. P-SOCRATES: A parallel software framework for time-critical many-core systems. *Microprocess. Microsyst.*, 2015.

[134] J A Poovey, T M Conte, M Levy, and S Gal-On. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro*, 2009.

[135] Aayush Prakash and HD D Patel. An instruction scratchpad memory allocation for the precision timed architecture. In *Des. Autom. Test Eur. ...*, 2012.

[136] I Puaut and C Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Des. Autom. Test Eur. Conf. Exhib. 2007. DATE '07*, 2007.

[137] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. A real-time capable coherent data cache for multicores. *Concurr. Comput. Pract. Exp.*, 2014.

[138] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core

systems. In *2014 Int. Conf. Embed. Comput. Syst. Archit. Model. Simul. (SAMOS XIV)*. IEEE, 2014.

[139] Eduardo Quiñones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *2009 21st Euromicro Conf. Real-Time Syst.* IEEE, 2009.

[140] Jan Reineke. Randomized caches considered harmful in hard real-time systems. *Leibniz Trans. Embed. Syst.*, 2014.

[141] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Syst.*, 2013.

[142] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Syst.*, 2013.

[143] Abhik Sarkar, Frank Mueller, Harini Ramaprasad, and Sibin Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *Proc. 2009 ACM SIGPLAN/SIGBED Conf. Lang. Compil. tools Embed. Syst. - LCTES '09*. ACM Press, 2009.

[144] Martin Schoeberl. One-way shared memory. In *2018 Des. Autom. Test Eur. Conf. Exhib.* IEEE, 2018.

[145] Paul Sebexen and Thomas Sohmers. Software Techniques for Scratchpad Memory Management. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 98–102. ACM, 2015.

[146] M. She. *Semiconductor Flash Memory Scaling*. University of California, Berkeley, 2003.

[147] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. Semi-Partitioned Hard-Real-Time Scheduling under Locked Cache Migration in Multicore Systems. In *2012 24th Euromicro Conf. Real-Time Syst.* IEEE, 2012.

[148] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[149] Zheng Shi and Alan Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *Second ACM/IEEE Int. Symp. Networks-on-Chip (nocs 2008)*. IEEE, 2008.

[150] Sudhir Singh. Performance optimization in gang scheduling in cloud computing. *Int. Organ. Sci. Res. Comput. Eng.*, 2012.

[151] C. Slayman. Whitepaper on Soft Errors in Modern Memory Technology. Technical report, 2010.

[152] Muhammad R Soliman and Rodolfo Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management with Compiler-Directed Prefetching.

[153] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synth. Lect. Comput. Archit.*, 2011.

[154] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.*, 53, 2004.

[155] V Suhendra and T Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proc. 45th Annu. Des. Autom. Conf.*, 2008.

[156] V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *26th IEEE Int. Real-Time Syst. Symp.* IEEE, 2005.

[157] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.

[158] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[159] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S.S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time Embed. Technol. Appl. Symp. RTAS 2016 - Proc.*, 2016.

[160] R Tabish, R Mancuso, S Wasly, S S Phatak, R Pellizzoni, and M Caccamo. (under review) A Real-Time Scratchpad-centric OS with Predictable Inter/Intra-Core Communication for Multi-core Embedded Systems. In *Real-Time Syst. RTS*. Springer, 2018.

[161] R. Tabish, R. Mancuso, S. Wasly, S.S. Phatak, R. Pellizzoni, and M. Caccamo. A reliable and predictable scratchpad-centric OS for multi-core embedded systems. In *Proc. IEEE Real-Time Embed. Technol. Appl. Symp. RTAS*, 2017.

[162] Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, 2010.

[163] Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *2010 Des. Autom. Test Eur. Conf. Exhib. (DATE 2010)*. IEEE, 2010.

[164] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *Work. Interact. between Oper. Syst. Comput. Archit.*, 2007.

[165] Corey Tessler and Nathan Fisher. BUNDLE: real-time multi-threaded scheduling to reduce cache contention. In *Real-Time Syst. Symp. (RTSS), 2016 IEEE*, 2016.

[166] Sascha Uhrig, Lillian Tadros, and Arthur Pyka. MESI-Based Cache Coherence for Hard Real-Time Multicore Systems. In *Archit. Comput. Syst. 2015*. Springer, 2015.

[167] Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 2010.

[168] R. Vargas, S. Royuela, M. Serrano, X. Martorell, and E. Quinones. A lightweight OpenMP4 run-time for embedded systems. In *Proc. ASP-DAC*, 2016.

[169] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *2013 25th Euromicro Conf. Real-Time Syst.* IEEE, 2013.

[170] Saud Wasly and Rodolfo Pellizzoni. A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems. In *2013 25th Euromicro Conf. Real-Time Syst.* IEEE, 2013.

[171] Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *2014 IEEE 19th Real-Time Embed. Technol. Appl. Symp.* IEEE, 2014.

[172] Saud Wasly and Rodolfo Pellizzoni. (under review)Bundled Scheduling of Parallel Real-time Tasks. In *Real-Time Syst. Symp. RTSS*. IEEE, 2018.

[173] Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. HopliteRT: An efficient FPGA NoC for real-time applications. In *F. Program. Technol. (ICFPT), 2017 Int. Conf.*, 2017.

[174] J Whitham and N Audsley. Implementing time-predictable load and store operations. In *Proc. EMSOFT*, 2009.

[175] J Whitham, RI I Davis, N Audsley, S Altmeyer, and C Maiza. Investigation of Scratchpad Memory for Preemptive Multitasking. *jwhitham.org*, 2012.

[176] Jack Whitham and Neil C. Audsley. Explicit Reservation of Local Memory in a Predictable, Preemptive Multitasking Real-Time System. In *2012 IEEE 18th Real Time Embed. Technol. Appl. Symp.* IEEE, 2012.

[177] Jun Yan and Wei Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *2008 IEEE Real-Time Embed. Technol. Appl. Symp.* IEEE, 2008.

[178] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Syst.*, 2012.

[179] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time Embed. Technol. Appl. Symp.* IEEE, 2014.

[180] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time Embed. Technol. Appl. Symp.* IEEE, 2013.

[181] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Comput. Archit. News*, 2010.