Open access • Proceedings Article • DOI:10.1109/RTAS.2016.7461321

A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems — Source link

Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad ...+3 more authors

Institutions: University of Illinois at Urbana–Champaign, University of Waterloo, Hitachi

Published on: 11 Apr 2016 - Real Time Technology and Applications Symposium

Topics: Memory hierarchy and Multi-core processor

A REAL-TIME SCRATCHPAD-CENTRIC OS FOR MULTI-CORE EMBEDDED SYSTEMS

BY

ROHAN TABISH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Marco Caccamo

Abstract

Multicore processors have been increasing in development by the industry to meet the ever-growing processing requirements of various applications because these processors offer benefits such as reduced power consumption, more processing power and efficient parallel task execution for general purpose work- loads. However, in hard real time systems where predictability is a key aspect, the average performance of these multicore processors is even worse than the scenarios in which the same task set is executed on a single core processor. This performance degradation is due to the fact that the multicore systems have shared resources such as DRAM, BUS and caches which make the system highly unpredictable. One way to achieve predictability in such systems is to serialize the access of the cores to the shared resources such that there is no contention. Another widely emerging approach is the integration of the scratchpad memory. Using scratchpad, at run time, the code and data for the requested task is made available in the scratchpad and contention can be avoided.

In this thesis, we approach the problem of shared resource arbitration at an OS-level and propose a novel scratchpad-centric OS design for multi-core platforms. In the proposed OS, the predictable usage of shared resources across multiple cores represents a central design-time goal. Hence, we show (i) how contention-free execution of real-time tasks can be achieved on scratchpad-based architectures, and (ii) how a separation of application logic and I/O operations in the time domain can be enforced. To validate the proposed design, we implemented the proposed OS using a commercial-off-the-shelf (COTS) platform. Experiments show that this novel design delivers predictable temporal behavior to hard real-time tasks, and it improves performance up to 2.1x compared to traditional approaches.

To my Parents and all those who are important to me.

Acknowledgments

I would like to express my deep gratitude to Professor Marco Caccamo, my research supervisor, for his patient guidance, enthusiastic encouragement, and useful critiques of this research work. My grateful thanks are also extended to Renato Mancuso, a PhD student in Department of Computer Science, at UIUC for his constant support in building our system. I would also like to thank Professor Lui Sha and other CS faculty members for their continuous support and encouragement. My special thanks are also extended to all my friends including Fardin Abdi, Or Dantsker, C.Y. Chen and others.

This thesis is dedicated to my parents for their continuous support and encouragement throughout my graduate studies, and to my supernatural mother especially. Without her support it would not be possible to be this successful in my life. I would also like to dedicate this thesis to Dr. Fatima Ayub for her continuous support during rough times. Finally, the work in this thesis is also dedicated to my brother Hasan Javed and to my sister Senia Shafaq.

Table of Contents

List of Figures . . . . **vi**

List of Tables . . . . **vii**

Chapter 1 Introduction **1**

Chapter 2 Related Work **3**

Chapter 3 System Model and Assumptions **6**
- 3.1 Scratchpad Memories 6
- 3.2 DMA Engines 7
- 3.3 Dedicated I/O Bus 7
- 3.4 Memory Organization 8
- 3.5 Task Model 8

Chapter 4 Proposed OS Design **10**
- 4.1 Overview 10
- 4.2 Scratchpad and CPU Co-scheduling 11
- 4.3 I/O Subsystem Design 13

Chapter 5 Implementation **15**
- 5.1 Architectural Overview of Considered Platform 15
- 5.2 Preliminary Development Efforts 16
- 5.3 Implementation of SPM-Centric OS Using Erika Enterprise 18
- 5.4 Integrating I/O to SPM-Centric OS 19
- 5.5 Support For OS Configurator to Manage Relocatable Tasks 22
- 5.6 Example Showing The Process of Defining Relocatable Tasks 22

Chapter 6 Evaluation **30**
- 6.1 SPM-Centric OS Overhead Evaluation 30
- 6.2 Results of Achievable I/O Bandwidth 30
- 6.3 Results of Synthetic Benchmarks 31
- 6.4 Results of EEMBC Benchmarks 32
- 6.5 Schedulability Analysis 33

Chapter 7 Conclusion **35**

References **36**

List of Figures

3.1 Multicore architecture satisfying our hardware assumptions 8

4.1 Scheduling CPU, DMA and local memory 12
4.2 Interaction between I/O Core and Core 1 for task scheduling. 13

5.1 I/O SPM partition showing task and device buffers 20

6.1 Experimental execution time for synthetic benchmarks. 32
6.2 Experimental execution time for EEMBC benchmarks. 33
6.3 Schedulability with SPM-based and traditional scheduling models. 34

List of Tables

3.1 Suitable Commercial Multicore COTS platforms 8
3.2 Task's Parameters 9

5.1 Characteristics of Freescale MPC5777M SoC 16

6.1 Details of OS Parameters 30
6.2 Details of EEMBC Benchmarks. 33

Chapter 1

Introduction

Multi-core platforms are mainstream products. Multi-core chips allow different processing tasks to execute in parallel while accessing a set of shared hardware resources, including: main memory, buses, caches, and I/O peripherals. Unfortunately, when one or more of these resources is utilized up to its saturation point, severe and unpredictable inter-core interference can heavily affect the system's temporal behavior. From a real-time point of view, unregulated contention on shared resources induces significant execution time variance. Hence, specific mechanisms to manage and schedule shared resources need to be designed and validated. This problem has also been acknowledged by the Federal Aviation Administration (FAA), which currently imposes the use of a single core for safety-critical avionic applications unless proper analysis and mitigation of inter-core interference channels are demonstrated [1].

The problem of shared resource contention in a multicore environment has been approached in literature from different perspectives: a) novel multi-core hardware platforms have been designed [5,23], b) new OS-level techniques have been developed to perform shared resource partitioning and management on commercial-off-the-shelf (COTS) platforms [16]. While a hardware solution might be desirable to meet the needs of modern real-time systems, however, it is not a cost-effective solution for embedded industry. Conversely, enforcing determinism at software level on a general-purpose COTS architecture may trade some performance with execution time predictability. In this work, we propose an approach that lies in between the methodologies mentioned above. In fact, (i) we consider a segment of COTS platforms that are designed to support desirable features for hard real-time computation and (ii) redesign parts of the operating system (OS), leveraging such features to guarantee predictability and preserve performance. With these objectives in mind, we focus on scratchpad-based multi-core platforms. Scratchpad memories, in fact, have been proven to provide better temporal isolation when compared to traditional caches [17,19]. Alongside, we exploit additional hardware features that vendors now include 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.

1. A novel operating system design is built ground-up to achieve temporal predictability. Our OS design targets multi-core embedded COTS platforms and exploits core specialization and low level resource management policies.

2. To the best of our knowledge, this is the first OS that integrates a scratchpad-based CPU scheduling mechanism with a task schedule-aware I/O subsystem.

3. A novel analysis is derived to calculate the response time of real-time tasks under the proposed scheduling strategy.

4. Finally, a full implementation of the proposed OS has been performed using a commercially available multi-core micro-controller. Its design has been validated using a combination of synthetic tasks and EMBC benchmarks.

The rest of the thesis is organized as follows. Chapter 2 briefly reviews the related work. Next, Chapter 3 introduces the considered system model and architectural assumptions. The design of the proposed OS is described in Chapter 4, We describe the performed implementation in Chapter 5, and discuss the experimental results in Chapter 6. Finally, the thesis concludes in Chapter 7.

Chapter 2

Related Work

Temporal predictability is a crucial design-time constraint for real-time operating systems (RTOS). Several RTOS designs have been proposed, and a number of implementations are available, such as: QNX Neutrino[1], FreeRTOS[2], Wind River VxWorks[3]. These RTOS were designed for single-core platforms, where the use of real-time scheduling policies, efficient inter-process communication and prioritized interrupt handling were enough to ensure temporal predictability. Support for multi-core platforms was later introduced without a substantial change in design. Unfortunately, however, a new set of challenges (mainly related to shared hardware resource management [5, 16, 23]) is faced when trying to achieve predictability on multi-core systems.

In avionic standards such as ARINC 653 and ARINC 651, the concept of resource partitioning is central for the design of safety-critical systems. Even if different partitions execute on the same physical processor, the behavior/misbehavior of a software component should not affect the execution of another component running on a separate partition [28]. In single-core systems, requirement for inter-partition isolation can be achieved by employing time division and fault containment strategies. On multi-core systems, however, how to enforce and certify strong partitioning across different cores is still an active research topic.

In [12], Jean *et al.* provide a high-level discussion of the main issues for the extension of existing avionic standards to multi-core systems. The work considers multi-core integrated modular avionic (IMA) systems were partitions may run in parallel on different cores. The authors raise the concern that in the presence of faults, the use of shared hardware resources may lead to a violation of strict inter-partition isolation requirements. In multi-core systems, interference channels (if not carefully mitigated) are also present under normal operating conditions. This has been acknowledged by certification authorities [1] and it represents a source of concern for the use of multi-core processors in avionics systems. In this work, we propose a RTOS design that leverages co-scheduling techniques of shared resources to mitigate inter-core performance interference. Although we envision that some of the proposed design principles could be reused to enhance temporal protection in multi-core avionics systems, the proposed OS design rather targets embedded platforms suitable for automotive systems and its extension to IMA is currently out of the

[1]http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html
[2]http://www.freertos.org/
[3]http://www.windriver.com/products/vxworks/

scope of this work.

The proposed layer of OS-level strategies to perform co-scheduling of shared resources is in line with the concept of Deterministic Platform Software (DPS) as defined in [11]. Specifically, in our system we enforce a deterministic execution model for running applications, constructing a DPS that actively controls and schedules access to shared resources. Following the nomenclature proposed in [11], since tasks need to be specifically engineered and compiled to comply with our task model, the proposed solution is *application aware*. In this work, the multi-stage task model is also consistent with the Acquisition Execution Restitution (AER) task model proposed in [8].

The AER model proposed in [8] achieves predictability by executing tasks from local core memories (scratchpads), while shared memory resources are only used for inter-core communication and device I/O during acquisition and/or restitution phases. Inter-core interference arising from unregulated access to shared memory is mitigated by ensuring that: (i) the execution phase of different tasks can progress in parallel on multiple cores; and (ii) at most one acquisition or restitution phase is in execution at any instant of time. In [8], the fundamental assumption is that the total footprint of all the tasks assigned to a core fits inside the core's local memory. In this work, we relax this constraint and only require that a task fits in half of the local memory space. This relaxation leads to important differences in the RTOS design: in fact, dynamic loading and unloading of tasks from/to local memories (together with I/O data) need to be handled. For these reasons, tasks' execution phases are parallelized; additionally, task loading/unloading is pipelined with execution by using DMA engines. Finally, asynchronous I/O device activity is deconflicted from applications by exploiting hardware specialization at the bus level and by handling system-to-device interaction inside an isolated I/O subsystem.

Techniques to derive WCET bounds on a multi-core system accounting for the major sources of unpredictability have been thoroughly analyzed in [6]. The latter work provides an in-depth overview of the state-of-the-art analysis methodologies for shared buses, shared caches as well as scratchpad memories. Its focus, however, is the derivation of safe WCET bounds in presence of typical platform features and given a known task set. However, there is no discussion on how a real-time OS can be designed on multi-core platforms to support multi-tasking subject to temporal constraints.

The design of multi-core architectures that are able to provide worst-case execution time guarantee have been proposed in [5, 23]. Specifically, the precision timed (PRET) architecture [5] introduces task runtime control and deadline enforcement at the instruction set (ISA) level. Additional hardware modifications allow to achieve better performance without sacrificing predictability. Similarly, in the MERASA project [23, 29], predictability is achieved at hardware level by controlling inter-core interference. These works propose architectural features that have been prototyped on field-programmable gate array (FPGA), but unfortunately such features cannot be found in COTS system-on-chips (SoC).

In [15,16,30,31] the authors presented the Single-Core Equivalence framework: that is a set of OS-level techniques that can be implemented on COTS platforms to enforce spatial and temporal partitioning of shared memory resources. Derived analysis and experimental validation showed that WCET of tasks can be bounded and that inter-core temporal isolation can be achieved. Three main differences exist with respect to the proposed approach. First, the work in [16] assumes a traditional task execution model, while in this work this assumption is relaxed using a three-phase execution model. Second, in this thesis we focus on scratchpad-based architectures. Finally, this work also proposes the design of a novel I/O subsystem.

The proposed work is a contribution to existing literature on the usage of scratchpad memories (SPM) for real-time systems [17, 19, 22, 24, 26, 27]. In fact, a number of works have explored the benefits of scratchpad memories over traditional caches for multi-core platforms [17, 19]. Other works on embedded systems exist that propose scratchpad memory allocation strategies targeting real-time applications [3, 7, 9, 14, 21]. In [22] 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, (ii) temporal, and (iii) hybrid approaches.

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 [22] is not predictability but energy efficiency, making its contribution substantially different from the proposed work.

Finally, our design shares some similarities with scratchpad scheduling approaches that have been proposed in [24–27]. Compared to these works, our approach mainly differs in three aspects: (i) it is not focused exclusively on scratchpad management, but we rather show how a scratchpad can be integrated within an overall OS design; (ii) a full OS design is implemented on a commercially available (COTS) micro-controller; and (iii) it is also discussed how I/O traffic issued by different cores is deconflicted.

Chapter 3

System Model and Assumptions

This chapter summarizes the task model that we use and the hardware assumptions we rely on for the design of the proposed predictable operating system, namely SPM-centric OS.

3.1 Scratchpad Memories

The first assumption we make is the presence of scratchpad memory (SPM). We assume that each core in our system features a block of private scratchpad memory. Moreover, in this work we assume that the size of each per-core scratchpad memory is big enough to fully contain the footprint of any two tasks in the system. Hence, the footprint of the largest task in the system is at most half the size of the scratchpad memory. Although this assumption may appear restrictive, we make the following considerations. First, modern scratchpad-based micro-controllers provide scratchpad memories that have a size in the same order of magnitude as the main memory. For instance, in the MPC5777M that we use for our evaluation, each core includes 80 KB of scratchpad with a total main memory size of about 400 KB. Second, hard real-time control tasks typically are compact in terms of memory size. Third, if a task violates this size constraint, known methodologies exist [2, 13] to split a large application into smaller sub-tasks that are individually compliant with the imposed constraint.

As we will discuss in chapter 4, before tasks can be executed from SPM, their code and data need to be transferred from main memory. Thus, we adopt a task model that is composed of three phases: a *load phase*, an *execution phase* and an *unload phase*. First, during the load phase, the code and data image for the activated task is copied from main memory to the SPM. Next, during the execution phase, the loaded task executes on the CPU by relying on in-scratchpad data. Finally, the portion of data that has been modified and needs to remain persistent across subsequent activations of the task is written back to main memory during the unload phase.

3.2 DMA Engines

To avoid to stall the CPU when load/unload operations are performed, we assume that copy operations toward/from the scratchpad memories can proceed in parallel with task executions. This can be achieved as long as execution and load/unload phases belong to two distinct tasks. In order to parallelize load/unload operations with task execution, we rely on direct memory access (DMA) engines. We assume that the hardware provides DMA engines that are able to transfer data from the main memory into the scratchpad and vice versa. By exploiting (i) the capability of parallelizing load/unload operations together with task execution, and (ii) the assumption that any task image can fit in half of the scratchpad memory, it is possible to hide task loading/unloading overhead during task execution, as we discuss in chapter 4.

3.3 Dedicated I/O Bus

The next made assumption is about the organization of the I/O subsystem. Since the activity of I/O devices is typically triggered by external events, it is inherently asynchronous. Unfortunately, unregulated I/O activity on the system bus can lead to unpredictable contention with CPU activity [4]. Hence, unarbitrated I/O traffic represents one of the major sources of unpredictability in real-time systems. To deconflict the inherently asynchronous activity of I/O devices from application cores' activity, we assume that a dedicated bus exists 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 [4, 18]. However, the increased awareness of chip manufacturers about this problem has resulted in the design of COTS platforms that use dedicated buses to handle I/O transactions. Table 3.1 shows a non-exhaustive list of COTS platforms with this feature. In this work, we assume that suitable hardware exists to enforce a separation between I/O and CPU-originated memory transactions. Furthermore, traffic transmitted over the dedicated I/O bus needs to be handled, pre-processed and scheduled before reaching the application cores. Thus, we assume that an I/O processor exists, which we hereafter refer to as *I/O core*. 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. As previously discussed, this design choice allows us to perform co-scheduling of CPU and I/O activities to achieve higher system predictability. A summary of the architectural assumptions discussed so far is provided in Figure 3.1.

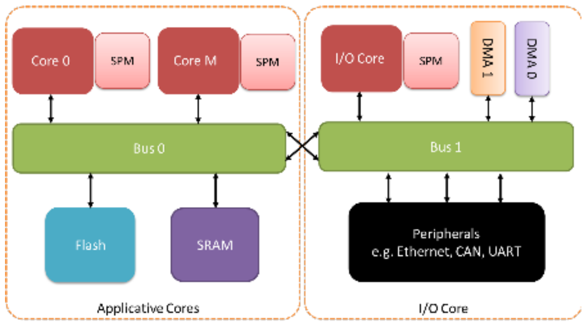

Figure 3.1: Multicore architecture satisfying our hardware assumptions

3.4 Memory Organization

As micro-controllers evolve into complex multi-core systems, more advanced support of memory protection schemes is provided. However, for the purpose of this work, no specific assumption needs to be made about platform memory protection features. Hence, the presence of a memory management unit (MMU) is not a necessary requirement. We discuss in Chapter 5 how task relocation from main memory to scratchpads can be achieved without MMU support. Intuitively, MMU support allows for a straightforward implementation of task relocation by relying on page table manipulation. Usually, systems without MMU include a memory protection unit (MPU). MPUs support the definition of per-core access permissions based on linear ranges of physical memory addresses. Although they are not necessary to implement our system, MPUs can be easily supported within our design.

The hardware assumptions described so far represent desirable features that are becoming increasingly common in modern COTS micro-controllers used for safety-critical applications. Table 3.1 provides a list of some of the available COTS platforms that satisfy the described assumptions.

Table 3.1: Suitable Commercial Multicore COTS platforms

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

3.5 Task Model

For the proposed design, we consider a partitioned and fixed priority scheduling policy; additionally, each core has 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. The deadline of each task is assumed to be less than or equal to its minimum inter arrival time. Table 3.2 summarizes the notation used for task parameters. As discussed in chapter 4, tasks follow a three-phases model. Hence, to satisfy temporal constraints, the last phase (unload) of a task needs to complete before the deadline.

For ease of implementation, this work assumes non-preemptive tasks, although we plan to relax this assumption as part of our future work.

Table 3.2: Task's Parameters

Term	**Definition**
τ_i	a task in the system
$\tau_i.T$	task's MIT or period(if task is periodic)
$\tau_i.c$	task's execution time including all overheads
σ	TDMA slot size for the DMA operation

Chapter 4

Proposed OS Design

In this chapter, we describe the design of the proposed SPM-centric OS by relying on the previously discussed assumptions.

4.1 Overview

The central idea of the proposed SPM-centric OS 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 3.1. 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. The benefit of this design is twofold. First, it allows high-level scheduling of accesses to main memory, ultimately achieving conflict-free execution of tasks from local memories. Second, performance benefits derived from the usage of fast scratchpad memories are exploited, ultimately combining better performance with higher temporal determinism.

We refer to the capability of our SPM-centric OS to dynamically move applicative tasks in and out of the SPM memories as *support for relocatable tasks*. As mentioned in Chapter 3, if hardware MMU support exists, task relocation can be achieved using page table manipulation. Otherwise, advanced compiler level techniques can be exploited to generate position independent code, as described in Chapter 5.

In the proposed SPM-centric OS, 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*. Typically, a single DMA engine is capable of utilizing the full main memory bandwidth in micro-controller platforms. Nonetheless, the design constraint that imposes the use of a single applicative DMA can be relaxed if the main memory subsystem allows two or more DMA engines to transfer data concurrently without saturating the main memory bandwidth.

4.2 Scratchpad and CPU Co-scheduling

Load/unload operations for tasks running on the M applicative cores need to be serialized to prevent unregulated contention over the memory bus. Hence, only a single DMA is required as application DMA for all the M applicative cores. Several schemes are known to fairly share a single resource across different consumers. For the scope of our design, we employ a time division multiple access (TDMA) scheme to serialize task load/unload operations among M applicative cores. The main advantage of the TDMA scheme lies in its simplicity of implementation. Although in this work we restrict our discussion to TDMA sharing of the applicative DMA, the proposed OS can be extended to consider round-robin policies as well as budget-based schemes.

In order to perform TDMA-based scheduling of the application 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 4.1 depicts the sequence of operations in our TDMA scheme for a system with $M = 2$ application CPUs. Note that the TDMA enforcement needs to be centralized. Hence, in our design, the I/O core is responsible for interfacing with application cores' schedulers through active/ready queues, programming the application DMA as well as enforcing the time-triggered TDMA slots. In particular, Figure 4.1 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 the I/O core can neither load nor unload any applicative core scratchpad. Effectively, the slot is âĂIJwastedâĂİ.

Since tasks need to be loaded/unloaded in parallel with respect to CPU activity, two partitions are created on the scratchpad. There is logically no difference between the two scratchpad partitions. Thereby, tasks may execute from either one of the two, depending on their arrival time. Interchangeably, one of them contains the image of the task which is currently being executed, while the second half is used to load (unload) the image of the next (previous) task to be executed (that was completed). Note that when a task is executing on the CPU while a second task is loaded/unloaded in background, CPU and DMA contend for scratchpad access. However, the impact of this contention on the timing of the tasks is typically negligible for two main reasons. First, scratchpads are often implemented as dual-ported memories; thus, they are able to support stall-free CPU and DMA operations. In fact, on the considered MPC5777M platform we have verified this by experimentation and found that both the core and the DMA module do not suffer any delay when they access the SPM simultaneously. Second, in a system with M CPUs, DMA-CPU contention over scratchpad involves only two masters, as opposed to the traditional approach where up to M masters could contend for main memory.

As depicted in Figure 4.1, the application 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.

In the proposed design, the next task to be executed is loaded in background while the foreground running task is not interrupted until its completion. The described mechanism allows to hide the DMA loading overhead, avoiding contention in main memory and exploiting performance benefits deriving from SPM usage.

The work-flow followed by an applicative core and the I/O core at the boundary of each TDMA slot is depicted in

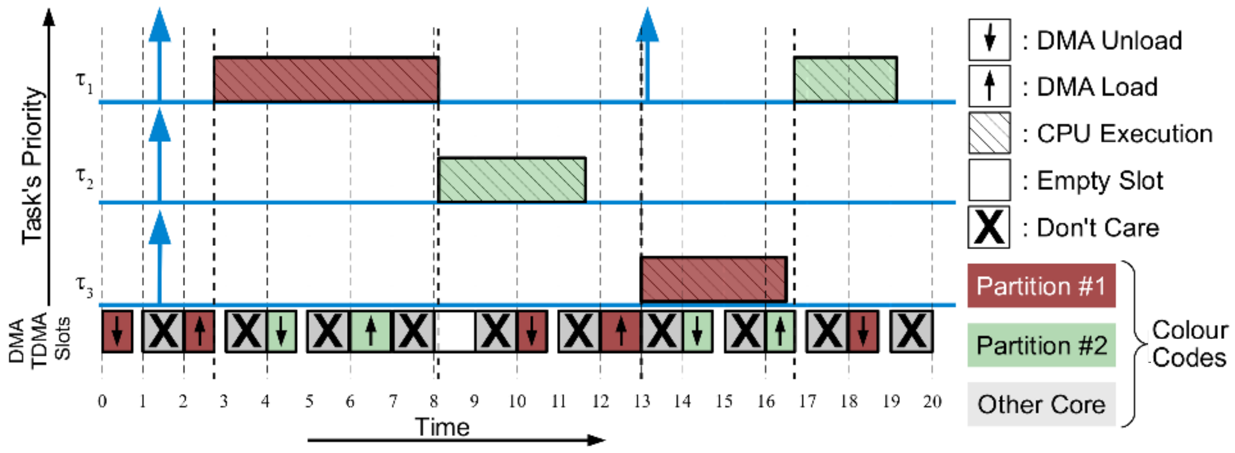

Figure 4.1: Scheduling CPU, DMA and local memory

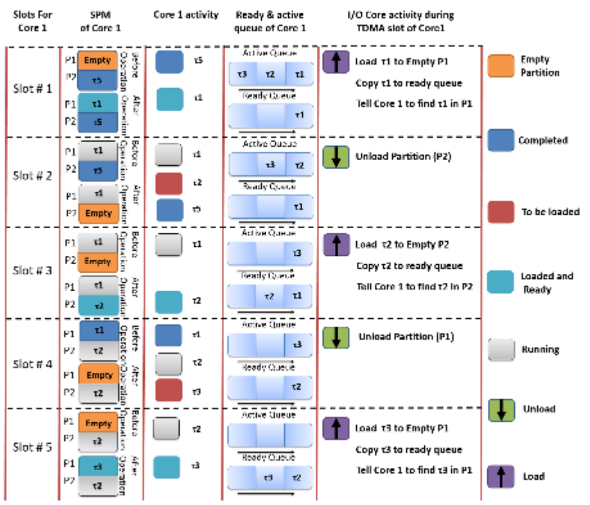
Figure 4.2: Interaction between I/O Core and Core 1 for task scheduling.

Figure 4.2. 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.

4.3 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 mentioned in Chapter 3, we assume that 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 (output) device buffer represents the position in memory where data produced by devices (tasks) is accumulated before being dispatched to tasks (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.

Chapter 5

Implementation

In this chapter, we provide the details of SPM-centric OS implemented using a COTS platform that supports the hardware assumptions described in chapter 4.

5.1 Architectural Overview of Considered Platform

The MPC5777M is a system-on-chip (SoC) produced by Freescale and represents the highest performing chip in the MPC production line as of Q1 2016. A brief summary of the architectural features of the MPC5777M MCU is provided in Table 5.1. The chip includes four processors: two independent E200Z710 cores operating at 300 MHz, a single 300 MHz delayed lockstep core and a single E200Z425 I/O core operating at 200 MHz. The two E200Z710 cores are part of the computational shell and are considered applicative cores, whereas the single E200Z425 is part of the peripheral control shell and embeds an ISA extension for DSP operations as well as a floating point unit (FPU). The I/O core also represents the master core that is active at boot time and is responsible for performing the bootstrap sequence for the rest of the system (peripherals and applicative cores). The core running in delayed lockstep mode cannot be configured to run in decoupled parallel mode.

Each core features a private instruction cache (16 KB, 8KB on the master core). Also, a 4 KB data cache is available on the applicative cores. Apart from local instruction and data caches, each core features independent fast local memories (scratchpads) for instructions and data: I-RAM (16KB) and D-RAM (64KB). No memory translation unit (MMU) is available on this platform; hence, no virtual memory can be defined. However, by using the MPU (Memory Protection Unit), masters can be allowed/denied to access linear regions of physical memory. All the cores present on the computational and peripheral control shell share a SRAM array of 404KB. Application cores can access the SRAM directly through a high bandwidth XBAR switch that operates at a frequency of 200MHz. A separate and slower XBAR is dedicated for transferring peripheral data to and from the I/O core. The slower XBAR operates at 100MHz.

Table 5.1: 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)
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)
SRAM Size	404 KB
Flash Size	8 MB
Main Peripherals	Ethernet, FlexRay, CAN, I2C, SIUL

5.2 Preliminary Development Efforts

The proposed SPM-centric OS 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 that provides an essential infrastructure for vehicular software. Erika Enterprise OS features a small memory footprint, can run on multi-core platforms, and supports the definition of periodic, preemptive tasks with both RM and EDF scheduling.

The default Erika OS implementation does not support the considered MPC5777M microcontroller. Therefore, the first part of our implementation effort focused on setting up the required development environment. This included: (A) installation and test-compilation using the WindRiver DIAB compiler; (B) instantiation of an experimental version of RT-DRUID used internally at Evidence Erika Enterprise; (C) instantiation of an experimental Erika OS porting for the MPC5777C (unfortunately, the direct porting to MPC5777M was not developed by Evidence). Finally, we performed the duplication of architecture-specific resources to port Erika OS to the new MCUs.

Next, our implemementation efforts focused on porting the main functionalities of the OS that are architecture-dependent. Specifically, we focused on: (A) adapting the boot sequence used for the MPC5777C to the differences in the cores included in the MPC5777M platform; (B) rewriting the architecture-specific code to bootstrap the two applicative cores (slaves) of the MCU from the master core (I/O core); (C) performing proper clock initialization of the I/O subsystem and peripherals, with particular focus on the GPIO pads which can provide a visual feedback of the operational state of the board and be used as a basic input/output mechanism; (D) configuring and assessing

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

the operational state of the System Timer Module (STM) that can be used to generate periodic interrupts and thus the activation of tasks; (E) adding support for the Periodic Interrupt Timer (PIT) in order to support general timing functionality within the task; and (F) rewriting the architecture-specific code to initialize and configure the Interrupt Controller (INTC), which can be used to dispatch tasks on CPUs, as well as to perform inter-core communication.

The next step of our implementation focused on including functionalities that are required for the evaluation of time-sensitive code and high-level debugging. Specifically, we performed: (A) the development of a basic driver for the UART communication interface. A Print statement can be inserted in any section of the OS and in user-defined tasks. The corresponding output then can be captured through a serial interface; (B) the development of the L1 cache initialization code (for all the three cores) and assessment of timing behavior, in terms of: (1) time to execute cached instructions and (2) the time to reference cached data; and (C) the timing behavior of local memories (scratchpads) with respect to the baseline (timing for SRAM access). The results obtained in steps B and C are reported in the next Chapter.

To deepen our understanding of the main memory bus (fast XBAR), we developed benchmarking code to synthetically enforce a series of corner-case memory access patterns on applicative code and implemented performance monitoring mechanism leveraging on the support introduced in the new Freescale E200Z425 and E200Z710 cores. Using this mechanism made it possible to correlate runtime data from the benchmarks with measurement on their resource usage. Specifically, each benchmark was able to produce data for the following metrics: the runtime in CPU cycles; the number of executed instructions; the number of generated SRAM memory fetches for instructions (used to estimate instruction cache misses); and the number of generated SRAM memory fetches for data (used to estimate data cache misses). The results of these experiments are reported in the Evaluation chapter.

As a part of our implementation efforts, we also added the support of the high speed I/O device on the I/O core. The main goal is to transition towards an I/O subsystem that aggregates peripheral events asynchronously one one side, while synchronously transmitting peripheral data to and from applicative cores on the other side. Various high bandwidth I/O peripherals exist within MPC5777M. These include the CAN adapter, the FlexRay adapter and the Fast Ethernet Controller (FEC). For our implementation, we selected FEC as a suitable candidate since it is capable of data transfers of upto 100Mbps. The Erika OS does not provide native support for the FEC. Therefore, as a preliminary part of the development effort we added the support for FEC into the Erika OS. The functionality of the FEC module was verified by transferring the data between a commodity laptop and the development board.

5.3 Implementation of SPM-Centric OS Using Erika Enterprise

From the results, provided in the Evaluation chapter it emerges that 3.5x speedup can be introduced on applicative tasks if their code is executed from local memories (scratchpads âĂŞ see Figure 6.1). Since this is an extremely constrained resource (16 KB for instructions, 64 KB for data), we propose a solution in which: (A) task images are permanently stored in SRAM (or flash); (B) a dedicated DMA channel is used to move task images to and from the local memories upon task activation; and (C) a secondary DMA channel is used to perform task-related transfers of I/O data from the peripheral shell. This solution would allow the deployment of complex applicative code by exploiting the performance benefits of local memories, while at the same time hiding their constrained size using advanced scheduling techniques at OS-level as explained in the proposed OS design chapter.

In order to move the tasks images to and from SRAM to the local memories, we had to define position independent (relocatable) tasks. The Erika OS by default does not support relocatable tasks. In order to generate position independent tasks, we rely on 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. For basic implementation, we verified this by moving the tasks in and out of the SPM from and to the SRAM using CPU. Once proper functionality of relocatable tasks was verified and tested, the task movement from in and out of the SPM was extended using eDMA module.

Each relocatable task in our OS is assigned a set of specific metadata. These metadata allow the system software to locate the data and code image of a relocatable task in SRAM. Moreover, they contain information about the status of the task and its position in SPM (if already relocated). The code listing 5.1 shows the structure with different feilds containing this information. To move relocatable tasks at runtime and in parallel with respect to applicative CPU activity from the main memory to scratchpad memory, we use the eDMA module.

Since there is only one eDMA module that can be used to move data on the behalf of two applicative cores, a scheduling strategy is required. As discussed in Chapter 4, we have implemented TDMA based scheduling of the I/O. Nevertheless, we believe that more efficient strategies can be investigated as a part of the future work. As described in Figure 4.2 during a particular TDMA slot of a specific core either a load or an unload operation is performed. The load operation includes movement of the code and data of a task from the SRAM to the SPM memory, whereas, the unload operation only requires copy back of the task's data from the SPM to the SRAM.

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

Listing 5.1: Metadata containing relocatable task definition

```
struct reloc_entry {
  /* Base Register for program-counter-relative jumps and indirect data */
  EE_INT32 r13val;
  /* Function pointer to the body of task */
  void (* funct) (void);
  /* Unique id of the task in the system */
  unsigned int task_id;
  /* Flags for different options. One options is to transfer the task image
  from/to SPM using eDMA or the CPU */
  EE_TYPELOADFLAGS flags;
  /* Pointer to code location in the SPM where the code is loaded */
  void * code_loaded_at;
  /* Pointer to data location in the SPM where the task is loaded */
  void * data_loaded_at;
  /* Pointer to code start and end location of a task in the SRAM */
  void * code_start;
  void * code_end;
  /* Pointer to data start and end location of a task in the SRAM */
  void * data_start;
  void * data_end;
};
```

5.4 Integrating I/O to SPM-Centric OS

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 application tasks a synchronous view of I/O data. This is achieved by distinguishing between data production and their dispatch to and from tasks. In fact, we allow I/O data to flow from and to I/O subsystem to tasks only at the boundary of load/unload operations.

On MPC5777M, a dedicated bus connects the SPM of I/O core to peripherals. Hence, asynchronous peripheral traffic can reach the I/O subsystem without interfering with task execution. For each device used in the proposed

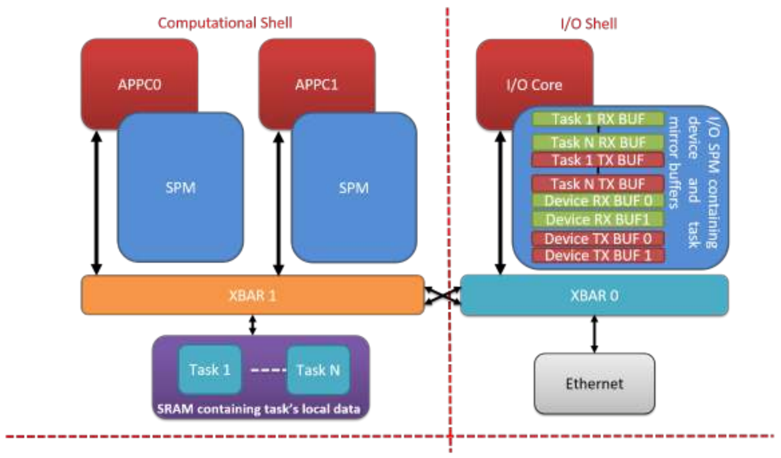

Figure 5.1: I/O SPM partition showing task and device buffers

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 (output) device buffer represents the position in memory where data produced by devices (tasks) are accumulated before being dispatched to tasks (devices). Figure 5.1 shows the partitioning of the I/O scratchpad. In particular, we have considered FEC device for our implementation. However, the procedure for any I/O device will be same.

In our design, peripheral drivers can operate with interrupt-driven or polling mechanisms. For DMA- capable peripherals supporting interrupt-driven interaction, the driver only needs to specify the address in the SPM of the device buffer from orto 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 the interaction with the considered device is. Nonetheless, all the device-related events are served with priority levels that are lower than task-scheduling events, such as the T DMA slot timer event and the completion event 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. In order for a relocatable task to subscribe to a I/O data, we have updated the metadata shown in Listing 5.1 to that in Listing 5.2 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 and 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 new 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.

Listing 5.2: Metadata containing relocatable task definition with I/O

```
struct reloc_entry {
  /* Base Register for program-counter-relative jumps and indirect data */
  EE_INT32 r13val;
  /* Function pointer to the body of task */
  void (* funct) (void);
  /* Unique id of the task in the system */
  unsigned int task_id;
  /* Flags for different options. One options is to transfer the task image
  from/to SPM using eDMA or the CPU */
  EE_TYPELOADFLAGS flags;
  /* Pointer to code location in the SPM where the code is loaded */
  void * code_loaded_at;
  /* Pointer to data location in the SPM where the task is loaded */
  void * data_loaded_at;
  /* Pointer to code start and end location of a task in the SRAM */
  void * code_start;
  void * code_end;
  /* Pointer to data start and end location of a task in the SRAM */
  void * data_start;
  void * data_end;
  /* Contains the Input device buffer of the task in I/O SPM*/
  unsigned int inpData;
  /* Contains the Output device buffer of the task in I/O SPM */
   unsigned int outData;
};
```

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 within 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 applicative core. The reverse path is followed by task output data during the task unload phase.

5.5 Support For OS Configurator to Manage Relocatable Tasks

The traditional RTOSs lack support for relocatable tasks, same is the case with Erika OS. In the current implmentation of Erika OS, the process of manually defining the relocatable tasks is relatively complex and extremely error-prone. In fact, it is requires manual modifications in nine different files of the Erika source. In order to simplify this process, we have implemented a basic OS configurator. The OS configurator takes in the input definition of as set of tasks (one per file). Next, it generates Erika OS source files that contain the necessary logic and metadata to configure the user-defined tasks as relocatable.

Specifically, in Erika OS, each core specifies two configuration files, named `eecfg.h` and `eecfg.c`, to describe system related structures: the number of tasks, their priority, task entry points, the initial status and task chaining. When adding relocatable tasks (and normal tasks) these files need to be configured accordingly. The body of all the relocatable tasks configured in the system is defined in a custom file named `reloc_tasks.c`. The implemented tool is responsible for merging the body of all the user-defined tasks inside the mentioned file.

For linking purposes, the data addressed by the relocatable tasks are defined in a different file namely `reloc_tasks.h` and surrounded with appropriate compiler-specific PRAGMAs. This is fundamental to ensure that: (A) task-specific data are placed in the correct linker section, and (B) that the code referring to these data uses position-independent addressing. The latter file also contains the definition of the relocatable task table which holds the status of each relocatable task as shown in Listing 5.2.

Finally, appropriate sections to place data and code for tasks need to be added in the linker script for the I/O core only. For a 3-core system like the one considered in our implementation, a total of nine files need to be updated upon task configuration change. The developed tool is able to perform this operation automatically upon modification of any of the user-defined tasks.

5.6 Example Showing The Process of Defining Relocatable Tasks

Each core in the Erika OS runs its own RM scheduler and a separate copy of the OS. As mentioned before, the Erika OS does not have the support of the relocatable tasks, so adding the support for the relocatable tasks is one of the main contributions of this work. Basically, each core has its own main file that defines the main for each core. The name of these files for our current implementation are defined as `master.c`, `app_core1.c` and `app_core2.c`. The file master.c belongs to the I/O core, whereas `app_core1.c` and `app_core2.c` belong to the applicative core one and applicative core two respectively. The main file for each core also includes two configuration files named eecfg.h and eecfg.c. These files contain system related structures such as: the number of tasks, their priority, task entry points, the initial status and task chaining. As mentioned before, the relocatable tasks are defined in a custom file named `reloc_tasks.c`. However, the `reloc_tasks.c` and `reloc_tasks.h` files are populated with the definition of the task, which is defined separately in a file named taskX.c, where X is the number of task.

As an example we will show the different steps associated with integrating the taskX.c file for two tasks on one of the applicative core. The process is same for both applicative core. At first the tasks, along with their body and other parameters such as priority, taskID, coreID, stack and global variables is defined in the separate taskX.c file. For a two-task system, we define two seperate files named task1.c and task2.c. These two files are shown in Listings 5.3 and 5.4. From the Listings 5.3 and 5.4, we can see that each parameter of the task is encoded with /* and ends with */. This is done to help the OS configurator find out the begining and end of various parameters.

The files shown in Listings 5.3 and 5.4 are read by the OS configurator as part of the make process. The OS configurator reads each unique taskX.c file and merges it to generate `reloc_tasks.c` and `reloc_tasks.h` files. For this example the OS configurator generated `reloc_tasks.c` and `reloc_tasks.h` files that contain definitions of all the relocatable tasks as shown in Listings 5.5 and 5.6. The file `reloc_tasks.h` also shows the updated structure `reloc_entry` defined as `reloc_table`[RELOCTASKS]. In our example RELOCTASKS is defined as 2, since we only have two relocatable tasks. This `reloc_entry` structure shown in `reloc_tasks.h` is same as the one shown in Listing 5.2. We can clearly see how the structure `reloc_entry` is filled based upon the various parameters of the task. The `reloc_table` is used by the TDMA schduler running on the I/O core to program the eDMA module to move the tasks in and out of the SPM.

Listing 5.3: task1.c

```
/* @name: _test_one */
/* @core: 1 */
/* @prio: 1 */
/* @stack: 128 */
/* Body */
{
        int i = 0, count = 50UL;
        actcount1++;
        global_var = 1111;
        for (i=0; i<count; ++i) {
                ++relvar;
        }
        arloc[0] = 1;
        urelvar = 1;
        tt1();
}
/* Variables */
int relvar = 0;
int arloc[2] = {0, 0};
int urelvar;
int actcount1 = 0;
/* End */
```

Listing 5.4: task2.c

```
/* @name: _test_two */
/* @core: 1 */
/* @prio: 2 */
/* @stack: 128 */
/* Body */
{
        int i = 0, count = 50;
        actcount2++;
        global_var = 2222;
        for (i=0; i<count; ++i) {
                ++t2_a;
        }

        t2_b = 2;
        t2_c = 1;
        tt1();
}
/* Variables */
int t2_a = 0;
int t2_b;
int t2_c;
int actcount2 = 0;
/* End */
```

Listing 5.5: reloc_tasks.h

```
#ifndef __RELOC_TASKS_HEADER__
#define __RELOC_TASKS_HEADER__
/* Definitions of functions for the dynamic loader */
int load_task(int taskid);
void unload_task(int taskid);
int start_task(int taskid);
int load_start_task();
/* Variables for Task: _test_one */
extern void * reloc_test_one_cstart;
extern void * reloc_test_one_cend;
extern void * reloc_test_one_istart;
extern void * reloc_test_one_uiend;
/* Variables for Task: _test_two */
extern void * reloc_test_two_cstart;
extern void * reloc_test_two_cend;
extern void * reloc_test_two_istart;
extern void * reloc_test_two_uiend;
#if EE_CURRENTCPU == 0
#pragma option -Xsmall-data=0
#pragma option -Xsmall-const=0
/* Definition for Relocatable task: _test_one */
#pragma section DATA "reloc_test_one_init" "reloc_test_one_uinit" far-data
int relvar = 0;
int arloc[2] = {0, 0};
int urelvar;
int actcount1 = 0;
#pragma section CODE "reloc_test_one_code"  far-code
DeclareTask(_test_one);
/* Definition for Relocatable task: _test_two */
#pragma section DATA "reloc_test_two_init" "reloc_test_two_uinit" far-data
```

Listing 5.5 reloc_tasks.h (cont'd)

```
int t2_a = 0;
int t2_b;
int t2_c;
int actcount2 = 0;
#pragma section CODE "reloc_test_two_code"  far-code
DeclareTask(_test_two);
#define RELOCTASKS 2
struct reloc_entry reloc_table [RELOCTASKS] = {
        {
                0,                          /* R13 reg val at load-filled by loader */
                Func_test_one,                /* Task entry point  */
                1UL,                    /* task ID  */
                LOAD_ASYNC | LOAD_DMA,  /* load flags */
                0,                          /* code load location - filled by loader */
                0,                          /* data load location - filled by loader */
                &reloc_test_one_cstart,     /* beginning of task code */
                &reloc_test_one_cend,       /* end of task code */
                &reloc_test_one_istart,     /* beginning of task data */
                &reloc_test_one_uiend,      /* end of task data */
                TASK0_RXDATA,     /* begginning of task I/O data */
                TASK0_TXDATA,     /* begginning of task I/O data */
        },
        {
                0,                          /* R13 reg val at load-filled by loader */
                Func_test_two,                /* Task entry point  */
                2UL,                    /* task ID  */
                LOAD_ASYNC | LOAD_DMA,  /* load flags */
                0,                          /* code load location - filled by loader */
                0,                          /* data load location - filled by loader */
                &reloc_test_two_cstart,     /* beginning of task code */
                &reloc_test_two_cend,       /* end of task code */
```

Listing 5.5 reloc_tasks.h (cont'd)

```
        &reloc_test_two_istart,     /* beginning of task data */
        &reloc_test_two_uiend,      /* end of task data */
        TASK1_RXDATA,     /* beginning of task I/O data */
        TASK1_TXDATA,     /* beginning of task I/O data */
},
```

Listing 5.6: reloc_tasks.c

```
#include "reloc_tasks.h"
#if EE_CURRENTCPU == 0
extern int global_var;
unsigned char EE_SHARED_IDATA (* tt0) (void) = TerminateTask;
unsigned char EE_SHARED_UDATA (* tt1) (void);
unsigned char EE_SHARED_UDATA (* tt2) (void);
/* START of relocatable task definition */
DeclareTask(_test_one) {
        int i = 0, count = 50UL;
        actcount1++;
        global_var = 1111;
        for (i=0; i<count; ++i) {
                ++relvar;
        }
        arloc[0] = 1;
        urelvar = 1;
        tt1();
}
DeclareTask(_test_two) {
        int i = 0, count = 50;
        actcount2++;
        global_var = 2222;
        for (i=0; i<count; ++i) {
                ++t2_a;
        }
        t2_b = 2;
        t2_c = 1;
        tt1();
}
```

Chapter 6

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. Finally, we investigate the schedulability results of the proposed strategy.

6.1 SPM-Centric OS Overhead Evaluation

A crucial parameter of the proposed system is the size of the TDMA slot. This should be long enough to allow the completion of a load (or unload) operation for the task with the largest footprint in the system. However, in order to derive an upper-bound, we assume that a task footprint is constrained by the size of an SPM partition. Thereby, we measured the time to copy from/to half SPM (one partition) of an applicative core and derive the TDMA slot size accordingly. The results are reported in Table 6.1.

The application DMA needs to be programmed by the I/O Core to perform task relocation. Hence, DMA programming time represents an overhead introduced by our design. The time required to program the DMA has been measured and is reported in Table 6.1. Similarly, Table 6.1 reports the measured context-switch overhead of the implemented scheduler.

Table 6.1: Details of OS Parameters

Parameter	Time (μs)
Partition load time	432
Partition unload time	432
DMA setup	3.16
Context switch	0.46

6.2 Results of Achievable I/O Bandwidth

The performance of the proposed I/O subsystem (see Section 4.3) depends on the frequency of load/unload operations. In order to measure the achievable I/O bandwidth of the 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.

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[1]. 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.

6.3 Results of 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 6.1 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.

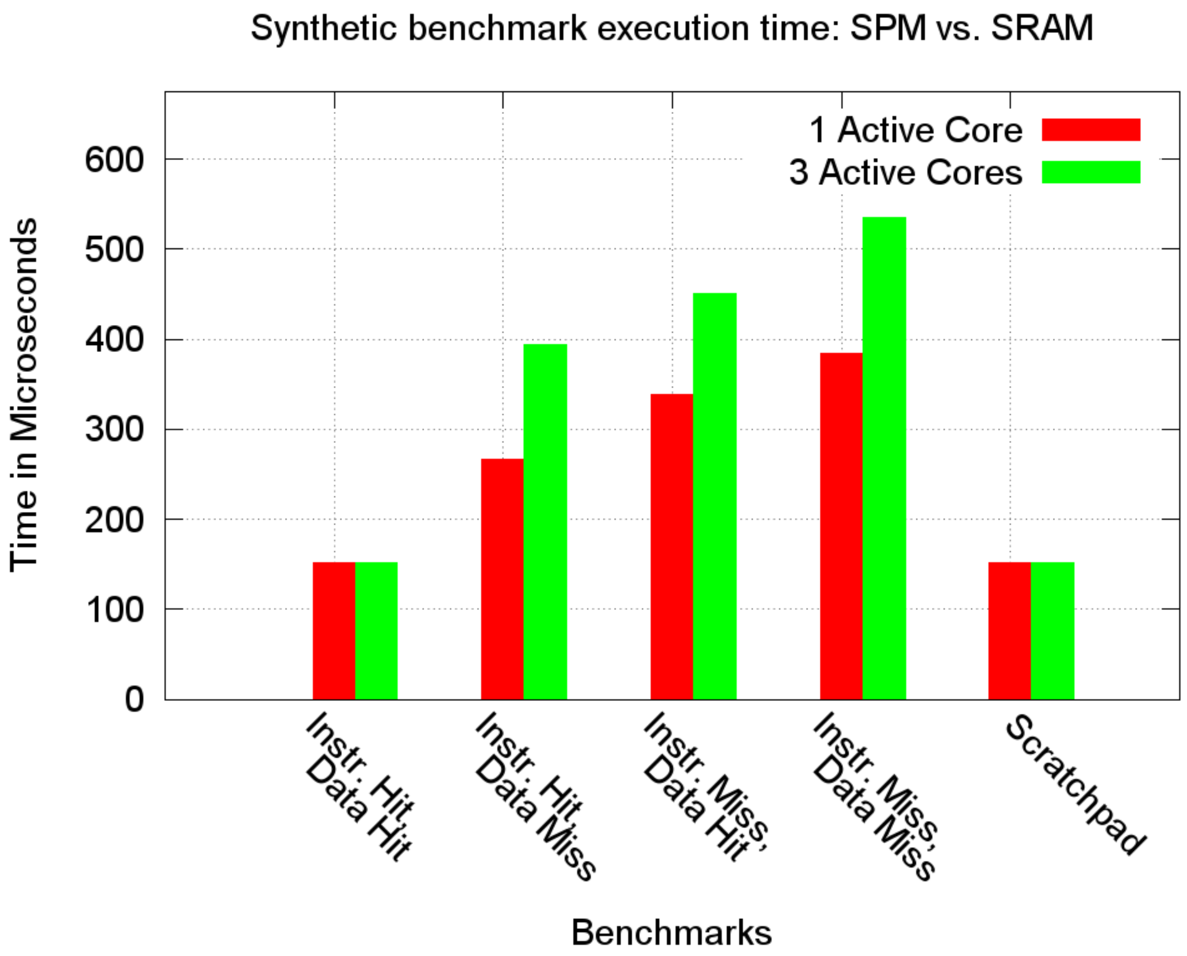

Figure 6.1: Experimental execution time for synthetic benchmarks.

6.4 Results of 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 the proposed SPM-based execution. The results of normalized execution times are shown in Figure 6.2. 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 6.2 shows the execution time of the full suite of EEMBC automotive benchmarks. Furthermore, Table 6.2 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 the proposed design in real scenarios.

[1]`https://www.wireshark.org/`

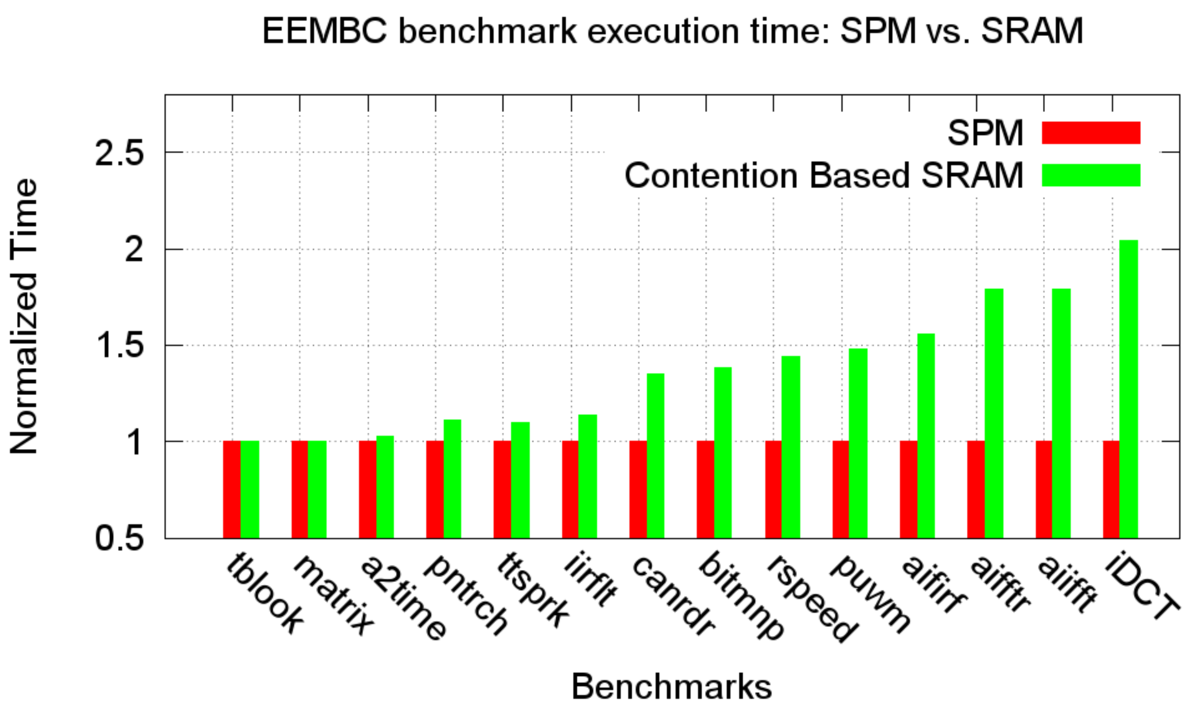

Figure 6.2: Experimental execution time for EEMBC benchmarks.

Table 6.2: Details of EEMBC Benchmarks.

Benchmark	SPM Time (μs)	SRAM Time (μs)	Code Size (bytes)	Relocatable Code Size (bytes)	Data Size (bytes)
tblook	1013	1015	1804	1892	10516
matrix	1053	1054	4430	4774	4488
a2time	1002	1029	2175	2538	1704
pntrch	1036	1145	1000	1398	4924
ttsprk	383	425	4124	4772	8160
iirflt	1040	1189	3288	3512	1000
canrdr	1009	1359	1370	1562	12440
bitmnp	990	1389	3152	3282	1116
rspeed	1012	1457	710	1208	13212
puwm	1036	1540	1716	2500	2412
aifirf	1005	1564	1554	2286	1552
aifftr	916	1642	3720	4458	8448
aiifft	1170	2092	2796	3540	9224
idct	1045	2126	4498	4690	244

6.5 Schedulability Analysis

For the schedulability evaluation of our approach, we compare our system against the contention-based system, in which cores use caches but are left unregulated when accessing main memory. Standard response time analysis is applied on both our system and the contention-based system for the same simulated workload. We have considered

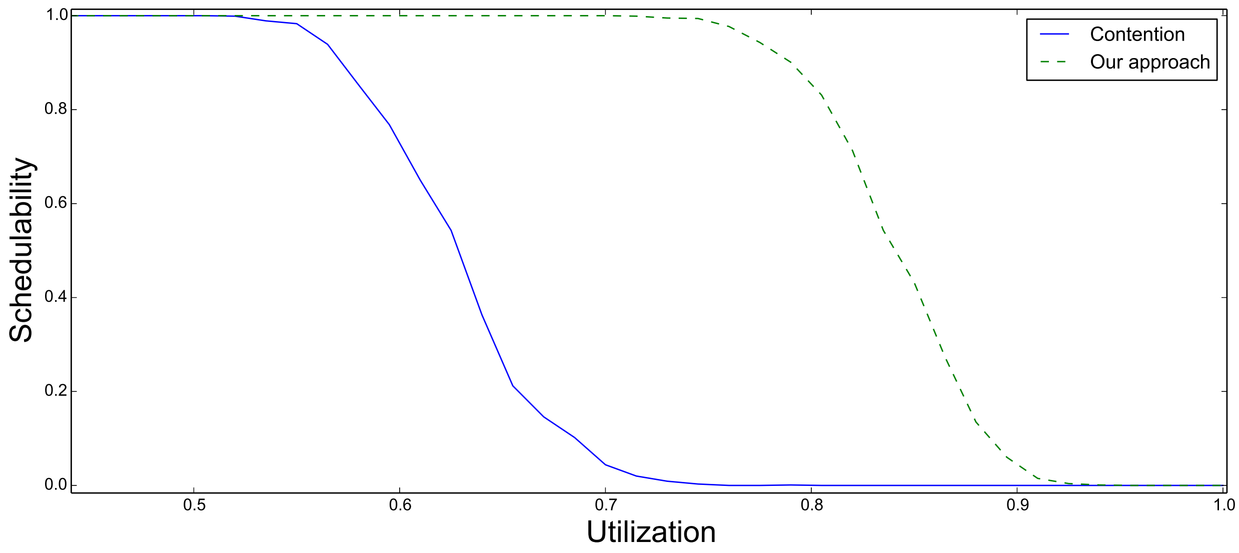

Figure 6.3: Schedulability with SPM-based and traditional scheduling models.

the applications in Table 6.2 to generate sets of random tasks (workloads). Given a system utilization, each application is randomly selected and assigned a random period in the range between 10 ms to 100 ms. The task's utilization is then computed based on the measured execution time of each application and its 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. For the contention-based system, the execution times reported in SRAM column in Table 6.2 are used to represent the worst-case execution time including the contention overhead.

Figure 6.3 shows the result of the schedulability analysis when using the proposed SPM-centric OS versus a contention based SRAM 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 the schedulability of the system increases significantly when the proposed SPM-centric approach is used. Hence, the described SPM-centric OS 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.

Chapter 7

Conclusion

In this thesis, we presented a novel OS design, namely SPM-centric OS. Proposed SPM-centric OS aims at providing predictability for hard real-time applications on multi-core embedded systems. In order to achieve this goal, we combined resource specialization, high-level scheduling of shared hardware resources as well as a three-phases task execution model. Theoretical results on how to perform schedulability analysis of the proposed scheduling strategy were presented. A complete implementation using a commercially available multi-core platform was also performed to assess the feasibility of our design.

Finally, in order to validate proposed OS design, we have combined experimental results from synthetic and automotive EEMBC benchmarks on the considered platform. In addition to the strong temporal predictability achieved by enhancing inter-core isolation, we are able to exploit the performance benefits of scratchpad memories. Hence, a schedulability improvement over traditional contention-based approaches was obtained. As part of our future work on SPM-centric OS, we plan to investigate the following aspects: support for task preemption, inter-process communication, and compliance with standard application interfaces (e.g. AUTOSAR, POSIX).

References

[1] FAA position paper on multiâĂŘ–core processors, CASTâĂŘ32 (rev 0). `http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/castâĂŘ32.pdf`. Accessed: 2015-01-26.

[2] Software techniques for scratchpad memory management. `http://memsys.io/wp-content/uploads/2015/09/p98-sebexen.pdf`. Accessed: 2015-01-26.

[3] Bai, Ke and Lu, Jing and Shrivastava, Aviral and Holton, Bryce. CMSM: an efficient and effective code management for software managed multicores. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–9. IEEE, 2013.

[4] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time I/O management system with COTS peripherals. *Computers, IEEE Transactions on*, 62(1):45–58, 2013.

[5] D. Bui, E.A. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference (DAC)*, pages 274 – 279, June 2011.

[6] S. Chattopadhyay, A. Roychoudhury, J. Rosén, P. Eles, and Z. Peng. Time-predictable embedded software on multi-core platforms: Analysis and optimization. *Foundations and Trends in Electronic Design Automation*, 8(3-4):199–356, July 2014.

[7] Deverge, J-F and Puaut, Isabelle. WCET-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 179–190. IEEE, 2007.

[8] Durrieu, G. and Faugere, M. and Girbal, S. and Pérez, D. G. and Pagetti, C. and Puffitsch, W. Predictable flight management system implementation on a multicore processor. *ERTSS'14*, 2014.

[9] Falk, H. and Kleinsorge, J. C. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, pages 732–737. ACM, 2009.

[10] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[11] Girbal, S. and Jean, X. and Le Rhun, J. and Perez, D. G. and Gatti, M. Deterministic platform software for hard real-time systems using multi-core COTS. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D4–1. IEEE, 2015.

[12] Jean, X. and Faura, D. and Gatti, M. and Pautet, L. and Robert, T. Ensuring robust partitioning in multicore platforms for ima systems. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 7A4–1. IEEE, 2012.

[13] 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*, pages 329–338. IEEE, 2005.

[14] Lu, J. and Bai, K. and Shrivastava, A. SSDM: smart stack data management for software managed multicores (SMMs). In *Proceedings of the 50th Annual Design Automation Conference*, page 149. ACM, 2013.

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

[16] R. Mancuso, R. Pellizzoni, M. Caccamo, Lui Sha, and Heechul Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183, July 2015.

[17] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Architecture of Computing Systems-ARCS 2011*, pages 122–134. Springer, 2011.

[18] 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*, RTAS '11, pages 269–279, Washington, DC, USA, 2011. IEEE Computer Society.

[19] I. Puau and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[20] Bryan Schauer. Multicore processors–a necessity. *ProQuest discovery guides*, pages 1–14, 2008.

[21] Suhendra, Vivy and Roychoudhury, Abhik and Mitra, Tulika. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(4):13, 2010.

[22] Takase, H. and Tomiyama, H. and Takada, H. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1124–1129. IEEE, 2010.

[23] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[24] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013.

[25] S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 75–86. IEEE, 2014.

[26] J. Whitham and N.C Audsley. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 3–12. IEEE, 2012.

[27] J. Whitham, R.I. Davis, N.C. Audsley, S. Altmeyer, and C. Maiza. Investigation of scratchpad memory for preemptive multitasking. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 3–13. IEEE, 2012.

[28] Wilding, M. M. and Hardin, D. S. and Greve, D. A. Invariant performance: A statement of task isolation useful for embedded application integration. In *dcca*, page 287. IEEE, 1999.

[29] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. RTOS support for parallel execution of hard real-time applications on the MERASA multi-core processor. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 193–201. IEEE, 2010.

[30] H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[31] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.