A Dynamic Scratchpad Memory Unit for Predictable 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
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this 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.

Abstract

Scratch-pad memory is a popular alternative to caches in real-time embedded systems due to its advantages in terms of timing predictability and power consumption. However, dynamic management of scratch-pad content is challenging in multitasking environments. To address this issue, this thesis proposes the design of a novel Real-Time Scratchpad Memory Unit (RSMU). The RSMU can be integrated into existing systems with minimal architectural modifications. Furthermore, scratchpad management is performed at the OS level, requiring no application changes. In conjunction with a two-level scheduling scheme, the RSMU provides strong timing guarantees to critical tasks. Demonstration and evaluation of the system design is provided on an embedded FPGA platform.

Acknowledgements

My great gratitude is to Allah the only god, the Almighty who guided me in every step of this work with his infinite graciousness and bounties.

I would like to thank all the kind people around me who made this possible, to only some of whom it is possible to give particular mention here. This thesis would not have been possible without the help and support of my supervisors, Prof. Andrew Morton and Prof. Rodolfo Pellizzoni.

I'm grateful to Prof. Morton for his great kindness, continuous encouragement and extended support. Prof. Morton has guided me through my Master's, and introduced me to several subjects in the embedded domain. The experience I gained from him has not only developed my academic skills, but also has refined my character.

I would also like to express my deepest appreciation and gratitude to Prof. Pellizzoni. His good advice, support and friendship have been invaluable on both an academic and a personal level. The knowledge and experience I gained from him has significantly improved and polished my research skills.

I would also like to express my appreciation and thanks to Prof. Sebastian Fischmeister, and Prof. Hiren D. Patel for serving in my examination committee.

A big thanks to all ECE department staff, who always have been kind, supportive and willing to help. Thanks are also extended to my colleagues and friends in Waterloo; studying and living in Waterloo would not be as enjoyable without them. In addition, I would like to acknowledge the financial and academic support of both King Abdulazize University in Saudi Arabia, and the Saudi Arabian Cultural Bureau in Canada.

The heart words come at last! My special thanks go to my most beloved ones in my life: my parents, my wife and my son. I thank you for your endless encouragement, support, prayers, and love.

Dedication

TO

My parents

My wife

My whole family

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent past, embedded computer systems became significantly involved in many real-life situations [35]. This kind of involvement is expected to continue and affect more aspects of life. Embedded computers are present in wristwatches, cell phones, home TVs, kitchen stoves, pacemakers, car and aircraft engine controllers, medical devices, and many more other examples.

Without doubt, there are cases where computers do a much better job than humans. Computers are fast at calculating and storing and retrieving data. Therefore, computer systems have been deployed to take advantage of these capabilities and replace human operators: for instance, in automatic lighting systems that turn off the lights when they are not needed. Depending on the application, the system can be classified as high critical, low critical or not critical. For example, in Cyber-Physical System (CPS) [31], where computations are done on data retrieved from the physical environment and the actions are also applied to the physical environment, the decision taken by the computer program can be highly critical. In such systems, not only the decision (action) taken has to be correct, but also the response has to be at the right time. When timing is equally important to the functional correctness, the system is classified as a Real-Time System (RTS) [44]. RTS applications or tasks are further classified into Hard Tasks (HTs) and Soft Tasks (STs). Tasks must satisfy the bounded response time (meet deadlines). Otherwise, severe consequences or a failure can happen in the case of hard tasks (critical), or degraded service

quality can be experienced in the case of soft tasks (not critical). Cyber-physical systems are real-time systems by nature. In the domain of the CPS, both hardware and software must be certified in order to guarantee safety and security.

1.1 The Problem

The demand for high-performance computing platforms, especially in the domain of embedded RTSs, has increased dramatically during the last decade. For instance, in automotive systems, new safety features like 'night view assist' require real-time video processing of sensors' data and then warn the driver when an obstacle is detected on the road. Commercial-Off-The-Shelf (COTS) components are increasingly used in an effort to raise performance and lower production costs. However, COTS components such as General-Purpose Processors (GPPs) are not the best fit for RTSs. General-Purpose Architectures (GPAs) are mainly designed to improve average-case performance. In RTSs, the Worst-Case Execution Time (WCET) of applications (tasks) has to be guaranteed. Otherwise, the requirements of RTSs are not met. The predictability, knowing the WCET, of real-time tasks is a fundamental requirement for the correct functioning of the whole system. General-purpose architectures are known for their predictability problem: for instance, the time needed to finish executing a task is not precisely determined and can vary from one run to another. In general-purpose architectures, there are several sources for unpredictability in task execution time, such as dynamic branch prediction, interconnect contention, and inconsistent memory access as in a cache. All these sources cause unpredictable execution time for tasks because they are stateful. In other words, their behaviour is dependent on their current states, which are updated dynamically. For example, the next branch prediction of a dynamic history-based branch predictor is highly influenced by the previously executed branch instructions. The effect of this behaviour is clearer in preemptive multitasking systems where the preempting task can influence the branch predictor in a way that leads to a longer or shorter execution time of the preempted task.

Knowing the WCET is essential to schedulability analysis. Although Real-Time Operating Systems (RTOSs) can provide predictable scheduling policies, no guarantees can

be provided without precise WCET bounds. Whilst the precise knowledge of WCET is difficult in general-purpose architectures, a workaround solution is to estimate a safe upper bound for the WCET. One way to do that is by always assuming the worst-case situation. For instance, in the case of a branch predictor, the worst-case situation will assume a misprediction at every branch instruction. Using this method, a pessimistic upper bound WCET is estimated. However, this method still safely enforces the timing requirements needed by the RTS. The drawback of this method is the wasted time (performance) incurred by the pessimistic estimation, as the ability to schedule more tasks is reduced.

The problem increases when the cost of the wasted time increases. For example, in the case of cache memory, the cost of assuming a cache miss every time the CPU accesses memory is just too large to tolerate. Therefore, several studies have used techniques to come up with tighter WCET bounds. For instance, instead of assuming each access to every instruction in a loop is a cache miss, a miss is only considered at the first iteration, thereby giving a tighter bound. These techniques are highly dependent on static code analysis and precise knowledge of the hardware platform. Not surprisingly, the increased complexity of general-purpose architectures makes this precise knowledge of individual components' states challenging. Thus, the difference between the actual WCET and tighter estimation of it is still significant.

In multitasking systems, it has become more difficult to predict the state of a cache. The cache is shared among all tasks running on the system. The execution of tasks is interleaved, so each task keeps changing the state of the cache during its course of execution. To show a more concrete example, Figure 1.1 depicts a schedule of two tasks (τ_1 and τ_2). When τ_1 runs for the first time, its contents (code and data) are fetched from main memory to the cache; thus, τ_1 suffers external memory latency. If τ_1 is rescheduled and its cache states are kept intact, then it will run out of cache without suffering external memory latency. Hence, a faster execution of τ_1 will be experienced. However, in multitasking systems, it is common for a higher priority task to preempt a lower priority task, with a high possibility that the cache states of the preempted task will be altered (invalidated). For example, τ_2 in Figure 1.1 preempts τ_1 and possibly invalidates some or all cache lines that belong to τ_1. τ_2 causes the cache to save the contents of τ_1 back to main memory and to replace the cache lines by new contents loaded from main memory. As a result, the next time

τ_1 is scheduled, the execution time of τ_1 can be increased significantly, depending on the number of invalidated cache lines. All dirty cache lines are written back to main memory, and their contents are replaced by the new data fetched from main memory. Therefore, the cost related to an unpredictable cache is much more than other components such as a branch predictor in a deep pipeline [17].

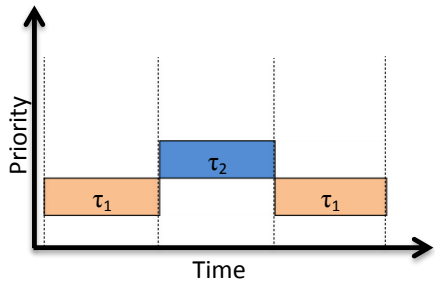

Figure 1.1: Preemption by Higher Priority Task

1.2 Motivation and Research Objectives

As mentioned above, current general-purpose architectures have significant barriers that prevent real-time applications from unleashing the maximum available processing power. Therefore, current software techniques that allow real-time applications to run on general purpose architectures by using pessimistic bounds on WCET under-utilize the processor. On the other hand, one can consider designing a real-time oriented architecture from scratch. This type of architectures can provide an explicit notion of time to the upper levels of abstraction, so that software can be deterministic. Such architecture can be the perfect match for real-time applications whereby the WCET is determined precisely, and the maximum processing performance is exploited. However, this solution is not very practical, as there are complications related to cost and marketing. Large computer hardware manufacturers prefer to invest in platforms that can reach more consumers, such as personal computers, cell phones, and servers. This way, they can pay off the related cost more easily. Instead, modifying current general-purpose architectures, as in

[34], is an appealing approach. However, in this particular example, the modifications were significant. Therefore, adopting it would be less appealing to computer hardware manufacturers, largely because of the dramatic changes needed in the original architecture. In addition, the programming model would also be affected by the hardware changes. In the provided example, programmers need to use new instructions in order to take advantage of the features offered by the architecture, which would compromise software portability.

The work presented in this thesis provides a practical solution that may be appealing to both computer hardware manufacturers and real-time programmers. The proposed solution provides minimal modifications to the typical general-purpose architectures, without affecting the average case performance, which is the main goal in general-purpose architectures. In addition, the programming model has been kept intact, so that applications can be ported to this platform seamlessly. This work tries to close the gap between GPA and RTS requirements by providing an appealing solution for both parties. The cache problem is the main focus of this thesis. However, the same design principles (minimal modifications in original architectures) apply to other components as well.

The proposed solution in this thesis is a hardware-software technique to dynamically manage on-chip local memory for predictable multitasking embedded real-time systems. ScratchPad Memory (SPM [7]) is used in addition to a cache. Tasks are classified into two groups, Critical Tasks (CTs) and Non-critical Tasks (NTs). The critical tasks run out of the SPM and the non-critical tasks run out of the cache, hence, as shown later, maximizing the performance of both types. Execution of the critical tasks is very predictable as they run out of the SPM. The Real-time Scratchpad Memory management Unit (RSMU) is an optimized hardware component that is integrated into the architecture and allows dynamic management of the SPM at the OS level. A novel dynamic scratchpad management algorithm for multitasking systems is designed to hide the latency of loading and unloading the SPM by overlapping it with task execution on the CPU. The result is higher schedulability. The applicability of this approach is discussed in a two-level scheduling framework, where critical tasks are in the higher level. More details are provided in Chapter 3.

The main objective is to provide predictable execution time for critical tasks with minimum impact on cost and performance. The significance of this design is in its simplicity. It involves minimal hardware and software modifications. In addition, it provides

a good abstraction of the underlying hardware while providing precise timing for critical tasks. Furthermore, this platform offers seamless software porting, as no special compiler or extra instructions or code annotation is needed. The technique presented in this thesis simplifies the WCET analysis of real-time multitasking systems. More details about the overall picture of the design are provided in Chapters 3 and 7.

1.3 Thesis Content

A brief review of relevant studies reported in the literature is given in Chapter 2. The review starts with a short comparison between cache and SPM. Then it focuses on two main points. One point reviews different approaches to managing SPM in embedded domains. The other point reviews works that tried to solve the cache predictability issue, either by integrating SPM into the solution or without using it. Chapter 3 depicts the overall picture of how the proposed technique works, while Chapter 4 shows how RSMU and SPM are managed and integrated into a typical computer system. The details of the hardware architecture are given in Chapter 5. Similarly, the details regarding software are explained in Chapter 6. Chapter 7 details how the schedulability analysis is done. After that, an overall platform evaluation, including hardware, software and schedulability, is given in Chapter 8. This chapter also discusses the experimental results. Chapter 9 concludes the thesis and highlights future work.

Chapter 2

Background

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. Therefore, any task that runs out of the SPM is guaranteed to have a predictable execution time. The SPM is also better than the cache in terms of area and power efficiency [13]. Figures 2.1 and 2.2 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 [29].

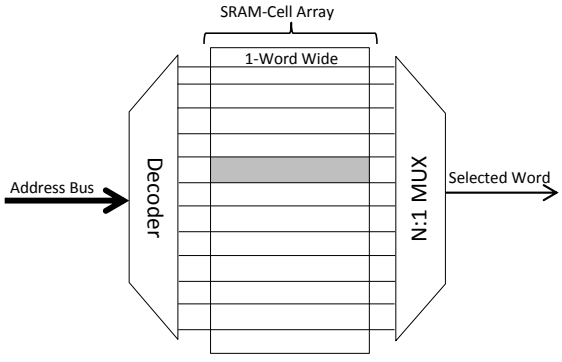

Figure 2.1: SPM Architecture

The SPM memory, as shown in Figure 2.1, 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 [13]. Each data memory (SRAM array) is indexed in the same way as the SPM. That means cache is, at least, n times more complicated and consumes more power than SPM, where n is the number of data memories the cache utilizes. For instance, if the cache has 32-byte cache lines (8 words), there will be 8 data memories. In addition, valid-bit and tag arrays are indexed for each access. 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.

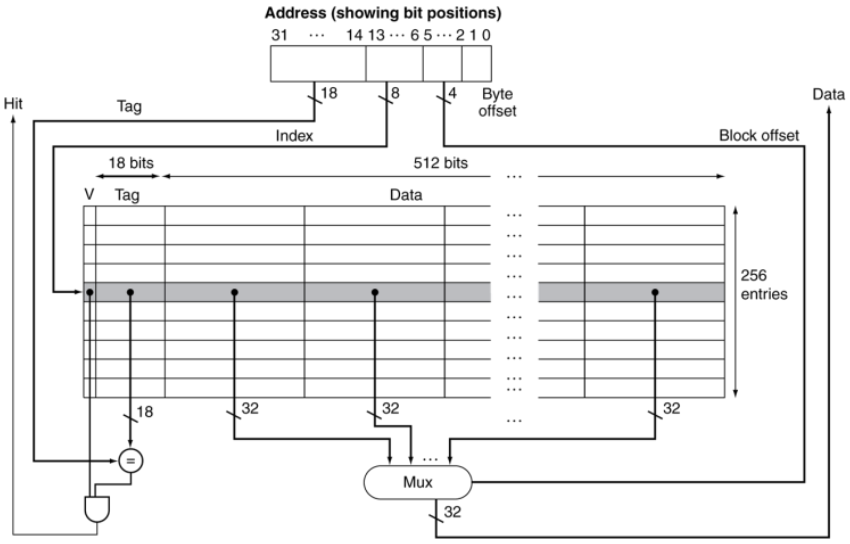

Figure 2.2: Direct-Mapped Cache Architecture (From [1])

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 full associative caches, as shown in Figure 2.3, 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 caches have 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. [13] is a study in the embedded domain that compares energy, area and performance of both SPM and cache. The study indicates that SMP achieves better than cache almost on all counts.

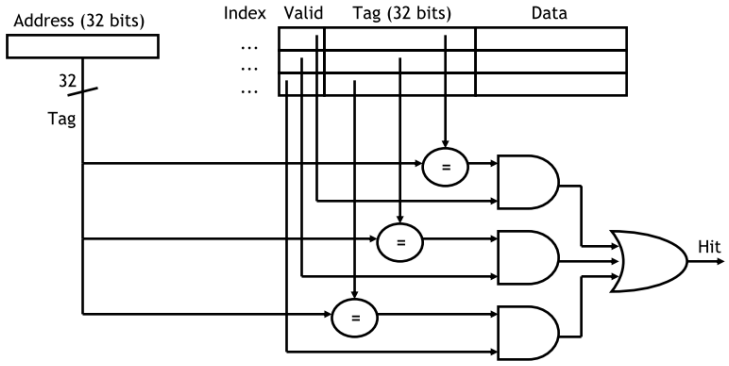

Figure 2.3: Full Associative Cache Architecture (From [1])

Faster access, smaller area, lower power, and predictable access time make scratch-pad 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.

Researchers in the real-time domain have focused on the predictability problem caused by caches. They attempt to solve it by either using smart allocation algorithms for the cache, or by adding scratchpad memories as a hardware component to the platform. Researchers in the general embedded domain, not really interested in predictability, have focused more on ways to utilize the scratch-pad memories in embedded platforms due to its interesting qualities such as power efficiency and performance. The following two sections review recent work in the general embedded domain and the real-time embedded domain.

2.1 General Embedded 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 employing 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. [23][22][42][38] used conventional MMU to manage the SPM at runtime. These works were based on simulation only. Managing the SPM at runtime allows different applications to utilize the SPM dynamically resulting in better resource utilization.

B. Egger et al [22] 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. Although this work is a significant step toward dynamic runtime managed SPM, it has some drawbacks. The first drawback is that it is based on simulation and we don't know what complications can occur in real implementations. Second, it requires off-line profiling of the application in order to make it optimized for the SPM. In addition, 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. Finally, this work did not investigate the case of multitasking system, which is more realistic than a single task system.

P. Francesco et al [23] 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 the SPM in dynamic applications. The study even showed an improvement in the performance. Figure 2.4 shows the hardware extensions that they proposed. The DMA component, which is in the Transfer Engine of the extended figure (right side), was used to relieve the CPU of copying data to/from the SPM. Loading applications code into the SPM is 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 [38], 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

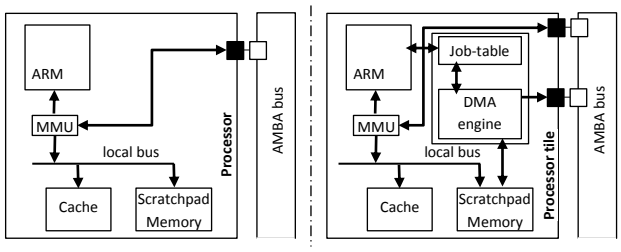

Figure 2.4: Hardware extensions for scratchpad management:(left) original, (right) extended. From ([23])

`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.

Work presented in [42] 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 [22], 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 [22], this work is only

based on simulation, not a real platform. Furthermore, there is no DMA component to load/unload the SPM, which results in reduced performance.

The advantage of using the MMU for the SPM management is that it can be done dynamically by the OS at runtime. Thus, it becomes transparent to application programmers. The disadvantage is that the conventional MMU is not designed to serve real-time embedded systems and introduces performance barriers, as explained later in Chapter 5. Our solution, as introduced in the following chapters, overcomes the aforementioned problems and barriers while enforcing reasonable restrictions on critical tasks towards a guaranteed predictable execution.

2.2 Real-time Embedded Oriented Approach

For the embedded real-time domain, predictable execution time is the more important characteristic of the scratch-pad memory. However, in order to get this predictability the SPM itself has to be present in the system in the first place. Therefore, researchers in the embedded real-time domain also explored ways toward predictable execution based on cache utilization only.

Suhendra and Mitra[48] explored the effect of locking and partitioning of L2 shared cache on predictability in multi-core 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 type 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. It can be concluded from this research that adopting this technique is not desirable as the predictability gained comes at a significant cost in performance and it does not scale well. In addition, lockable caches are considered specialized features and not available on all embedded systems. Using an SPM that is more power efficient and occupies a much smaller silicon area may be a better approach.

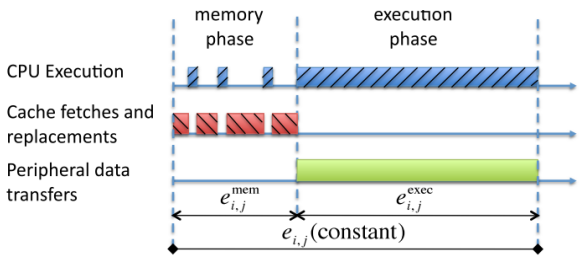

Figure 2.5: PREM Predictable Interval with Constant Execution Time (From [43])

Pellizzoni et al [43] recently introduced the PRedictable Execution Model (PREM) that enforces predictable execution for some tasks. First, In PREM, any task can have one or more critical code partitions, called Predictable Intervals (PIs). The PIs are not preemtable and are preloaded into the cache. Therefore, PREM achieves constant execution time for the PIs. See Figure 2.5. PREM is based on code annotation and uses compiler extensions to inject extra code before the PIs in order to pre-fetch all required data into the cache. This technique divides the CPU execution into two phases, memory phase and execution phase. In the execution phase, the CPU does not initiate any memory request. As a result, not only do PIs not suffer cache misses, but IO peripherals can also be scheduled to access main memory without suffering bus contention from the CPU. However this technique requires code annotation and support from both compiler and OS which might make porting applications a concern.

Other works such as [46] investigated the use of the SPM in embedded systems. [46] quantitatively compared memory-mapped SPM to locked cache in terms of WCET. They use a compile-time algorithm to dynamically load tasks into on-chip memory as needed. Similar to the work by Pellizzoni et al [43], they inject 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 close to each other. However, this comparison might be unfair as loading tasks into SPM uses the CPU which adds more overhead than the cache.

Other approaches attempted to manage SPM at runtime to achieve predictable execution time. J. Whitham and N. Audsley [53] proposed simplified runtime load/store opera-

tions 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 advantage of this approach over the compile-time approaches is that it avoids problems such as dynamic data limitation due to pointer aliasing, pointer invalidating and object sizing making program pointer analysis unnecessary. This significantly simplifies the WCET analysis, especially for applications that use dynamic data allocation. On the other hand, programmers use the specialized instructions to load and map data objects to the SPM, which is difficult to manage especially in multitasking systems. In addition, 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.

The most recent and appealing work to our knowledge in this domain is by J. Whitham et al [55] [54]. It 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 B fixed-size blocks. The top n blocks are stored in the SPM, and the remaining block are stored in 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 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 become not among the top n blocks of the stack. The process is reversed when ending a task. All task's 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. Figure 2.6 shows this mechanism.

Carousel shifts the responsibility of block swapping to tasks. Each task swaps out some blocks to free space when it starts, then it loads (opens) its own blocks, code and data, before it starts executing. After the task finishes executing, it saves (closes) its own blocks

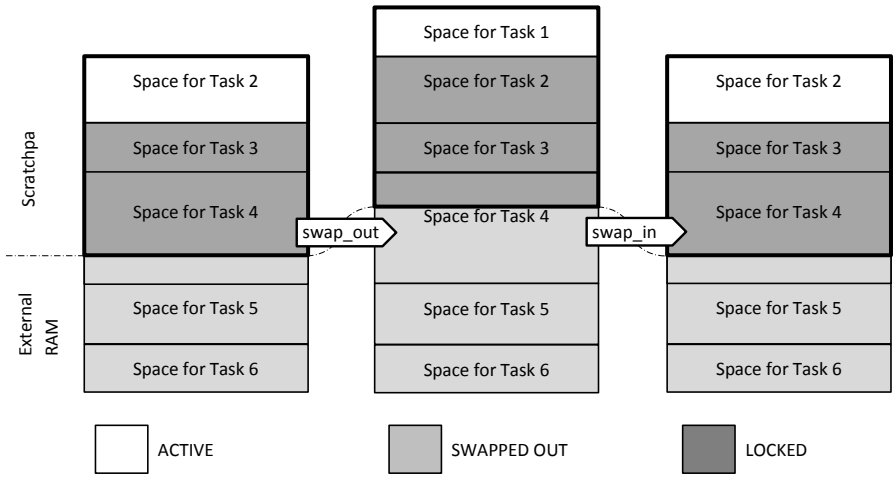

Figure 2.6: Carousel stack mechanism (From [55])

back to the main memory. Finally, the task swaps in all the blocks it swapped out earlier, so the stack is restored to the state earlier to the task invocation. Using Carousel, tasks are scheduled according to Rate Monotonic in a stack manner. Therefore, tasks are executed in a last-in first-out (LIFO) order, having the highest priority task at the top of the stack. Carousel allows each task to allocate any amount of the SPM it requires while preventing inter-task interference. However, the cost associated with moving blocks between the main memory and the SPM seems high. Figure 2.7 shows the overhead associated with each task invocation. From the figure, assuming that the blocks in each swappable set are adjacent, each task invocation incurs at least six DMA operations: two p blocks (code plus data), two y code blocks and two z stack blocks. In addition, a special operating system is designed for carousel architecture, called "Carousel OS". Supporting the architecture in existing operating systems is a better approach to facilitate easier software porting.

From hardware prospective, Carousel used DMA to move tasks' blocks between the SPM and the main memory. However, the CPU stalls until the transfer finishes. The Carousel mechanism that requires a task to load/unload itself impedes the ability of overlapping DMA transfers with the CPU execution. In addition, Carousel used a translation

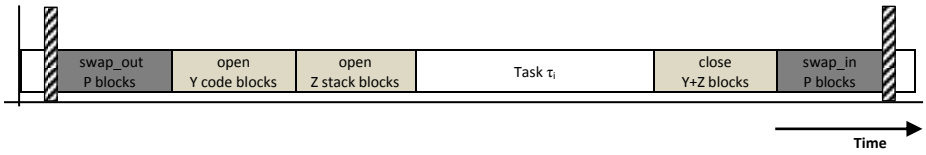

Figure 2.7: Carousel task invocation (From [55])

unit to virtualize block addresses. The translation unit translates all the n blocks stored in the SPM in parallel. This becomes a concern since the performance of the translation unit is dependent on the number of blocks it translates in parallel. Having many small blocks decreases the maximum operating frequency of the translation unit, while having few large blocks reduces the space utilization of the SPM. In addition, Carousel architecture requires a separate SPM only for the OS. This is required as the OS has to be in the SPM all the time, and it is not possible to put the OS on the Carousel SPM with other tasks. This approach is less appealing to adopt as the optimum size ratio between the two SPMs has to be determined at the hardware level.

This chapter reviewed several approaches either to manage the SPM dynamically, or to solve the predictability issue associated with cache memory. Each work has a contribution to solve the problem we are targeting from a certain perspective. Our work, introduced in the following chapters, is aimed to be more comprehensive and solves the problem from different aspects at the same time. Our solution avoids most of the problems highlighted in this review, and proposes a hardware-software platform that helps to bring real-time applications to general-purpose architecture with minimum impact on cost and performance. The next chapter introduces our solution at the system-level.

Chapter 3

System Model

The solution introduced in this thesis is a simple, novel and effective hardware-software technique that achieves the predictability required in real-time systems, without compromising performance and cost. It targets hard real-time systems running on single-core processors with the belief that it will scale well to multi-core systems. It is more suitable for critical task sets that have relatively small working sets, such as control tasks available in engine controllers.

A system with tasks of different criticalities is considered, similar to the Integrated Modular Avionic system [47]. For simplicity only two sets are considered: a set of periodic critical tasks, and a set of non-critical tasks. The main objective is to protect the critical tasks from interference of other critical or non-critical tasks. The critical tasks run out of the SPM, and the non-critical tasks run out of the cache. With this setup, the critical tasks have a predictable execution time, precisely determined WCET, without degrading the performance of other non-critical tasks.

A two-level scheduling scheme is adopted (Figure 3.1). Each critical task (τ_i^c) is assigned one or more fixed-time slots within a system period (major cycle). Non-critical tasks (τ_i^{nc}) are executed within partitions. Each partition gets one or more fixed-time slots. Within each partition, non-critical tasks can be scheduled using any scheduling policy. A fixed-priority scheduling used in [47] has been adopted to schedule non-critical tasks among their assigned partition. To illustrate how this scheduling works, in Figure 3.1, the level-1

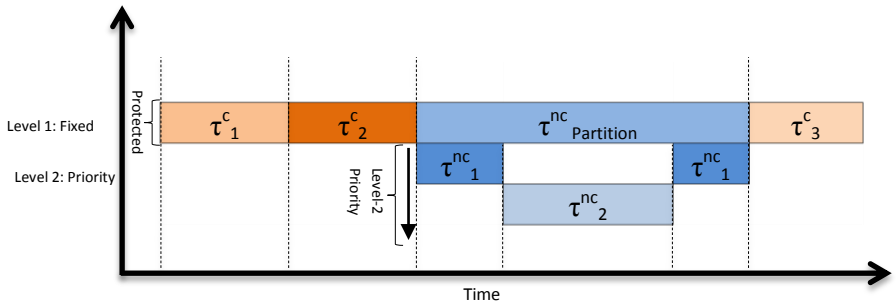

Figure 3.1: Two-level Schedule

schedule is fixed (predetermined off-line), and has the highest priority. At this level, critical tasks (τ_i^c) are executed within their assigned slots. In addition, one or more fixed-time slots, called non-critical partitions, are considered in this level. On the other hand, the level-2 schedule is priority based and has a lower priority than level-1. Non-critical tasks, such as τ_1^{nc}, τ_2^{nc}, are executed in this level, within the assigned non-critical partitions. Each non-critical task has a fixed priority, and its slot assignment is determined dynamically at runtime. This scheduling scheme provides the protection needed for critical tasks in level-1, while giving the flexibility to schedule non-critical tasks using any scheduling policy in level-2.

Figure 3.2 depicts the mechanism used to load tasks into the SPM. The SPM contents are shown at the end of each time slot. The SPM is partitioned into three dynamically sized partitions. One for the operating system and the other two are for critical tasks (see Chapter 4). Figure 3.2 shows only the two partitions dedicated for critical tasks. During the execution of a critical task, the system uses DMA to load the code and data for the next scheduled critical task into the SPM. To free space, the system also unloads the previously executed critical task and writes back the modified content (data) to the main memory. For instance, τ_2^c (code + data) is loaded into the SPM while τ_1^c is running; hence, it can run right after τ_1^c without any latency other than that for the context switch. In addition, τ_1^c (data) is unloaded at the same time τ_2^c is running; hence, a new task can be loaded. There is no need to unload code from the SPM back to main memory, as it is not modified.

As shown in Figure 3.2, no DMA activity is allowed while non-critical tasks are running. Consequently, the non-critical tasks, which run out of the cache, are not interfered with by the DMA activities, leading to better performance. In addition, loading and unloading the SPM while a critical task is running will not degrade the performance of the running critical task, as the SPM is dual ported. Therefore, the critical tasks, which run out of the SPM, are not interfered with by the DMA as well.

This work is different from [55] as it does not stall the CPU in order to load tasks into the SPM. Instead, overlapping the DMA transfers and the execution of critical tasks is considered. This scheme results in better overall schedulability by hiding the latency required by the DMA transfers. Details on schedulability analysis are provided in Chapter 7. Realization of the introduced technique requires hardware-software co-design. The next chapter discusses how hardware and software are integrated to achieve the desired goals, with both performance and cost in mind.

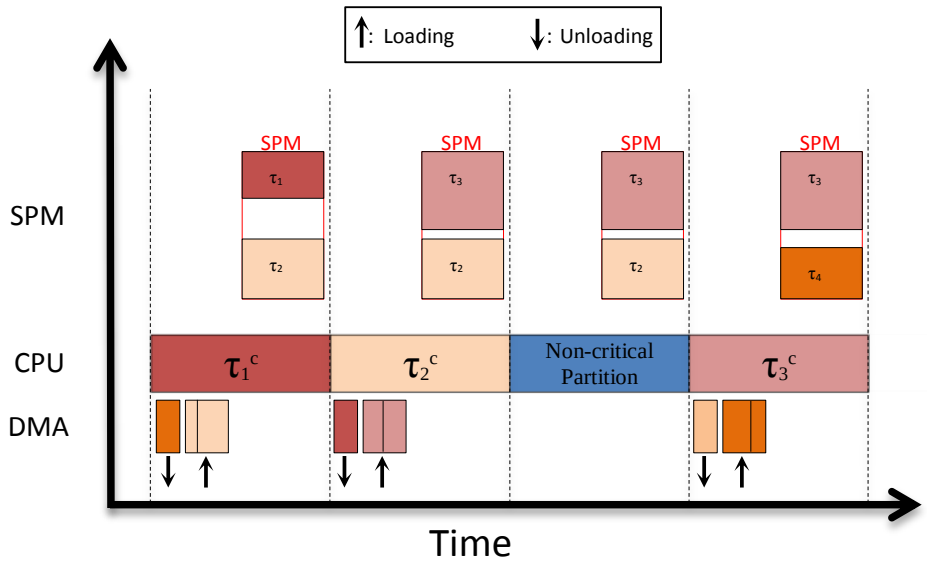

Figure 3.2: DMA Transfer of critical tasks to SPM

Chapter 4

The Real-time Scratchpad Memory Unit

This implementation of the system considers a typical modern 32-bit embedded processor, which has separate instruction and data buses. Therefore, the system consists of two ScratchPad Memories (SPMs) of the same size, along with two Real-time Scratchpad Memory Units (RSMUs). One SPM is for code and the other is for data, equivalent to L1 caches (I-cache and D-cache). These components are integrated into typical computer systems as shown in Figure 4.1. Each SPM is dynamically partitioned by the OS into three main partitions. The first partition is fixed in size, and contains critical parts of the OS and some shared libraries required by critical tasks. The sizes of the other two general partitions are variable, and are managed by the OS to fit the sizes of the loaded critical tasks. The DMA component is assumed to come standard in most modern embedded platforms. Since the sizes of SPMs are limited, and several critical tasks are loaded into the SPMs dynamically, a memory management unit is needed to manage each SPM. The I-RSMU manages requests targeting the I-SPM. Similarly, the D-RSMU manages requests targeting the D-SPM. When a critical task is loaded into the SPMs, the RSMUs translate the original addresses into the new addresses, into which the task has been loaded. The RSMUs handle the address translation very efficiently (see Chapter 5 for details). Managing the DMA and RSMUs is handled by the OS. In this particular implementation, the FreeRTOS

[18] has been used with minimal changes. However, the same concept can be realized using any other RTOS. Details on software implementation are presented in Chapter 6.

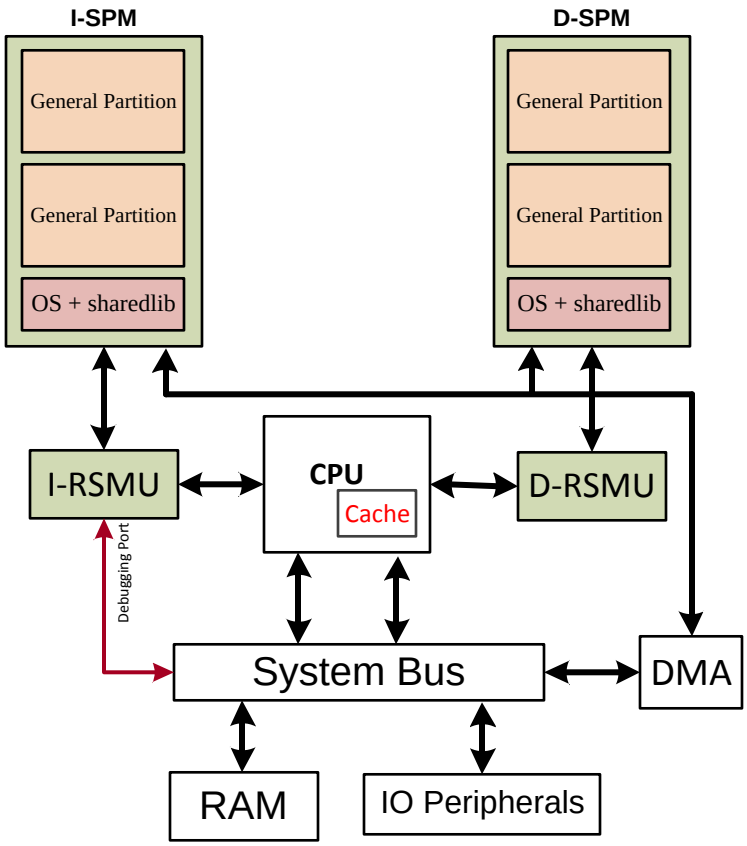

Figure 4.1: System-Level Block Diagram

From the hardware side, SPMs are dual-ported to allow simultaneous access from the DMA and the CPU. The CPU accesses the SPMs via the RSMUs. Each RSMU allocates much more address space than the physical SPM connected to it requires. As Figure 4.2 depicts, each RSMU has a physical address space and a virtual address space. The

SPMs occupy only the first N addresses of the RSMUs' address space, where N is the size of the SPMs in words. The first N addresses are in the physical address space, and addresses beyond that are in the virtual address space. Requests to the virtual address space require translation. The RSMUs are responsible for translating addresses from the virtual address space to the physical address space. (Refer to Chapter 5 for more details about how the RSMU works.) The critical tasks are compiled and linked against the RSMUs' virtual address space. This technique facilitates reuse of the free physical space for different critical tasks. Other non-critical tasks are just compiled and linked normally against physical addresses in the main memory. Some critical components of the OS, such as the scheduler, are linked to the physical address space of the RSMUs, so that they run out of the SPMs without translation.

A linker script is used to put each critical task into unique output sections (code and data) in the executable binary. For instance, a critical task named "tsk1" is put into "tsk1.text" and "tsk1.data" output sections. At runtime, the OS moves the contents of these sections to the SPMs, using DMA, according to the schedule. Upon a task's invocation, these sections will be mapped, using the RSMU, to the physical addresses into which the task has been loaded. For example, the task "tsk1" is first loaded into some SPM physical address while another critical task is running. After that, when the OS is switching to the task "tsk1", the OS maps "tsk1" to the physical address into which it has been loaded. This is done by instructing the translation units, which are the RSMUs, to translate addresses of this task. Hence, the CPU can execute "tsk1" out of the SPMs correctly. One thing about accessing the RSMUs is that CPU accesses to the RSMUs are uncached. Thus, the cache contents of non-critical tasks are not affected by the critical tasks.

Porting software to this system is straightforward. Software does not need code annotation, special machine-instructions or special compilers to take advantage of the introduced system. However, additional lines in the linker script are needed to define different sections for different critical tasks. Having the address translation at the hardware level and managed by the OS keeps the program layout unaltered and makes the address translation transparent to application programmers, just like in a cache. More over, unlike with static approaches, the use of dynamically allocated data is permitted. On the other hand, the

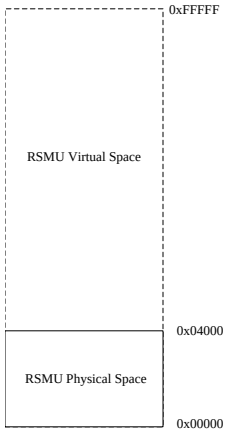

Figure 4.2: RSMU Address Map

proposed system imposes some constraints on critical tasks. First, the sizes of critical tasks have to be limited by the sizes of the on-chip SPMs, which is usually feasible for critical tasks. In addition, it is also important to consider the time needed for DMA to load/unload critical tasks into/from the SPMs, as the DMA time constraints the execution time of the running critical task. For instance, the execution time of a critical task has to be long enough to allow the DMA to finish unloading and loading the SPMs. These constraints are considered in schedulability analysis (Chapter 7). Unlike a conventional translation unit such as an MMU, the RSMU is designed specifically for real-time embedded systems. The RSMU is cost effective and does not incur timing anomalies. More details on the hardware architecture are provided in the next chapter.

Chapter 5

System Architecture

The main objective behind designing the Real-time Scratchpad memory Management Unit (RSMU) was to have a piece of hardware that is able to translate addresses in an efficient way suitable for Real-time systems. Therefore, unlike MMU, the RSMU translates addresses in constant time (no lookup tables). In addition, the RSMU has minimal impact on the critical path and silicon area. Using the RSMUs along with DMA and the on-chip SPMs provide predictable execution of the critical tasks seamlessly. Figure 4.1, in Chapter 4, shows the components added to the conventional embedded platform. The added components are two SPMs along with two RSMUs. The SPMs are constructed from on-chip memory blocks. Each SPM is dual-ported. One port services the CPU requests, and the other port is used for DMA transfers. As shown in Figure 4.1, the RMUs are connected to the CPU via dedicated buses. This is done because the RSMUs are meant to be level-1 cache alternatives.

Figure 5.1 depicts the internal architecture of the RSMU. It is small and simple, yet effective. The RSMU has minimal impact on the operating frequency, due to its simple design. More importantly, unlike the Scratchpad Memory Management Unit (SMMU) in [53], the performance of the RSMU is independent of the number of tasks it translates. As shown in the figure, the RSMU consists of two 20-bit registers. One is the Translation-Edge Register (TER), and the other is the Active-Task Offset Register (ATOR). The TER determines when the RSMU is supposed to translate an address. For example, if

the required address is greater than the value stored in the TER, then the translation is performed. The translated address is based on the value stored in the ATOR. The equation for translating addresses is $SPM_{physical} = (CPU_{virtual} > TER)?CPU_{virtual} - ATOR : CPU_{virtual}$. For example, if the value stored in the ATOR is "0x2000" and the value stored in TER is "0x4000", then the incoming CPU address, such as "0x5000", will be translated into "0x3000". Note that address "0x5000" is in the RSMU's virtual address space; thus, it needs to be translated. The RSMU can be configured to have signed math in order to have negative offsets. Usually the TER is set to the end of the RSMU's physical space. However, it is flexible and can be redefined by the OS at runtime.

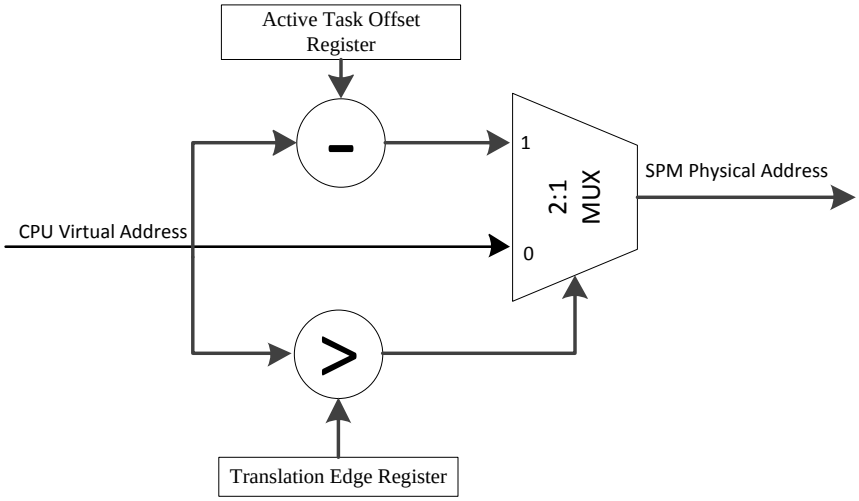

Figure 5.1: Real-time Scratchpad Memory Unit (RSMU) structure

The RSMU is configured to cover limited address space, e.g., 1 MB, and not the whole CPU's address space, as in conventional MMU. Therefore, the RSMU intercepts only the requests targeting it. As a result, the RSMU uses smaller address comparators, which leads to a faster hardware. On the FPGA test platform, accessing the SPM through the RSMU was achieved in one clock cycle, with a small reduction in the operating frequency compared to the reference design that does not have any translation unit. The hardware evaluation in Chapter 8 provides more details. The reduction in the operating frequency caused by the RSMU is deemed small compared to other translation units such as SMMU.

Each RSMU is exposed to software through the memory-mapped configuration registers, TER and ATOR. The addresses of these registers are determined in the hardware design. Any OS can utilize the RSMUs by having a simple device driver to communicate with them through the configuration registers. The OS has the flexibility to set any value to these registers.

It is worth mentioning that a problem was faced while debugging this architecture: the software debugger was not able access the I-SPM in order to inspect its contents. As we know, in the debugging environment, it is always beneficial to be able to inspect the contents of the memory, e.g., setting breakpoints and watching variables. This implementation uses Altera's NIOS-II processor as the CPU. The debugger used, Altera's JTAG debugger, accesses memory-mapped components either through the CPU's data bus, which is connected to the system bus, or through the data-tightly-coupled memory interface, which is the dedicated bus connected to the D-RSMU. Consequently, the I-SPM becomes inaccessible to the debugger, as the I-SPM is not connected to either the system bus or to the data-tightly-coupled memory interface. Therefore, the I-RSMU, connected to the instruction-tightly-coupled memory interface, can be configured to have an extra debugging port to allow the debugger to access the I-SPM. This debugging port is connected to the CPU's data bus through the system bus. Allowing the debugger to access the I-SPM though the I-RSMU enables setting breakpoints in the critical tasks at the original virtual addresses. For instance, the debugger can set breakpoints in several critical tasks according to the original program layout. However, the breakpoints' addresses are also translated at runtime by the I-RSMU to the correct physical addresses. The debugging port, which is used only for debugging, does not have the same timing restrictions that the regular port has, so it will not add any latency to the critical path.

5.1 Addressing Scheme

From the software prospective, the compiler can see three memory regions: the main memory, which is the external SDRAM, the I-RSMU connected to the I-SPM, and the D-RSMU connected to the D-SPM. The RSMUs' memory regions cover both the physical

and virtual address spaces of the RSMUs, which are defined in hardware. The boot loader copies each output section in the executable binary to the targeted memory region. Sections that target physical addresses, such as main memory and RSMUs' physical address space , are handled by the boot loader. On the other hand, sections targeting the RSMUs' virtual address space are handled by the OS at runtime. As mentioned, any code or data linked to the RSMUs' virtual address space need to be translated before they can be accessed, a step usually managed by the OS.

On the hardware side, as shown in Figure 4.1, The CPU has two sets of buses, instruction and data buses. Each set occupies a range of the CPU's address space. For example, the data bus connected to the system bus occupies address from "0x00000000" to "0x04000000", while the data bus connected to D-RSMU occupies addresses from "0x05000000" to "0x05100000". The interconnect is responsible of placing each CPU's request on the correct bus. As a result, the RSMU will only translate the addresses placed on the bus connected to it. Moreover, since the RSMUs are connected to special type of buses, called tightly-coupled memory interfaces. These types of buses bypass the cache circuitry as they are designed to connect to high performance on-chip memories. Consequently, CPU's requests to RSMUs are uncached. The CPU allows overlap of instruction and data buses, e.g., both instruction and data buses can access the same addresses. However, overlapping the addresses of the same type of buses is not allowed, such as instruction bus and instruction-tightly-coupled memory interface. Therefore, the debugging port connected to the I-RSMU is possible because it overlaps the address space occupied by the instruction-tightly-coupled memory interface connected to the I-RSMU and the part of the address space occupied by the data bus connected to the system bus. For instance, the CPU accesses addresses in the I-RSMU through the instruction-tightly-coupled memory interface for instructions fetch, whereas the CPU can also access the same addresses in the I-RSMU through the data bus for debugging.

5.2 System Initialization

When the system is powered-up, the boot loader loads the software-system components to their loading addresses, e.g., main memory or RSMUs' physical addresses. Some critical components, such as the OS scheduler and some shared libraries, are placed in the RSMUs' physical address space, so that the boot loader loads them into the dedicated partitions in the SPMs. After that, the OS runs normally out of the main memory. As a part of the booting process, the OS initializes the system hardware and configures it to the desired mode. According to the schedule, the OS partitions the SPMs to handle two critical tasks and the other critical software components. After that, the OS instructs the DMA to load a critical task into the SPMs. Once the system is initialized, the scheduler can run out of the SPM and invoke the first task. Upon each context switch, the scheduler swaps in and out critical tasks as needed. The next chapter provides more details on software implementation including how the swapping mechanism is done.

Chapter 6

Software Build Flow

The proposed system is designed to keep the software model simple and straightforward. The system is compiler independent, and there are only two restrictions on the software side. First, each critical task must be size limited by the size of the SPM's partition that is going to contain it at runtime, which can vary depending on the scheduled critical tasks. The second restriction is that programmers have to keep each critical task in one or more source files, and do not mix them in one source file. After that, a few lines corresponding to each critical task have to be added to the linker script. These lines put all input sections (code and data) of each critical task's object files contiguously into two output sections (code and data) named with the same name of that task, e.g. "tsk1.text" and "tsk1.data". This trick allows the OS to move the whole task as two blocks only. In addition, the linker script exposes several linker symbols to be read by the OS. These symbols help the OS figure out what is in the system. The most important symbols exposed by the linker script are the loading and virtual addresses of each critical task's section, the size of each critical task's section, and the SPM sizes.

Each SPM is partitioned into three main partitions. The first partition is occupied by the critical parts of the OS, such as the context-switch and scheduler routines. In addition, the OS reserves a part of this partition to keep what we call the OS-SPM heap in order to provide a dynamic memory allocation for critical tasks. This partition also contains some shared libraries used by critical tasks if needed. Depending on the required

features and libraries needed by the critical tasks, 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. The second and third partitions are reserved to run critical tasks. The idea behind dedicating only two partitions to run critical tasks is to avoid the complications associated with dynamic allocations of partitions with different sizes, such as memory fragmentation. On the other hand, the two-partition scheme provides a very simple and direct allocation method and serves well with the swap mechanism. For instance, a critical task can be running out of the first partition while another critical task is loading to the second partition. After the critical task in the first partition finishes executing, a new critical task can be loaded into the first partition while executing the critical task previously loaded into the second partition and so on. Consequently, there will always be a free partition to load a critical task into. Critical jobs are assigned to one of the two partitions according to an off-line schedule. The off-line schedule takes care of the tasks' sizes and execution order. Therefore, the OS uses the information provided by the schedule and resizes the partitions at runtime accordingly.

6.1 OS Support

As one of the goals in this thesis is to provide a practical and comprehensive solution, an OS support module has been developed to show the proposed architecture can be integrated into exciting Real-Time Operating systems (RTOSs). The importance of supporting the architecture at the OS-level is to abstract away all the hardware details from applications programmers, hence making porting software easier. In addition, instead of having a self-managed hardware, moving the hardware management to the OS-level is more cost effective and provides more flexibility. Simple and small hardware is more adoptable and easier to manage.

In this implementation small changes have been made to the FreeRTOS kernel. First, a device driver for the RSMUs has been developed. The driver exposes several macros in order to facilitate control of the RSMUs at a reasonable level of abstraction, e.g., `map_section(virt_addr, phys_addr, size)`. Second, a check-up routine, to be

executed at system initialization, has been developed to detect whether a certain critical task's compiled-size exceeds its allowed size. Using the linker symbols provided by the linker script, the check-up routine is able to verify the compiled code parameters by comparing them with the parameters defined by the system engineers. This capability provides a shorter design cycle as any design restrictions are identified earlier. If the check-up routine succeeds, the OS continues to initialize the system and then runs the scheduled tasks as described earlier.

FreeRTOS is very modular. Therefore, modules called RSMU and Application Management Plug-in (AMP) have been developed. These modules can be added to the compilation process by setting the `configUSE_RSMU` and `configUSE_AMP` macros in `FreeRTOSConfig.h` to the value "1". By turning these macros on, the task creation, task scheduling, context switching, and dynamic memory allocation of the FreeRTOS are changed to the way our system works. Since FreeRTOS does not distinguish between critical tasks and non-critical tasks, the AMP plug-in wraps some FreeRTOS original APIs, such as `xTaskCreate`, to handle critical tasks besides non-critical tasks. As a consequence of wrapping the original APIs, the original APIs of FreeRTOS are not modified and are kept as is. The AMP uses the original APIs with the preferred parameters. For example, when the AMP is used to create a critical task, the AMP allocates its stack on the SPM and gives it the highest priority.

No real modification is needed to the FreeRTOS scheduler to adapt our scheduling policy. The only trick is to know the next two scheduled tasks. This becomes important in order to pre-load critical tasks, using DMA, into the SPM while the next scheduled task is running. At context-switch, the current task and the next two scheduled tasks are checked to determine whether they are on the SPM. Depending on the location of the task that was running, the DMA will be instructed to move the dirty data back to main memory if necessary. If the next scheduled task has to run out of the SPM, then the RSMU is instructed to activate that task (map it). Moreover, when the task that is scheduled to run right after the next task has to run out of the SPM as well, then the DMA is instructed to pre-load that task into the SPM.

The DMA transfers are performed in background and do not not impact the context-switch time (CPU time). Meaning that at context-switch the DMA is only set-up to start

the transfer jobs. Due to the implementation of the DMA core used in the experiments, the DMA core can support only one transfer at a time. Consequently, The DMA has to be set-up three times to unload and load tasks into the SPMs. At context-switch the DMA is set-up to unload the data of the previously running critical task. After the DMA transfer is done, the DMA generate an interrupt to confirm the transfer. The running critical task is interrupted and the DMA is then set-up to load the code section of the next scheduled critical task. Similarly, the DMA interrupts the critical task for a second time, and then it is finally set-up to load the data section the next scheduled critical task. The overhead of setting-up the DMA three times is considered in the schedulability analysis in Chapter 7.

As mentioned earlier, at the system initialization phase, information about each critical task, such as task's size, virtual address and loading address, is extracted from the linker symbols and stored into the tasks' TCB (Task Control Block). This information helps when mapping and loading critical tasks to the SPM. Depending on the scheduled critical tasks' sizes, mapping critical tasks to the SPMs's physical locations involve partition resizing. The SPMs' partitions are maintained in a simple kernel data structure (partition table), shown in Figure 6.1. Mapping a critical task involves modifying the partition table and attaching the task to a partition. For example, when mapping a critical task, the size and the start address of the targeted partition in the partition table are altered to fit the mapped task. The task's ID is also attached to the targeted partition. After that, before staring/resuming the task, the RSMU is instructed to start translating the addresses of the mapped task. The partition table helps identify the SPMs' physical addresses for loading or unloading tasks.

Partition ID	task ptr	SPM address	Size	Status
1	0x0511568	0x1000	6.5 KB	Occupied
2	0x0517600	0x3000	4 KB	Free

Figure 6.1: An example depicting the structure of the partition table

For dynamic memory allocation, `malloc()` and `vpPORTmalloc()` functions are overridden. As a result, allocating dynamic data is now dependent on the current con-

text. For instance, the allocation can be either in the specialized OS-SPM heap in the case of critical tasks, or in the general system heap in the case of non-critical tasks. Sharing data is not preferable in critical real-time systems. However, the proposed system provides a persistent physically addressable SPM's heap to facilitate sharing data between critical tasks. For instance, the data allocated in OS-SPM heap is accessible from any task at any time without needing for address translation. the next chapter discusses how the off-line schedule is generated.

Chapter 7

Schedulability Analysis

This section discusses how to derive a predictable slotted schedule for critical tasks using our Real-time Scratchpad memory Management Unit (RSMU) architecture, according to the two-level scheduling mechanism described in Chapter 3. We assume that the system is comprised of a set Γ^c of N critical tasks $\{\tau_1^c, \ldots, \tau_N^c\}$. Each critical task τ_i^c is characterized by a period p_i^c, a worst-case computation time e_i^c, data size in scratchpad $data_i$ and code size in scratchpad $code_i$. The computation time e_i^c represents the worst-case execution time of the task, assuming that its data and code are already loaded into the scratchpad and discounting all OS overhead. Non-critical tasks run within a set of M non-critical partitions. We assume that the timing requirements of each partition τ_i^{NC} are expressed by a tuple $\{Q_i, p_i^{nc}\}$, meaning that the partition must receive at least Q_i units of execution time every p_i^{nc} time units.

Following the example in [47], we assume that all task and partition periods are multiples of a base period H; this is a common assumption in avionics systems. We then set the length of the minor cycle to be H. Each task is assigned one slot in each minor cycle. If a task has a period that is multiple of the minor cycle, it executes for a fraction of its execution time in the minor cycle. Next we construct a schedule with $N + M$ fixed slots in each minor cycle: each of M slots is assigned to partition τ_i^{nc} and has a length $\frac{Q_i}{p_i^{nc}/H}$; each of the remaining N slots is assigned to a critical task τ_i^c, which executes for $\frac{e_i^c}{p_i^c/H}$ in the minor cycle.

We next compute the size s_i^c of the slot assigned to critical task τ_i^c during each minor cycle. As detailed in Section 6.1, each task suffers OS overhead due to: 1) context switch overhead C_s; 2) DMA set-up time DMA_{setup}, which is paid three times; and 3) timer tick overhead t_s. Starting from the task execution $\frac{e_i^c}{p_i^c/H}$, the slot size must thus be enlarged to include all overheads:

$$s_i^c = \frac{e_i^c}{p_i^c/H} + C_s + 3DMA_{setup} + \lceil \frac{s_i^c}{1ms} \rceil t_s. \tag{7.1}$$

Note that with $1ms$ system-tick, $\lceil \frac{s_i^c}{1ms} \rceil$ represents the worst-case number of times that the timer can interrupt the slot execution; since this number depends on s_i^c, it is easy to see that the size of the slot can be computed by iterating over Equation 7.1, starting from the execution time $\frac{e_i^c}{p_i^c/H}$. Furthermore, note that the following equation represents an obvious necessary condition on the schedulability of the system:

$$\sum_{1 \le i \le N} s_i^c + \sum_{1 \le i \le M} \frac{Q_i}{p_i^c/H} \le H. \tag{7.2}$$

Table 7.1: An illustrative sample of critical tasks with $H = min(p^c) = 10$

	τ_1^c	τ_2^c	τ_3^c	τ_4^c
e^c (time units)	1	4	16	18
p^c (time units)	10	20	40	80
s^c (time units)	1.03602	2.03828	4.0428	2.28828

To conduct an illustrative example consider the set of arbitrary critical tasks in Table 7.1. There are four critical tasks with different harmonic periods. The length of the minor cycle is set to 10, which is the base period (H). Therefore, τ_1^c completes execution every minor cycle because its period is equal to the base period (H). On the other hand, the period of τ_2^c is two times the base period. As a result, τ_2^c completes execution every two minor cycles by running only for half the amount of its execution time in each minor cycle; τ_2^c gets one slot in each minor cycle. The slot size s^c includes overheads. Therefore, to ensure schedulability, the sum of all slots sizes has to be less than the length of the minor

cycle, which is 10 time units in this example. Figure 7.1 shows an example of how the four tasks can be scheduled using the minor cycle method. Each slot shows the fraction of execution time the task executes in the slot.

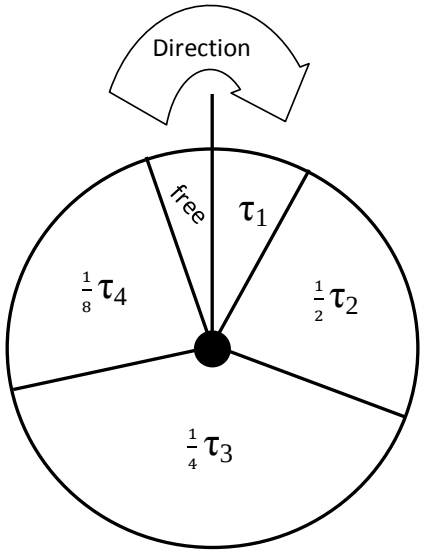

Figure 7.1: An illustration showing the slots of four critical tasks in a minor cycle schedule

To create the slotted schedule, we finally need to determine the order in which slots are executed within the minor cycle. Note that the order of non-critical partitions does not impact the schedulability of critical tasks in any way. Hence, we focus on computing the slot order for critical tasks; the schedulability of non-critical tasks can then be assessed based on the analysis in [47]. When deciding the slot order, we need to respect two constraints. First, the size of both the data and code scratchpad must be sufficient to execute the task in the current slot while performing the load/unload operations for the previous and next executed critical task, as detailed in Chapter 6. Second, the size of the slot must be sufficient to complete all DMA operations in time: the data of the previous critical task must be moved from the scratchpad to main memory, and both the code and data of the next critical task must be loaded from main memory.

To solve the slot assignment problem for critical tasks, we construct a simple satisfia-

bility problem. In the problem formulation in Equation 7.5, $DMA(b)$ represents the total time required to move b bytes from/to main memory using DMA, including OS overhead; we detail how to compute $DMA(b)$ in Section 8.1. Binary indicator variable $x_{i,j}$ indicates that task τ_j^c is executed in the i-th critical slot, with slots numbered from 0 to $N-1$. The code scratchpad size SPM_{C-SIZE}, and the data scratchpad size SPM_{D-SIZE} represent the space of the scratchpads dedicated for critical tasks.

$$\forall i, 0 \leq i \leq N-1 : \sum_{1 \leq j \leq N} x_{i,j} = 1 \tag{7.3}$$

$$\forall j, 1 \leq j \leq N : \sum_{0 \leq i \leq N-1} x_{i,j} = 1 \tag{7.4}$$

$$\begin{aligned} \forall i, 0 \leq i \leq N-1 : & \sum_{1 \leq j \leq N} x_{(i-1)\%N,j} \cdot DMA(data_j) + \\ & + \sum_{1 \leq j \leq N} x_{(i+1)\%N,j} \cdot \Big(DMA(data_j) + DMA(code_j) \Big) \leq \sum_{1 \leq j \leq N} x_{i,j} s_j \end{aligned} \tag{7.5}$$

$$\begin{aligned} \forall i, 0 \leq i \leq N-1 : & \sum_{1 \leq j \leq N} x_{i,j} \cdot data_j + \\ & + \sum_{1 \leq j \leq N} x_{(i+1)\%N,j} \cdot data_j \leq SPM_{D-SIZE} \end{aligned} \tag{7.6}$$

$$\begin{aligned} \forall i, 0 \leq i \leq N-1 : & \sum_{1 \leq j \leq N} x_{i,j} \cdot code_j + \\ & + \sum_{1 \leq j \leq N} x_{(i+1)\%N,j} \cdot code_j \leq SPM_{C-SIZE} \end{aligned} \tag{7.7}$$

Based on Equations 7.3 and 7.4, each slot has to be assigned to only one critical task and each critical task has to be assigned to only one slot. Equation 7.5 expresses the constraint on the DMA time. The right-hand side $\sum_{1 \leq j \leq N} x_{i,j} s_j$ computes the size of the i-th slot in the schedule, based on the task assigned to the slot. In the left-hand side, $\sum_{1 \leq j \leq N} x_{(i-1)\%N,j} \cdot DMA(data_j)$ represents the time required to unload from data scratchpad to main memory the data of the critical task executed in the previous slot, i.e., slot $(i-1)\%N$-slot, where % is the module operation. Note that in slot 0, we need to unload the data used by the task executed in slot $N-1$ in the previous minor cycle.

Finally, $\sum_{1 \leq j \leq N} x_{(i+1)\%N,j} \cdot \Big(DMA(data_j) + DMA(code_j) \Big)$ represents the time required to load from main memory to data/code scratchpad the data/code (respectively) of the critical task executed in the following slot $(i+1)\%N$; again, note that in slot $N-1$, we need to load the code and data used by the task executed in slot 0 in the next minor cycle.

Finally, Equations 7.6, 7.7 express constraints on the size of the data and code scratchpad, respectively. As detailed in Chapter 6, when a critical task is running, one portion of the scratchpad is used to contain the data/code of the running task, while the second portion allocates memory for the next executed critical task. Hence, we constrain the size of the data/code scratchpad to be at least equal to the data/code of the task running in slot i, plus the data/code of the task running in slot $(i+1)\%N$.

For instance consider Table 7.2, which depicts task parameters for an arbitrary set of tasks. Assume the sizes of each scratchpad is 16 size units and the total utilization of this set of tasks is less than one. The schedulability of the system depends on the task execution order. For example, the order of $\{\tau_1^c, \tau_2^c, \tau_3^c, \tau_4^c\}$ is not schedulable due to violation of DMA time constraint. The execution time of τ_4^c is not long enough to let the DMA finish unloading data of τ_3^c and loading code plus data of τ_1^c. On the other hand, a schedule such as $\{\tau_2^c, \tau_1^c, \tau_3^c, \tau_4^c\}$ has no timing violation but it violates the code scratchpad size constraint because the code sizes of $\tau_1^c + \tau_3^c$ exceed the limit of 16 size units. Actually, there are many different permutations for these four critical tasks. To get a task order that makes the system schedulable, we feed all system constraints to an SMT (Satisfiability Modulo Theories) solver, and the SMT solver finds a feasible schedule if it exists. A task order that can make the system schedulable is $\{\tau_2^c, \tau_1^c, \tau_4^c, \tau_3^c\}$. The next chapter discusses the experimental results and evaluates the system components: hardware, software and schedulability.

Table 7.2: An illustrative task's parameters of an arbitrary set of critical tasks

	τ_1^c	τ_2^c	τ_3^c	τ_4^c
e^c (time units)	11	15	9	10
code (size units)	12	3	6	4
data (size units)	6	6	4	1.5
$DMA(_{code})$ (time units)	8	2	4	3
$DMA(_{data})$ (time units)	4	4	3	2

Chapter 8

Platform Evaluation

In this chapter, hardware, software, and schedulability are evaluated in order to give a clear view of how the system is performing. The hardware and software platforms are first evaluated independent of any running task. Then, several benchmarks are run to evaluate both hardware and software combined. Finally, evaluation of the system schedulability concludes this chapter.

8.1 Hardware Evaluation

Table 8.1: System-Under-Test Parameters

	Without SPM	**With SPM**	**Difference %**
Area (LE)	9707	10212	5.2
Memory (bits)	36832	69600	88.9
Operating frequency MHz	100	98.2	1.8

The hardware platform is based on Altera's Cyclone II, a 90-nm process FPGA. The native platform has the fastest Nios-II/f soft-core processor, with separate instruction and data buses. The instruction cache is 16-KB and the D-cache is 16-KB. The cache is direct mapped with 32-byte cache line width, which is the maximum in this platform. Making

the cache line this wide increases the cache's efficiency, as with a smaller cache line the CPU makes more trips to main memory. There is a 64-MB off-chip SDRAM, running at 100 MHz, and a DMA core, which comes standard in most embedded platforms. The operating frequency of the native platform was 100 MHz. A 64-bit cycle counter running at the same speed as the CPU is used to measure the timing. This original platform is modified to form the competing platform. The modified platform has two RSMUs and two SPMs, as shown in Chapter 4 Figure 4.1. Each SPM is 16-KB in size. The RSMUs are connected to the Nios-II tightly-coupled memory interfaces. As shown in Table 8.1, the frequency has dropped to 98.2 MHz, the area increased by 5.2% in terms of logic elements, and the used memory bits increased by 88.9%. As explained in Chapter 3, the DMA does not interfere with cache while accessing main memory. As a result, the DMA core is able to transfer 16 KB in 45 us from main memory to scrachpad and vice versa. Figure 8.1 depicts the linear performance of the DMA. Both calculated and measured performance of the DMA are presented in the figure. However, the two lines are on top of each other due to the match in both calculated and measured performance. The equation for DMA timing is as follows:

$$DMA(b) = 280 + 0.2578 * b$$
Where $DMA(b)$ is in cycles, and b is in bytes.

It is important here to know that the above equation does not include the timing required to prepare the DMA transfer (see Table 8.2). To measure the DMA performance, first, the DMA set-up overhead is determined by reading the cycle counter before and after the set-up routine. After that, the DMA performance is measured by reading the cycle counter right before the DMA set-up routine and after the DMA finishes, by polling the status register of the DMA. Finally the overhead is subtracted from the measurements.

As mentioned earlier in Chapter 5, the hardware architecture is very simple. Thus, it has been easily integrated into the Altera's Nios-II platform without modifying any existing hardware component, just adding the new components. This confirms one of the main design goals, which is to involve minimal modifications to the original hardware architecture. Another interesting point about the proposed hardware architecture is the footprint

area it requires. On the other hand, the consumption of memory blocks is relatively high because the scratchpads are added in addition to cache. However, other solutions utilizing booth cache and scratchpad would require a similar area for memory blocks.

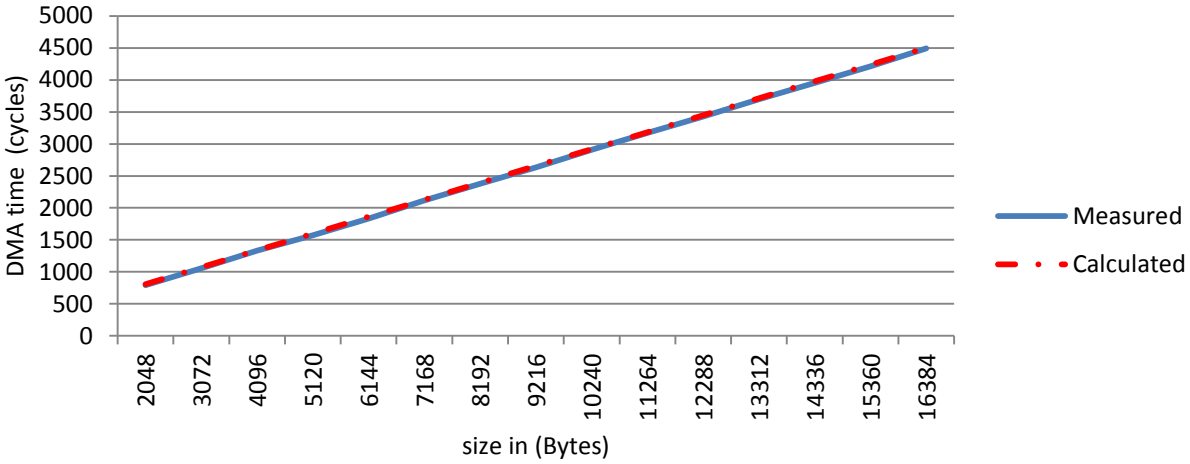

Figure 8.1: DMA Performance: the calculated performance is matching the measured one

8.2 Software Evaluation

The software platform used to evaluate the system consists of a real-time operating system (FreeRTOS), and some benchmarks. Table 8.2 shows the Software system parameters. The OS ticks every 1 ms. The system-tick overhead is measured by reading the cycle counter at the beginning of the timer Interrupt Service Routine (ISR) and reading it again before it returns. Then we added the time needed by executing assembly instructions before and after the ISR (interrupt response and recovery time). This time is only for system ticks that do not lead to a context switch. In the case of a context switch, the cycle counter is first read at the beginning of the timer ISR, then read again at the beginning of the new context considering the timer interrupt response time. In the same way, the context switch timing does not include any DMA set-up happening at the context switch. To take the timing of DMA set-up into account simply add the time required by the DMA set-up

routine to the normal context switch timing, eg., $450 + 900 = 1350$ cycles. It is noticeable from the table that the DMA set-up routine requires significant time. The reason for that is the non-optimized DMA driver provided by the Altera Hardware Abstraction Layer (HAL). The HAL provides Linux-like APIs in which all device drivers are treated in the same way. This abstraction adds many layers between the OS and the hardware. If we would write an optimised DMA driver, 100 - 150 cycles is expected for the DMA set-up routine. The RSMU driver is implemented as a macro that allows the kernel to control the RSMUs and maps a critical task in eight clock cycles. The software successfully managed the scratchpads at the OS level. As proposed in Chapter 6, the system achieved the second design goal by keeping the software model unchanged. As a result, several applications have been ported to this platform successfully without changing the applications' source code.

Table 8.2: Software Parameters

	Cycles
Context Switch	450
DMA prepare	900
System Tick	226

8.2.1 Benchmark Results

After the evaluation of hardware and software is done independently, the platform is evaluated by running some chosen benchmarks. A synthetic benchmark is used to evaluate the hardware performance, stressing the efficiency of the memory subsystem. It reads the first word (32-bits) of each cache line. This is done by iterating among all the 512 cache lines, for a 16-KB cache and 32-byte cache-line width, and reads only one word from each cache line. The proposed system not only provides a predictable execution time for critical tasks that run out of the scratchpad, but it also shows improvement in the performance as well. From the synthetic benchmark results shown in the last row of Table 8.3, the scratchpad performance is about 79% better than the cold-cache. The dirty-cache will be worse as the cache needs to write the evicted lines back to main memory before bringing the new

cache lines in. In the case of the scratchpad, there is no hot and cold situations, as critical tasks are preloaded into the scratchpad according to a schedule (refer to Chapter 3). As we know, a synthetic benchmark is not a real-life application. The synthetic benchmark is useful to show the hardware's capability, but does not necessarily predict how the system will work in real-life applications. Therefore, a set of real benchmarks has been selected to evaluate the platform and its applicability in the embedded real-time domain. The selection of the benchmarks is based on their relevant characteristics such as being memory intensive to stress the platform's memory subsystem. We chose three benchmarks from the well-known automotive EEMBC benchmark suite. The selected benchmarks, a2time (angle to time conversion), canrdr (response to remote CAN request), and rspeed (road speed calculation), have been studied by [12] and their characteristics are known. Another benchmark suite is DIS (Data Intensive System) benchmark [39]. Two benchmarks are selected from this suite, transitive and corner-turn. This selection of real benchmarks is aimed to represent several applications used in the embedded real-time domain.

Table 8.3: Benchmarks Results

Benchmark	code size(B)	Data size(B)	Elements	Iterations	Cold(cycles)	Hot(cycles)	SPM(cycles)	Difference(cycles)	Ratio %
a2time	3108	5420	1298	118	105162	100776	100497	4386	4.17
rspeed	1956	6864	1661	151	59420	55624	55688	3796	6.39
canrdr	2724	8030	1958	178	50720	47419	47280	3301	6.51
corner-turn	2032	8192	2040	1020	20681	16726	16728	3955	19.12
transitive	2080	3024	361	6859	105661	102995	102898	2666	2.52
synthetic	136	16384	512	512	12507	2592	2584	9915	79.28

Table 8.3 is generated by running each benchmark as a critical task, in which each task is limited either by the execution time of that task or by its size. Each benchmark will take more or less space based on number of iterations it performs [12]. Each benchmark, depending on its nature, performs a fixed number of ALU instructions per iteration. Similarly, it performs a fixed number of memory operation (read/write) per iteration. Each memory operation does not necessarily access a new memory location: it may access the same location more than once. The number of elements shows the number of memory locations that the benchmarks access after performing a certain number of iterations. These locations may not necessarily be adjacent or accessed one after another.

Each two critical tasks can share the SPM space as long as the sum of their required sizes in SPM is less than or equal to the SPM size; for example, one may occupy one

quarter and the other takes three quarters and so on. Tasks sizes have been taken into account according to the schedulability analysis presented in the previous chapter. Table 8.3 compares the cold-cache time versus hot-cache time. A small difference in the execution time between the hot cache and the SPM is caused by the dynamic branch predictor [1]. Although dirty cache is a common case in multitasking systems, comparing to the dirty cache is not considered: the comparison to a cold-cache is easier to analyze and gives better feedback on how the proposed system performs in a worst case scenario.

The result is read from the table as a cache stall ratio, which is the time the CPU stalls for the cold cache to fetch cache lines from the main memory divided by the execution time of a task on a cold cache. Higher ratio means that the execution time of a task is improved due to the improvement in the memory subsystem performance, which is the SPM in this case. The corner-turn benchmark, usually used in digital signal processing, scored the highest ratio, about 19%, due to its nature as it performs unit-stride and non-unit-stride access. Figure 8.2 depicts the cache stall ratio of this benchmark on our platform.

On the other hand, the transitive benchmark scored the lowest ratio as it performs regular unit-stride access N^3 times the number of elements. The other bench marks scored a fair ratio depending on each benchmark's nature. Note that, in this particular implementation of the proposed platform, the CPU and the main memory (SDRAM) are operating at the same frequency. A significant performance improvement is expected when the CPU operates at a higher frequency than the main memory. It is worth mentioning that the performance we describe here is not the only objective of our design. The main objective is to have predictable execution time of critical tasks with minimum impact on performance and cost. The proposed design is very cost effective as it has a tiny hardware implementation that is fast and independent of the number of tasks; it also offers seamless software porting.

[1] A dynamic history-based branch predictor can cause unpredictable execution time to a task. The execution time of a task is dependant on the current state of the branch predictor as mentioned earlier in Chapter 1

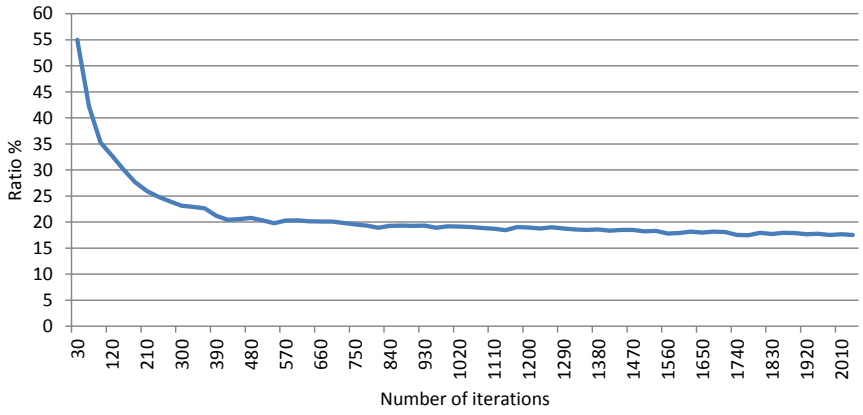

Figure 8.2: The cache stall ratio for the corner-turn benchmark

8.3 Schedulability Evaluation

The scheduling scheme proposed for the platform, explained in Chapter 3, is based on hiding the latency of the loading and unloading the scratchpads by overlapping the execution of critical tasks and DMA transfers. This novel technique has significantly improved the system schedulability. The theoretical background on how to compute and generate the slotted schedule is detailed in Chapter 7. This section evaluates the proposed scheduling scheme and compares it to a recent work [55]. As described in Chapter 7, the formulation of the schedulability problem is kept simple. The Z3 [37] SMT solver is used to solve the satisfiability problem and produce the correct working order of tasks within the minor cycle. The applications in Table 8.3, excluding the "synthetic" benchmark, are used to generate sets of random tasks. Given a system utilization, each application is randomly selected and assigned a random period from a predefined set of harmonic periods, {10 ms, 20 ms, 40 ms}. After that, the task's utilization is computed. At every iteration a new task is randomly generated. The generation stops when the sum of the individual tasks' utilization reaches the required system utilization. After that, the overhead is added and the slot sizes are computed. All the constraining equations, described in Chapter 7, are then applied to the generated set of tasks using the Z3 SMT solver. If there is a valid schedule, the Z3 returns sat(satisfiable) and gives the working assignment for the binary

indicators that represents tasks and slots. Otherwise, Z3 returns un-sat(un-satisfiable) indicating that there is no valid schedule.

It is obvious from the description in Chapter 7 that assigning slots within a minor cycle that is as long as the smallest period incurs more context switches, hence more overhead. However, we consider this method for two reasons. First, evaluating the platform requires putting it in a challenging situation to get better feedback of its performance, hence more fair when comparing to other scheduling schemes. Second, it is less complex than other schedulability formulation, and solving a complex schedulability problem is not the main focus of this thesis.

On our platform, we simulated randomly generated sets of tasks using two scheduling techniques. Carousel technique presented in [55], and our proposed technique. We set the same system parameters for both schemes, such as the context-switch and the DMA time. Carousel schedulability is verified by applying response-time analysis as described by the authors of Carousel. The main difference between the two schemes is that Carousel blocks tasks' execution (CPU) to complete scratchpad load and unload operations, as explained earlier in Chapter 2. Harmonic periods favour Carousel because with harmonic periods rate monotonic scheduling can schedule up to 100% utilization. Figure 8.3 depicts the significant improvement in schedulability over Carousel technique. Each point in the graph represents 100 task sets. Our technique successfully hid the latency of loading and unloading the scratchpad, which led to this result.

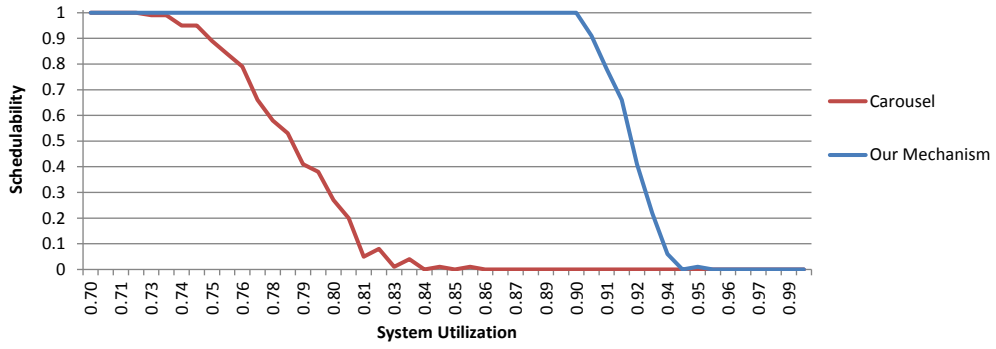

Figure 8.3: Our schedulability mechanism outperforms Carousel scheme

It is recommended to stick to the application design guidelines provided in Chapter 3 and 6 in order to get most of the platform. However, testing the platform with sets of tasks that violate the size constraint can provides further information about the system behaviour in such situations. This can be done either by increasing the tasks' working sets, or by reducing the size of the scratchpad. Evaluating the system schedulability when the size of the scratchpad is reduced is interesting, as there will be some tasks that cannot fit into the scratchpad together as they could before. In order to show the pure effect of the scratchpad size on schedulability, low utilization point is selected (10%). Figure 8.4 depicts the effect of changing the size of the scratchpad on the system schedulability. For instance, when the SPM size is reduced to 12KB, only 20% of the tasks sets are schedulable. In addition, it can be noticed from the figure that the schedulability is a quadratic function of the scratchpad size until it saturates. Therefore, it is important to consider the cost that can be expressed as trade-off between on-chip memory size and system utilization. Unlike in a cache where the size of cache has a similar effect on the average performance, the performance here is guaranteed once a set of tasks has a valid schedule. In Carousel, schedulability is not directly affected by the scratchpad size unless if a task cannot entirely fit into the scratchpad. However, Carousel is affected by the time required to swap blocks between the scratchpad and the main memory; thus, the sizes of tasks have a direct effect on Carousel schedulability. In addition, Carousel does not utilize cache besides scratchpad. Therefore, Carousel requires less on-chip memory than what we require.

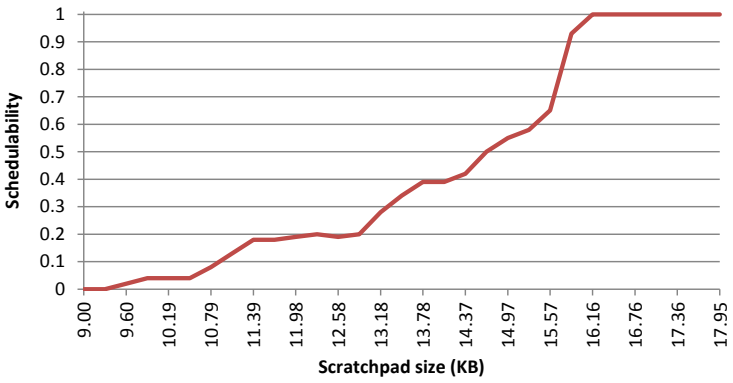

Figure 8.4: The effect of changing the size of the scratchpad on schedulability

Finally, when increasing the set of the harmonic periods that is assigned randomly to the participating tasks, there will be some tasks having large periods and small execution time. As a result, their slot's sizes will not be long enough for the DMA to load other tasks into the scratchpads. In addition, tasks with smaller slots incurs relatively more overhead compared to their slot's sizes. Figure 8.5 shows the effect of including larger periods into the set of the harmonic periods. A high utilization point (90%) is selected in order to show the system tolerance for overhead. The graph depicts that the system has a good tolerance to the overhead caused by scheduling many tasks with tiny slots. In the graph, the system goes below 10% schedulability only when the maximum period is 21 times double the base period, e.g. if the base period is 10 ms, the maximum in this case will be 10,485,760 ms.

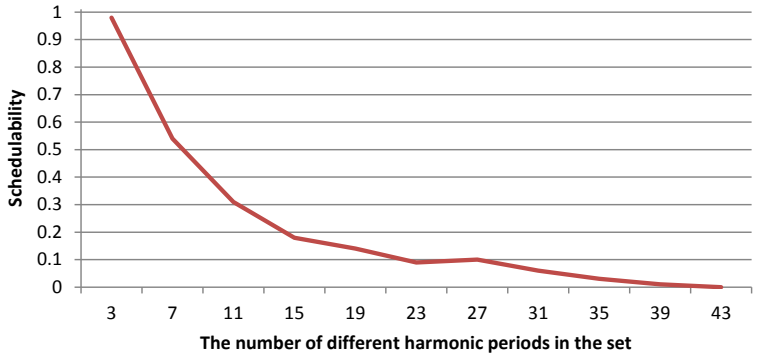

Figure 8.5: The effect of increasing the set of the harmonic periods on schedulability

Throughout this chapter, the functionality and performance of the proposed system were evaluated. The proposed system achieved the design objectives that aim to diminish the gap between the general-purpose architecture and real-time applications. In addition, the proposed system scored significant improvements in both execution performance and system schedulability. The next chapter concludes the thesis and highlights future work.

Chapter 9

Conclusions and Future work

The increasing complexity in general-purpose architecture has increased the gap between the low-cost general-purpose architecture and real-time systems requirements. Therefore, real-time applications are not taking complete advantage of new high-performance general-purpose processors. This thesis has introduced and evaluated a hardware-software technique in an effort to close the gap between real-time systems requirements and general-purpose architectures. Two main design principles were followed to achieve this goal. The first, involves minimal modifications in the original hardware architecture, thus making the new design more appealing to computer hardware manufacturers. The second keeps the software model unchanged so that applications are ported to the new platform easily. In this work, a solution to the problem of the inconsistent cache access time has been implemented on an FPGA platform. This work is distinguishable from previous works by looking into the problem more comprehensively and introduces a practical design with several features to solve different aspects of the problem.

The predictable cache-like SPM abstracted away the hardware details, such as address translation, from applications programmers. Since the SPM and the RSMU are managed by the OS, applications programmers benefit from the predictable execution offered by the architecture without being concerned about low-level details. Dynamic allocation of SPM's data is managed by the OS as well. Therefore, critical tasks are able to allocate and share SPM's dynamically allocated data.

Porting existing applications is one of the main objectives in this work. Applications are ported to this platform without modifications such as code annotation or post-compile processing. Writing applications to this platform uses the original platform's compiler and instruction-set architecture. The technique used to overlap the execution of critical tasks with loading/unloading the SPMs had a significant impact on improving system schedulability. The evaluation in Section 8.3 depicted the improvement in schedulability over the most recent work introduced in [55].

The architecture of the hardware is simple, yet efficient. The proposed hardware components are integrated into typical computer systems easily. The introduced RSMU has a small footprint area compared to other translation units. Unlike other scratchpad memory management units, the performance of the introduced RSMU is totally independent of the number of critical tasks it translates. In addition, unlike an MMU, the RSMU is predictable and fast. The RSMU has a very simple translating mechanism that translates addresses in constant time.

The solution introduced in this thesis can be further extended to support multi-core systems. The hardware architecture is already suitable for integration into multi-core systems. The coherency of the local SPMs can be handled either in hardware or software. Protecting shared data in multitasking/multi-core systems can also be investigated. Dedicating a special SPM partition that is monitored by hardware can help to protect shared data. In addition, the schedulability experiment provided in this thesis was used to evaluate the proposed system's architecture. The discussed schedule stressed the memory subsystem, and the system's ability to schedule more tasks. Therefore, an extended in-depth schedulability study can be conducted to obtain the optimal scheduling policy for this architecture.

A system without a cache at all can also be considered. The SPM will replace the cache and exploits the consumed area more efficiently. The SPM can be co-managed by hardware and software. This approach is expected to increase the system performance and reduce the power consumption. In addition, it will provide more flexibility in managing local on-chip memory, hence may increase schedulability. On the other side, the predictability issue incurred by the branch predictor in a deep pipeline can also be investigated using the same design principles provided in this thesis. The solution should involve minimal architectural

modifications and keep the software model unchanged. Solving the predictability issues in the major components of general-purpose architecture will lead to a more deterministic machine.

Finally, there is hope that the introduced design will be adopted by computer hardware manufacturers, so that the next generation of general-purpose processors will be able to support the real-time requirements natively. As a result, real-time applications can exploit the maximum processing power available in the new generation of processors.

References

[1] Cache Architecture, ECE 3055: Computer Architecture and Operating Systems. http://users.ece.gatech.edu/ dblough/3055/.

[2] DE2-70 development and education board. http://www.altera.com/education/univ /materials/boards/de2-70/unv-de2-70-board.html.

[3] ModelSim for Altera's devices. http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html.

[4] NiosII Embedded Processor. http://www.altera.com/devices/processor/nios2/ni2-index.html.

[5] Qsys : Altera System Integration Tool. http://www.altera.com/products/software /quartus-ii/subscription-edition/qsys/qts-qsys.html.

[6] Quartus-II design software. http://www.altera.com/products/software/quartus-ii/subscription-edition/design-entry-synthesis/qts-des-ent-syn.html.

[7] Scratchpad memory. http://en.wikipedia.org/wiki/Scratchpad_memory.

[8] SPEC Benchmarks. http://www.spec.org/benchmarks.html.

[9] Peter J Ashenden. *Digital Design An Embedded Systems Approach Using VHDL.* Morgan Kaufmann, 2007.

[10] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, November 2002.

[11] S Bak, E Betti, R Pellizzoni, M Caccamo, and L Sha. Real-time control of i/o cots peripherals for embedded systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 193–203. IEEE, 2009.

[12] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, August 2012.

[13] Rajeshwari Banakar, Stefan Steinke, BS Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the ...*, page 73, New York, New York, USA, May 2002. ACM Press.

[14] S Bandyopadhyay, F Huining, H Patel, and E Lee. A scratchpad memory allocation scheme for dataflow models. Technical report, Citeseer, 2008.

[15] D M Beazley, B D Ward, and I R Cooke. The inside story on shared libraries and dynamic loading. *Computing in Science & Engineering*, 3(5):90–97, 2001.

[16] D R Chase, M Wegman, and F K Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN Notices*, volume 25, pages 296–310. ACM, 1990.

[17] Y Chu. Cache and branch prediction improvements for advanced computer architecture. 2001.

[18] David Déharbe, Stephenson Galvão, and Anamaria Moreira. Formalizing FreeRTOS: First Steps. In Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods Foundations and Applications*, volume 5902 of *Lecture Notes in Computer Science*, pages 101–117. Springer Verlag, 2009.

[19] J.-F. Deverge and I Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 179–190, July 2007.

[20] Kaivalya M. Dixit. Overview of the SPEC Benchmarks. *The Benchmark Handbook*, pages 489–521, 1993.

[21] S A Edwards and E A Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th annual Design Automation Conference*, pages 264–265. ACM, 2007.

[22] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, pages 321–330, New York, NY, USA, 2006. ACM.

[23] 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 *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 238–243, New York, NY, USA, 2004. ACM.

[24] B R Gaeke, P Husbands, X S Li, L Oliker, K A Yelick, and R Biswas. Memory-intensive benchmarks: IRAM vs. cache-based machines, 2002.

[25] B Gough and R Stallman. An Introduction to GCC. *Network Theory, Ltd*, 2004.

[26] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The {M{ä}lardalen} {WCET} Benchmarks – Past, Present and Future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[27] J O Hamplen, T S Hall, and M D Furman. *Rapid Prototyping Of Digital Systems*. Springer, 2010.

[28] Wei Hu Wei Hu, Tianzhou Chen Tianzhou Chen, Qingsong Shi Qingsong Shi, and Feng Sha Feng Sha. Efficient utilization of scratch-pad memory for embedded systems, 2009.

[29] Jan M. Rabaey, Anantha Chandrakasan and Borivoje Nikolic. *Digital Integrated Circuits*. Prentice-Hall, 2002.

[30] M Kandemir, J Ramanujam, M J Irwin, N Vijaykrishnan, I Kadayif, and A Parikh. Dynamic management of scratch-pad memory space, 2001.

[31] EA Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. *University of California, Berkeley, Tech. Rep. UCB/ . . .*, 2007.

[32] J Lee, J Park, and S Hong. Memory Footprint Reduction with Quasi-Static Shared Libraries in MMU-less Embedded Systems. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 24–36. IEEE, 2006.

[33] M Lewandowski, M J Stanovich, T P Baker, K Gopalan, and A I Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 57–68. IEEE, 2007.

[34] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D Patel, Stephen A Edwards, and Edward A Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM.

[35] P Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. 2010.

[36] Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient dynamic heap allocation of scratch-pad memory. *Proceedings of the 7th international symposium on Memory management ISMM 08*, page 31, 2008.

[37] L De Moura and N Bjø rner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[38] Tiago Rogerio Muck and Antonio Augusto Frohlich. Run-time scratch-pad memory management for embedded systems. In *IECON 2011 - 37th Annual Conference on*

IEEE Industrial Electronics Society, pages 2833–2838. Software/Hardware Integration Lab, Federal University of Santa Catarina, Florianópolis, Brazil, IEEE, 2011.

[39] J Musmanno. Data Intensive Systems (DIS) Benchmark Performance Summary. 2003.

[40] O Ozturk, M Kandemir, and I Kolcu. Shared scratch-pad memory space management, 2006.

[41] PR Panda, ND Dutt, and A Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *... of the 1997 European conference on ...*, 1997.

[42] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A Novel Technique to Use Scratch-pad Memory for Stack Management. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[43] 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 and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

[44] Phillip A. Laplante. *Real-Time Systems Design and Analysis*. IEEE Press & Wiley-Interscience, third edit edition, 2004.

[45] Aayush Prakash and HD Patel. An instruction scratchpad memory allocation for the precision timed architecture. In *Design, Automation & Test in Europe ...*, 2012.

[46] I Puaut 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, April 2007.

[47] L Sha. Real-time virtual machines for avionics software porting and development. *Real-Time and Embedded Computing Systems and Applications*, 2004.

[48] V Suhendra and T Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303. ACM, 2008.

[49] H Takase, H Tomiyama, and H Takada. 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, March 2010.

[50] Hideki Takase Hideki Takase, H Tomiyama, and H Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems, 2010.

[51] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 276–286, New York, NY, USA, 2003. ACM.

[52] L Wehmeyer and P Marwedel. Influence of onchip scratchpad memories on wcet prediction. 2004.

[53] J Whitham and N Audsley. Implementing time-predictable load and store operations. In *Proc. EMSOFT*, pages 265–274, 2009.

[54] J Whitham, RI Davis, N Audsley, S Altmeyer, and C Maiza. Investigation of Scratch-pad Memory for Preemptive Multitasking. *jwhitham.org*.

[55] 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 and Embedded Technology and Applications Symposium*, pages 3–12. IEEE, April 2012.