Universidade do Minho
Escola de Engenharia

Tiago Agostinho da Silva Gomes

Multithreading RTOS Processor Design

outubro de 2015

Universidade do Minho
Escola de Engenharia

Tiago Agostinho da Silva Gomes

Multithreading RTOS Processor Design

Tese de Doutoramento
Plano Doutoral em Engenharia Eletrónica e de Computadores

Trabalho efectuado sob a orientação do
Professor Doutor João Luis Marques Pereira Monteiro
Professor Doutor José Araújo Mendes

outubro de 2015

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, _____

Full name: Tiago Agostinho da Silva Gomes

Signature: _____

Acknowledgments

I want to begin my acknowledgments with my mentor of embedded systems, Dr. Adriano Tavares, to whom I owe my deepest thanks for this interesting journey.

I should like to express my gratitude to all the staff of the Embedded Systems Research Group (ESRG) at the University of Minho, particularly my advisors, Prof. Dr. João Monteiro and Dr. José Mendes. Special thanks to Dr. Mongkol Ekpanyapong for his helpful advice and support during the year spent at Asian Institute of Technology and to Prof. Dr. Sergio Montenegro for the valuable time spent at Julius Maximilians University of Würzburg.

I want to give a huge shout-out to all fellow students at ESRG, particularly to Filipe Salgado, Tiago Gomes, Sandro Pinto and Dom Nuno Cardoso not only for the valuable feedback but also for the many entertaining moments we enjoyed. A sign of appreciation to Paulo Garcia for his particular help and advice during my PhD which I will always remember.

Without the support of all my family this thesis could not have been completed. My thanks go to them, specially to, my father, brother and girlfriend for all the love, comfort and guidance during this journey.

Thank you all.

This thesis was supported by a PhD scholarship from Fundação para a Ciência e Tecnologia, SFRH/BD/81682/2011.

Tiago Gomes

Guimarães, October 7th, 2015.

This One Is For My Mother

Abstract

Multithreading RTOS Processor Design

Embedded systems are increasingly more complex computational systems, often heterogeneous and also with real-time requirements, supporting sophisticated and demanding software tasks. To deal with this complexity, real-time constraints and increasingly shorter time-to-market, Real-Time Operating Systems (RTOSes) are used to provide an abstraction layer on top of the hardware, providing several mechanisms to simplify and coordinate the system's behavior. These mechanisms induce latencies and large CPU time consumption, which consequently increase overhead and contribute to the system's performance degradation.

This thesis proposes to study and implement tools and methodologies that, considering the application's requirements and the programmable hardware's restrictions, alleviate this overhead through hardware acceleration, by: (1) incorporating RTOSes' primitives and structures in the CPU, whenever possible; (2) providing customization capabilities to enable design space exploration (DSE) while migrating such primitives and structures, as well as, application specific functionalities to gateware and (3) offering an agnostic solution to promote portability among different RTOSes.

The implemented solution contributes to the real-time embedded systems field by presenting novel micro-architectural features to cope with real-time requirements. This thesis presents a co-designed software/hardware multithreaded architecture that promotes configurability, determinism, performance, energy-efficiency (to some extent) and portability from the outset. Experimental results demonstrate that the implemented solution surpasses the state of the art, by providing a complete and agnostic solution which is independent of any specific RTOS, with only a small cost on hardware area. Appropriate benchmarking shows the benefits of the implemented solution on tests targeting FreeRTOS and μCOSII.

Resumo

Desenho de Processador Multitarefa para Sistemas Operativos de Tempo Real

Os sistemas embebidos são sistemas computacionais que se têm tornado cada vez mais complexos, muitas vezes heterogéneos e com requisitos de tempo real, suportando tarefas sofisticadas e exigentes. Para lidar com toda esta complexidade e simultaneamente não comprometer um time-to-market cada vez mais curto, recorre-se a sistemas operativos de tempo real (RTOSes). Estes não só fornecem uma camada de abstração sobre o hardware como também disponibilizam vários mecanismos para simplificar e coordenar o comportamento do sistema. No entanto, estes mecanismos induzem latências e sobrecarga no processamento, o que se traduz numa degradação do desempenho do sistema.

Nesta tese propõe-se estudar e implementar ferramentas e metodologias que, considerando os requisitos da aplicação e as restrição do hardware programável, aliviam esta sobrecarga através de aceleração por hardware, possibilitando: (1) incorporação de primitivas e estruturas de RTOS no processador; (2) fornecimento de recursos de customização que permitam a exploração do espaço de projeto durante a migração de tais primitivas e estruturas, bem como funcionalidades específicas da aplicação; e (3) uma solução agnóstica para promover a portabilidade entre diferentes RTOSes.

Nesta tese é apresentada uma arquitetura híbrida de hardware-software que promove configurabilidade, determinismo, desempenho, eficiência energética e portabilidade. A contribuição para a área de sistemas embebidos de tempo real revê-se nas novas funcionalidades micro-arquiteturais para lidar com os requisitos de tempo real. Os resultados experimentais demonstram que esta solução supera o estado da arte ao fornecer uma solução completa e agnóstica. *Benchmarks* adequados mostram os benefícios da solução implementada usando FreeRTOS e μCOSII.

Contents

List of Figures

List of Tables

Acronyms

ADL	**A**rchitecture **D**escription **L**anguage
AIC	**A**dvanced **I**nterrupt **C**ontroller
ALU	**A**rithmetic **L**ogic **U**nit
API	**A**pplication **P**rogramming **I**nterface
ARM	**A**dvanced **R**ISC **M**achine
ASIP	**A**pplication **S**pecific **I**nstruction-Set **P**rocessor
BMT	**B**locking **M**ulti**T**hreading
CCU	**C**ustom **C**omputational **U**nits
CPU	**C**entral **P**rocessing **U**nit
DDR	**D**ouble **D**ata **R**ate
DDRF	**D**esigner-**D**efined **R**egister-**F**ile
DMA	**D**irect **M**emory **A**ccess
DoE	**D**esign **o**f **E**xperiment
DSE	**D**esign **S**pace **E**xploration
EDA	**E**lectronic **D**esign **A**utomation
DSP	**D**igital **S**ignal **P**rocessor
EDF	**E**arliest **D**eadline **F**irst
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPOS	**G**eneral **P**urpose **O**perating **S**ystems
hAPI	**h**ardware-based **A**pplication **P**rogramming **I**nterface
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh-**L**evel **S**ynthesis
HW	**H**ard**W**are
HW-MT	**H**ard**W**are Multi**T**hreading
IP	**I**ntellectual **P**roperty
IMT	**I**nterleaved Mul**T**ithreading
IRQ	**I**nterrupt **R**e**Q**uest
ISA	**I**nstruction **S**et **A**rchitecture

ISR	**I**nterrupt **S**ervice **R**outine
ITP	**I**nterrupt-**T**hread **P**riority
LC	**L**oosely-**C**oupled
LDM	**L**oa**d** **M**ultiple
LUT	**L**ook **U**p **T**able
MCR	**M**ove to **C**oprocessor from ARM **R**egister
MMU	**M**emory **M**anagement **U**nit
MRC	**M**ove to ARM **R**egister from **C**oprocessor
MT	**M**ulti**T**hreading
NoC	**N**etwork-**o**n-**C**hip
OS	**O**perating **S**ystem
PC	**P**rogram **C**ounter
PIT	**P**eriodic **I**nterval **T**imer
PM	**P**erformance **M**onitor
RAM	**R**andom **A**ccess **M**emory
RISC	**R**educed **I**nstruction **S**et **C**omputer
ROM	**R**ead-**O**nly **M**emory
RTL	**R**egister-**T**ransfer **L**evel
RTOS	**R**eal-**T**ime **O**perating **S**ystem
RTP	**R**unning **T**hread **P**riority
RT-SHADOWS	**R**eal-**T**ime **S**ystem **H**ardware for **A**gnostic and **D**eterministic **O**Ses **W**ithin **S**oftcore
RSM	**R**esponse **S**urface **M**odeling
SoC	**S**ystem **o**n **C**hip
SMT	**S**imultaneous **M**ulti**T**hreading
SVC	**S**upervisor
STM	**St**ore **M**ultiple
SW	**S**oft**w**are
SWI	**S**oft**W**are **I**nterrupt
SYS	**S**ystem
USART	**U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter
USR	**U**ser
TAIC	**T**ask-**A**ware **I**nterrupt **C**ontroller
TC	**T**ightly-**C**oupled
TCB	**T**ask **C**ontrol **B**lock
VLIW	**V**ery **L**ong **I**nstruction **W**ord
WCET	**W**orst-**C**ase **E**xecution **T**ime

Chapter 1

Introduction

Advances in integrated circuits' technology have allowed the integration of more and more tasks and features in embedded systems, leading to a more complex development process. Instead of creating from scratch, something which has become nearly impossible, embedded system designers follow a platform- and/or chip generator-based methodologies (Hameed et al., 2010; Shacham et al., 2010, 2012; Solomatnikov et al., 2009; Shacham, 2011), where Intellectual Property (IP) cores are used as building blocks from which the final system is implemented. Moreover, new architectures with novel architectural features must be developed taking into account configurability and parametrization to promote re-use and application-specific solutions.

Current applications' demands pose several challenges to modern embedded systems which now face a performance barrier that hinders the execution of real-time operations. These real-time execution requirements lead to the emergence of new architectures such as multithreading (MT) processors and multicore chips. Although both are able to exploit the concurrency of workload and thus improve performance throughput and real-time responsiveness, designers need to study what fits a specific real-time application better, even though they are complementary solutions since designers can opt for using multithreading, multicore or a combination of both.

A way to alleviate the complexity of current real-time embedded systems development process is to incorporate Real-Time Operating Systems (RTOSes) which provide several different mechanisms, such as multithreading, semaphores, timers and interrupts handling in order to simplify and coordinate the systems' behavior.

With the increase in application demands (e.g., in terms of functionalities and real-time) it is hard to find an embedded system without an Operating System (OS) included in its software (SW) stack.

A well structured RTOS considers the application requirements to generate only specific functionalities used by the application, avoiding waste of resources. For instance, the final RTOS image does not need to include mutexes or queues support if the application does not use them. Nevertheless, RTOSes offer a myriad of mechanisms to abstract the hardware (HW) layer from the application layer. For some classes of real-time systems, however, the overhead caused by these RTOS mechanisms cannot be ignored. For instance, overheads related to scheduling and context switching eat up a lot of Central Processing Unit (CPU) time, while the overhead related to interrupt management contributes to interrupt latency, limiting the system's response time. A way to lessen these overheads has been to implement RTOSes with hardware acceleration, since the scheduler and other mechanisms are excellent candidates for hardware migration.

An important aspect to take into consideration in the development of an embedded system is which processor should be used to run the RTOS, in terms of the support which it is capable to provide. General-purpose processors may be a good solution thanks to the great flexibility provided. However, certain applications require specific processing capabilities which most general-purpose processors may not possess. To tackle these requirements, Application-Specific Instruction-set Processors (ASIPs) have emerged as processors completely dedicated to a target application.

To this purpose, generic Field Programmable Gate Arrays (FPGAs) have been used as a prototyping and implementation platform in embedded systems, where the entire embedded system may be implemented, to test and validate its functionalities, and then deployed. This has motivated embedded system designers to use FPGAs intensively not only as a development platform but also as final product, minimizing the development cost and time-to-market. With the widespread emergence of the new low-cost, high-density and high-performance FPGAs, the use of FPGA-based platforms becomes even more realistic. FPGAs offer the possibility of creating customizable hardware with high degree of parallel processing capabilities, allowing a development methodology that follows a hardware-software co-design philosophy. Therefore, FPGAs are a good platform candidate to develop co-designed hybrid embedded systems by easily merging SW and HW domains.

ASIPs developed over a FPGA platform appear as a great opportunity to implement parameterisable softcores with specialized processing units to efficiently execute, in terms of performance, energy consumed or resources usage, a particular range of applications. For instance, a softcore processor can be extended with new processing units to be used in many applications, such as control systems applications, signal processing applications, applications with complex numbers processing, etc. On the process of designing an ASIP, the system designer must find a processor architecture that meets the target application. ASIPs implemented in FPGAs offer a great level of configurability, since parameters such as the bus width, register-file width and type, number of pipeline stages, size and type of caches, number of threads or number of instructions can be easily modified. Therefore, measuring the design quality of different architecture configurations in terms of area, performance and power consumption is essential to find the best fit solution. In applications with rigid constraints meeting these metrics must be ensured. However, due to the huge number of parameters, choosing the perfect ASIP configuration to execute a particular application is not a trivial task.

Design Space Exploration (DSE) helps system's designer in finding an optimum solution. Using customizable/configurable tools/platforms/artifacts can reduce the design space, enabling DSE under time-to-market constraints, while maintaining sufficient solution variety to ensure the development of an adequate implementation. DSE selects an optimal configuration to a certain application fulfilling one or more design constraints, taking in consideration several aspects related with the target application. Hence, new ASIP architectures must be configurable, offering parametrization capabilities to allow design exploration.

The development of real-time embedded systems towards ASIP architectures with specific OS hardware support, can take advantages of the configurability provided by FPGA platforms to provide customization capabilities, allowing DSE to be performed to create custom tailored solutions suitable for a particular application.

The remainder of this chapter is organized as follows: Section 1.1 describes the scope of this thesis; Section 1.2 presents the research questions and the methodology proposed to answer those questions; Section 1.3 shows the state of art in different hardware acceleration solutions; Section 1.4 describes the solution envisioned by us; Section 1.5 presents the structure of this thesis.

1.1 Scope

Hardware multithreading (HW-MT) is a processor-level optimization to improve area and energy efficiency ideal for applications with some degree of concurrency where different threads require high coordination and inter-communication to perform a specific task. A multithreaded processor ensures multiple thread contexts within the same core by replicating task-specific registers and connecting a hardware scheduler into the System-on-Chip (SoC). Therefore, two distinct thread contexts can be switched in hardware without saving and restoring their state in software. This makes multithreading architectures an optimal platform for real-time applications where external interrupt events can be serviced with zero-latency. For low-end systems, where area and energy efficiency are two important metrics, the multicore approach may fall out of the scope. Multicore architectures require an interconnection between the cores either using a Network-on-Chip (NoC) or a shared memory which needs to be fairly large and with high bandwidth. Also, in idle state, multicore architectures have an amount of leakage current proportional to the number of cores in the system (Kissell, 2007).

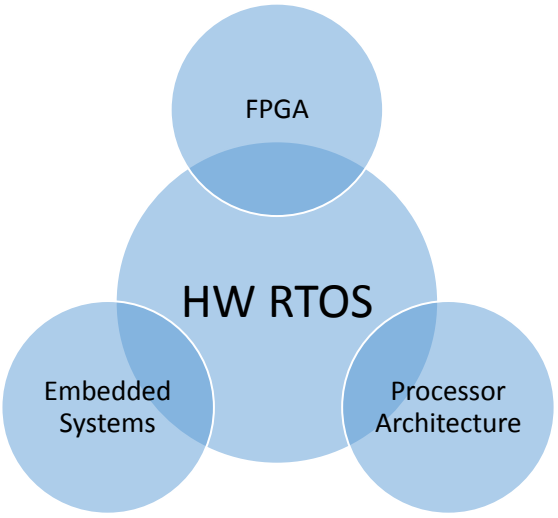


Figure 1.1: Scope.

Embedded system designers take advantage of RTOSes to develop multithreaded applications as they provide several Application Programming Interfaces (APIs) which ease and accelerate the development process and thus decrease the time-to-market. However, these APIs induce latencies and consume CPU time, which consequently increases overhead and contributes to the system's performance degradation. The emergence of FPGAs allows designers to exploit hardware-software

co-designed applications in order to reduce the overhead of SW-only implementations and thus improve the system performance of embedded systems (Figure 1.1). A multitude of approaches try to reduce the RTOS overhead by offloading some of the RTOS features into the hardware layer. Such approaches present some drawbacks, as they are restricted to a specific RTOS which requires a huge porting effort and limits the re-utilization of legacy software. Additionally, some of them targeted purpose-built cores which are usually proprietary. The migration of RTOS APIs to hardware is usually ensured through a coprocessor approach, where, for instance, the thread scheduler is migrated to hardware performing thread scheduling for real-time systems. There are two different approaches: (i) on the first one the thread scheduler can be connected to the system in a loosely-coupled (LC) fashion, i.e., interfaced through a bus to the core; this solution is adopted in systems using hardcore processors where no modifications are allowed within the core; (ii) the second approach is tightly-coupling (TC) the thread scheduler into the processor datapath. Thus, the processor can also be extended with hardware multithreading support and perform the APIs and context-switch operations in shorter time.

Nevertheless, hardware multithreading and RTOS hardware support have not been synergistically applied to take into consideration portability to several different RTOSes solutions. Focusing on improving performance and determinism while maintaining portability simultaneously will capture the attention of the industrial community since these solutions will cover a wide spectrum of RTOSes and will be RTOS-independent, not demanding software developers to have in-depth knowledge of the RTOS architecture.

The following are important identified requirements for hardware-based RTOS solutions: (1) low cost and low power consumption; (2) improved real-time processing; (3) low-latency execution; (4) portability to ensure legacy software re-use.

The scope of this thesis focuses on low-end single-core multithreaded architectures exploiting FPGA platforms to provide configurable RTOS hardware support while maintaining an acceptable level of portability. This thesis focuses on the arrangement between the following three layers and their dependencies: (i) application layer; (ii) RTOS layer and (iii) hardware layer. All research will take into account the impact on the following aspects such as performance, determinism, hardware costs and memory footprint. Furthermore, this thesis targets low- to medium-grade embedded systems not requiring virtualization technologies neither multicore architectures.

1.2 Research Questions and Methodology

We predict that getting the benefits of hardware offloading without sacrificing portability to different solutions requires an agnostic approach to the software to hardware migration. For this reason, this thesis tries to answer to the following questions:

1. Which (micro)architectural hardware features can be explored to increase performance without sacrificing determinism and energy-efficiency?
2. How can these features be exploited to full potential without sacrificing software flexibility?
3. How can portability be guaranteed across software stacks (RTOSes) while supporting common functionalities acceleration?

applying the following methodology:

1. Identify which RTOS sub-systems contribute to performance, power and determinism degradation;
2. Analyze the possibility of migration to hardware of each sub-system according to the application demands and analyze the impact in performance, determinism, portability, area, power and scalability;
3. Propose new micro-architectural features and compare them to the state of art.

1.3 State of Art

Demanding applications require complex hardware. Embedded systems rely on RTOSes services to abstract this complex hardware from the software layer. The counterpart of this abstraction is the processing overhead on the CPU. Different approaches have been proposed in the literature to address this problem. The following sections present several research works that use different methods to reduce the impact of RTOSes, such as using a coprocessor approach, implementing RTOS services in hardware, using hardware-based RTOSes or applying instruction-set architecture (ISA) customization.

1.3.1 Operating System Hardware Acceleration

The attempt to make an embedded system capable of providing hardware support to a RTOS is not recent, as there is a large number of projects/publications focusing on this subject, typically following a component based model (van Ommering et al., 2000; Levis et al., 2004). Back in 1991, efforts were being made to integrate operating systems' functionalities in hardware, providing microprocessors with the ability to offer support to the RTOSes' primitives (Lindh, 1991). The emergence of high capacity reconfigurable FPGAs at a lower cost, renewed the interest in this field in recent years (Sindhwani et al., 2004; Panneerselvam and Swift, 2012).

The support to RTOS primitives allows increasing performance in an embedded system, since actions such as context switching become atomic operations, executed with zero latency. This support may be provided through a coprocessor approach, such as the one presented in (Cooling and Tweedale, 1997), which performs threads scheduling for hard real-time systems. This approach leads to execution latencies due to the need to access the bus and, as such, will never provide a null overhead context switching. Another approach is the integration of RTOS primitives in the processor's datapath (tightly-coupled), a more efficient solution since it provides null-overhead context switching, requiring, however, some degree of flexibility at the processor design level (e.g., as provided by Xtensa and ARC architectures (Gonzalez, 2000; ARC, 2012)).

The wide availability of softcore processors, both open-source or intellectual property (Consortium, 2012), makes this approach much more attractive. Wijesinghe (2008) presented a processor that supports RTOS primitives. The processor used was Microblaze, Xilinx Corp. IP, adapted and expanded to provide the required functionalities. Bahri et al. (2012) offloaded the scheduling and communication layers from software to hardware achieving significant acceleration. Varela et al. (2012) developed a loosely-coupled coprocessor scheduler for NIOSII allowing different scheduling algorithm to be used.

The usage of IP processors shortens the time-to-market and provides flexibility, due to some extension capabilities offered by manufacturers. Their ISAs may be extended in order to provide specific instructions to the application's domain (Labrecque and Steffan, 2007). However, flexibility is limited as these manufacturers allow only the expansion of some ISAs' aspects, e.g., changes in some datapath's functional units such as the Arithmetic Logic Unit (ALU).

With processors developed from scratch, the flexibility is higher and it's possible to develop a specific solution, optimized for the application, although the time-to-market is substantially higher. In (Dimond et al., 2005), a framework identifies sets of instructions frequently used in a given application and adds to a standard processor's datapath several Custom Computational Units (CCUs), capable of performing these operations. Zaykov et al. (2014) proposed a hardware task-status manager CCU to reduce the Worst-Case Execution Time (WCET) of the RTOS. Bloom et al. (2012) introduced hardware data structures to reduce latency and jitter of data structure operations improving predictability of memory accesses. Kumar et al. (2012) and Tang and Bergmann (2015) presented coprocessor hardware schedulers connected to the processor bus to improve processing overhead and timing predictability of the scheduler. Balfour et al. (2008) implemented a custom processor architecture to speed up the performance of applications. Dittmann (2006) presented a processor dedicated to an application domain; its ISA was implemented based on an analysis of the domain's requirements. Iturbe et al. (2010) and Agron et al. (2006) presented a reliable reconfigurable real-time operating system (R3TOS) and a multithreaded RTOS kernel for hybrid FPGA/CPU systems (Hthreads), respectively, where some run-time system components such as scheduler and thread manager are migrated to hardware.

1.3.2 Application-Specific Instruction-Set Processors

Due to the capabilities of current hardware description languages as well as the widespread use of FPGAs as prototyping and deployment platform, designing application-specific processors is becoming more feasible (Zhang et al., 2008; Grad and Plessl, 2011). Thus, the development of application-specific processors is an active field of research in the area of embedded systems (Nsame et al., 2014; Marzouqi et al., 2015). A large majority of applications, due to their complexity and tight constraints, requires specialized processors to execute in an efficiently way a particular task. ASIPs are application-specific processors which are custom tailored to execute a certain task, or a set of applications, and are often used to tackle performance issues when general-purpose processors cannot meet the application demands (Gour and Jain, 2011; Nery et al., 2011) while keeping a fair degree of flexibility when compared with strict dedicated solutions (i.e., without software programmability) (Hohenauer and Leupers, 2009). In order to achieve the performance required by a specific application, while meeting other design aspects such

as area or energy consumption, an ASIP must be developed to achieve the required acceleration or parallelism over a general-purpose processor (Nery et al., 2011).

Since most general-purpose processors lack the requirements for many embedded systems, such as determinism or real-time predictability, ASIPs have emerged as processors specifically developed to a target application (Oliveira et al., 2011). Industrial available ASIPs such as ARC (ARC, 2012), LEON3 (Gaisler, 2012) and Xtensa (Gonzalez, 2000) presented large improvements over traditional processors for several applications (Nery et al., 2011). All are highly configurable allowing the implementation of a full range of processors optimized for a specific set of applications. While some ASIPs are completely dedicated to perform a specific task others offer more flexibility. The latter ones have a configurable instruction set where a minimum ISA is available and can be extended to design custom instructions using available configurable logic (e.g., Altera's Nios II (Altera, 2011a), Xilinx's MicroBlaze (Xilinx, 2008)). The advent of high-density FPGAs and the flexibility offered by these softcores open the possibility of customization capabilities since the use of these custom instructions allows the integration of complex functions in the processor's datapath (Pothineni et al., 2010).

To meet the computational demand of modern applications, Vassiliadis et al. (Vassiliadis et al., 2009) presented a systematic approach to extend an embedded processor to support several number and types of CCUs connected in TC or LC coupled fashions. In (Bordoloi et al., 2009), several studies have demonstrated the tradeoffs of using different CCU configurations in terms of area and performance metrics. A CCU is a unit that implements the equivalent of several software instructions in hardware to allow faster execution of those instructions and therefore it provides an increase of performance. There are two types of CCUs, the ones that are TC to the processor, i.e., the CCUs are integrated in the processor core and treated as internal units of the processor's datapath that interact directly with other functional units. The LC CCUs are the ones connected to the processor as an external coprocessor; therefore, they are outside of the processor core. The former allow a deterministic communication of the processor with the CCU. The latter approach leads to overhead due to bus arbitration but it allows a variable number of CCUs to be connected. Several studies (Huynh et al., 2010; shuai Lu et al., 2008; Atasu et al., 2012) have been presented where different methodologies and approaches were demonstrated in order to efficiently migrate instructions from an application code to custom instructions implemented in hardware.

With the advent of Systems-on-a-Chip (SoCs), partitioning functionalities between hardware and software, has become more and more feasible in commercial RTOS for a wide range of embedded processors. The growing complexity of today's systems have led the researchers to develop tools and programming environments that allow system designers to use the full potential of new reconfigurable chips. The existing computational models for hybrid systems created the need to adopt abstraction computational models that offer a proper abstraction level of the CPU/FPGA platform distinctions to the programmers.

Andrews et al. (2008) created and developed the architecture, models and tools, to design systems for real-time applications using hybrid CPU and FPGA systems. The system includes operating system and middleware layer abstractions that enable all platforms components to be abstracted into a unified multiprocessor architecture platform. A programming model (Hthreads), with unified APIs, was implemented which is fully compliant with the standard pthreads APIs. Thus, the architecture allows the execution of concurrent threads implemented both in software and hardware within a hybrid computer processor unit.

Mooney and Blough (2002) introduced a framework for RTOS-SoC co-design that aids the designers in building a SoC platform architecture and a customizable hardware-software RTOS. With a graphical user interface, the RTOS features can be selected as the number of processors and other software/hardware components. Therefore, complex SoCs with custom configurations of hardware-software RTOS can be generated by this framework tool to speed up applications using a small amount of hardware area.

Oliveira et al. (2011) presented a real-time processor architecture optimized for multitasking real-time embedded systems which efficiently explores modern FPGA technology. A specialized, parameterized and predictable coprocessor that gives hardware support for real-time applications was designed to demonstrate that migration of RTOS functions to hardware allows a faster, predictable and deterministic execution, as well as a reduced RTOS overhead. The developed coprocessors implement the basic RTOS functions, such as task scheduling, interrupt handling, timing, synchronization tasks, etc.

Interrupt handling is an important OS function which has a large impact on both performance and predictability in hybrid systems. In (Liu et al., 2011) a hybrid real-time operating system with two levels of hardware interrupts based on the Advanced RISC Machine (ARM) architecture was implemented with a scheme to

improve the latency of real-time interrupts, as well as other aspects related with performance overhead.

There are other implementations using hardware migration of applications' functionalities, such as FASTCHART (Lindh, 1991), RTBlaze (Wijesinghe, 2008), CUSTARD (Dimond et al., 2005), Silicon (Murtaza et al., 2006), etc. However, only a few support the migration of RTOSes' functionalities, but always rigidly without ever exploring the tradeoffs between different project metrics, namely hardware resources, performance and power consumption.

1.3.3 Design Space Exploration Methodologies

The growing architectural complexity of today's multiprocessor chips and the need to reduce the time-to-market, have forced researches to develop tools capable of performing DSE to find the optimal solution that satisfies all the required metrics for a specific application range. The size of a constraint-based DSE tool is defined by the wide number of customizable parameters used to create an ASIP. For a given application, there are a large number of design alternatives (Radhakrishnan et al., 2006). This number tends to grow with the advancement of technology, allowing flexibility to develop more and more complex designs. These parameters can be tuned to find the optimal configuration in order to meet the required constraints; examples of such parameters are the performance, cost, energy consumption and the available silicon die area.

The major work carried out for ASIP DSE is by using a simulator based approach (Gour and Jain, 2011). However, DSE tools have a problem associated with the number of required simulations to be executed in order to find the optimal architectural configuration for a particular application. The use of "brute force" approaches to search all the important points in the design space could be completely unfeasible due to (1) the great number of existent configuration possibilities and (2) the extremely high simulation time required to simulate all the configurations executing the specific application. Several heuristical approaches have already been applied to address the DSE challenge but none of them respect all the application-specific constraints successfully. The following works present the major contributions to perform efficient DSE and show different approaches and methodologies to tackle the DSE challenge.

Configuring an ASIP by performing exhaustive exploration of the design space is

computationally unfeasible due to the increase of the number parameters introduced. In order to tackle this, Hallschmid and Saleh (2008) proposed a method of modeling the design space using a novel algorithm that uses local regression statistics. The main goals were to find the best configuration of an ASIP, for example to reduce the energy dissipation, using only simulation of a small portion of the design space. This proposal brings great benefits to processors developers since it allows the designers to develop ASIPs without in-depth knowledge of processor architectures. This solution allows drastically reducing the number of simulations required to find the optimal configuration.

Beltrame et al. (2009) proposed the Reflective Simulation Platform which is a transaction-level multiprocessor simulation platform based on the integration of two languages: SystemC and Python. The reflection technique integrates SystemC components without changing the source-code, thus almost no overhead is introduced, and at the same time, provides full observability of the SystemC components internal structures. Using this approach, the simulation platform offers fine-grained control of the simulation, enabling the support to evaluate different hardware/software configurations of a specific application and allowing complete DSE introducing less than 1% of overhead in the simulation comparing with SystemC-only simulation.

Palermo et al. (2009) proposed an efficient DSE methodology for application-specific MPSoCs, ReSPIR, a Response Surface-based Pareto Iterative Refinement. The novelty of the author's approach is their methodology's capability to find a series of good candidate architectures configurations requiring a minimal number of simulations. Their methodology combines design of experiments (DoEs) and Response Surface Modeling (RSM) to manage the system-level constraints. The DoEs generates a plan of experiments used to create a draft of the design space to be then explored by simulations. After, a group of RSM techniques is used to refine the draft created by the DoEs. Basically, the approach is iteratively repeated by performing simulation based refinements of the approximate Pareto set derived from the results given by the RSM model. The RSM techniques explore the application-specific constraints to find the number of feasible solutions.

Radhakrishnan et al. (2006) explored DSE on ASIPs with heterogeneous multiple pipelines in order to efficiently exploit the parallelism at instruction level. For a given application specified in C language and using a simulation approach, the design system can generate a processor with the number of pipeline stages specif-

ically suitable to the application. Results shown that performance improvements can be achieved compared to single-pipeline ASIPs with some overheads in terms of other metrics, such as area, leakage power, switching activity or code size.

DSE can also be realized through high level synthesis as demonstrated in Ahuja et al. (2009). Using an already existing compiler (C2R - C to RTL), capable of automatically translating software code (e.g. in C language) to synthesizable hardware code, encryption and compression software algorithms were automatic converted to the corresponding hardware modules. These modules could be connected to the central CPU using a coprocessor approach in order to reduce the workload of the central CPU. Using such algorithms which automatically generate the corresponding hardware can dramatically reduce the time to market. Also, their proposal allows the control of parameters as timing, area and power consumption (e.g., clock gating) at the high level. Thus, different architectures can be explored at the system level, by changing only the C source code of the functional software model, allowing simulation at several orders of magnitude faster than the corresponding hardware simulation. In addition, with faster simulations, an extensive exploration of the parameters can be tested in short time using the functional model.

Ascia et al. (2004) explored the DSE on parameterized SoC platforms using genetic algorithms. The strategy focuses on exploration of the architectural parameters at the system level for the design of embedded systems with tight power consumption and performance constraints. In order to reduce the time-to-market, some portions of the chip's architecture are predefined for a specific type of application and therefore designing a chip from scratch is avoided. Also, evaluating the various alternatives at lower levels of abstraction require higher design time so the possibility of estimating with a sufficiently high degree of accuracy the impact of several design metrics at highest levels is exploited. This approach was successfully validated for two different parameterized architectures, one based on a Reduced Instruction Set Computer (RISC) processor and another based on a Very Long Instruction Word (VLIW) architecture. However, new techniques to improve the efficiency of DSE are being exploited such as neural networks or mathematical equations to estimate the variables to be optimized as well as the definition of new heuristics.

Kim et al. (2005) exploited the DSE challenges specifically in terms of the area and critical path issues. The main goals were to reduce the hardware cost by

sharing critical functional units which occupy large area and to minimize the critical path by pipelining the critical resources. For fast architecture exploration, all explorations are performed in a SystemC model. Carrying out performance estimation at early stage by transaction level simulation enabled early detection of the optimal architecture specification. Thus, this approach effectively reduced the hardware cost without any performance degradation.

There are also other frameworks which explore the design space of softcore processors such as SPREE (Yiannacouras et al., 2005), UNUM (Dave and Pellauer, 2005), PEAS-III (Kitajima et al., 2001), Xtensa Xplorer Gonzalez (2000) and ARChitect (ARC, 2012) not allowing, however, the gateware acceleration of RTOS functionalities.

In conclusion, a myriad of parameters must be explored to efficiently perform DSE at all levels. The overall goal is to minimize the time required to find the optimal solution for a specific application. From this perspective, there is a need to study other approaches rather than simulation-only approaches in order to perform faster and efficient DSE of constraint-based systems.

The optimization techniques used to explore the solution space of softcore processors (which functionalities should be implemented in hardware or software) are presented in (Labrecque and Steffan, 2007; Dimond et al., 2005). There are several Architecture Description Languages (ADLs) which aim to provide a high-level description of a system, which use DSE in order to optimize the design. The most established ones are LISA (Schliebusch et al., 2002), ArchC (of Computing University of Campinas, 2012) and EXPRESSION (PROGRAM, 2012), while others have recently been created (Metrolho, 2008). Nevertheless, there is still no ADL capable of efficiently performing the entire application development process.

1.3.4 Acceleration Methods

RTOS hardware support implemented as hardware accelerators have been used to achieve power efficiency and to increase the performance and real-time responsiveness of RTOSes. However, coupling these hardware accelerators in the system may induce undesired overheads related with communication (data and signaling) between the processor and the hardware accelerator (Jääskeläinen et al., 2008; Panneerselvam and Swift, 2012). There are different approaches to implement and connect a hardware accelerator (e.g., hardware scheduler) to the processor core.

The hardware scheduler may be connected as a LC module, i.e., hooked up as a peripheral device on the SoC bus; or connected as a TC module, i.e., internally-connected on the core datapath and treated as any other datapath functional unit.

The drawback of the loosely-coupled approach is related to bus arbitration as the communication and synchronization performance can be compromised in systems where Direct Memory Access (DMA) based devices may take control of the bus, blocking or delaying the access to the loosely-coupled hardware scheduler (Tang and Bergmann, 2015). Also, energy efficiency can be compromised as this approach is expensive in terms of resources and power consumed by the required dedicated coprocessor interface (Tang and Bergmann, 2015). Nevertheless, this approach is applied in multicore approaches where other cores, apart from the main core, are used as accelerators, which may be connected using an external bus interface or NoC (Goulding-Hotta et al., 2012).

A more fine-grained approach to connect a hardware accelerator to the processor is coupling the accelerator as a TC module. This approach allows a hardware scheduler (as an example) to be visible to programmers as a regular functional units in the datapath (Jääskeläinen et al., 2008). Communication costs are reduced comparing with the LC approach since communication can be done directly within the datapath.

Table 1.1: Gap Analysis.

	-	2009	2009	2010	2013	2014	2015
	Standard SW RTOS	ReconOS	ARPA-MT	ARTESSO TC-HWRTOS	SEOS	ARTESSO LC-HWRTOS	RT-SHADOWS
Core architecture	Any	PPC/ Microblaze	MIPS32	Purpose-built	Any	ARM	ARM
API execution time	high	low	low	low	low	medium	low
Interrupt response	high	low	low	low	medium	medium	low
HW Scheduler	-	no	LC	TC	LC	LC	TC
Hardware context-switch	no	no	no	yes	no	no	yes
Scalability (no. tasks)	high	high	128	256	high	high	high
Flexibility (fabric tech.)	-	FPGA	FPGA	ASIC	FPGA	ASIC/ FPGA	FPGA
API Agnosticism level	-	posix	no	no	yes	no	yes
Thread Management	sw	hw	hw	hw	hw	hw	hw+sw
Thread Synchronization	sw	hw	hw	hw	hw	hw	hw+sw

"-" does not apply LC: Loosely-Coupled TC: Tightly-Coupled

Another approach to obtain hardware acceleration is based on HW-MT which allows several thread contexts to co-exist inside the datapath, speeding up the processor throughput. Although, many research have implemented HW-MT architectures (Oliveira et al., 2011; Koufaty and Marr, 2003; Labrecque and Steffan, 2007) none have addressed the possibility of running available RTOSes on this architectures, limiting its use to their custom solutions.

1.4 Conclusions

Table 1.1 depicts a gap analysis between the most important recent research compared to our envisioned solution. We conclude that a deterministic and agnostic co-designed hardware-multithreaded architecture may be the optimal solution for applications with high-throughput workloads and time-restricted. Any application invokes RTOS APIs frequently during its execution, therefore implementing several RTOS APIs in hardware will reduce power consumption as well as the CPU time taken by the RTOS, i.e., the system will be power- and performance-efficient independently of the application running on the platform. Doing so, migrating RTOS APIs to hardware combined with hardware multithreading support will

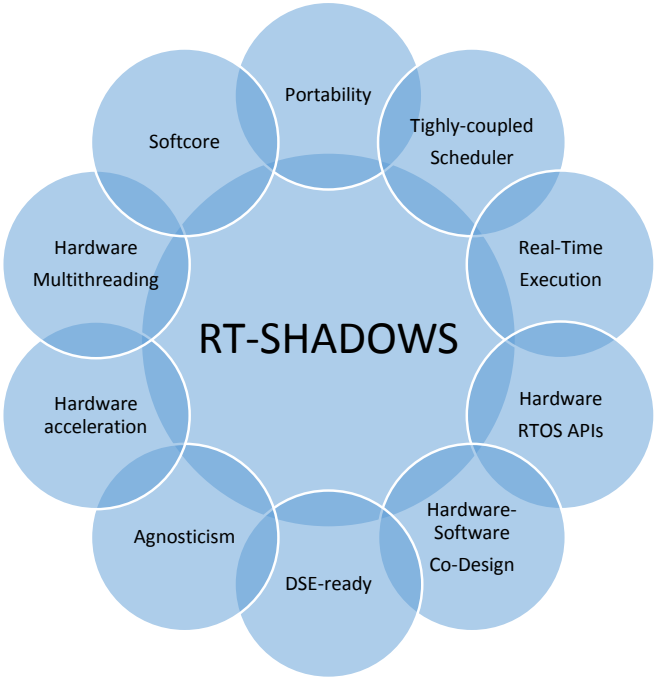


Figure 1.2: **R**eal-**T**ime **S**ystem **H**ardware for **A**gnostic and **D**eterministic **O**Ses **W**ithin **S**oftcore **(RT-SHADOWS)** features.

also reduce the interrupt response time and increase the overall system performance. Connecting a tightly-coupled thread scheduler in the hardware together with hardware multithreading support means that all API functions, including context-switching, are implemented in hardware.

The solution should be able to abstract the architecture to the designer making the RTOS HW-SW interface transparent to the applications and also independently of the RTOS used. A real-time system hardware solution ensuring deterministic execution of RTOSes based on softcore processor will ensure agnosticism by offering hardware multithreading support independently of the RTOS, alleviating the RTOS overhead through hardware acceleration. Additionally, the architecture must feature parametrization capabilities allowing DSE to find the optimal configuration for a particular application. Figure 1.2 depicts the keys features envisioned by us.

1.5 Thesis Structure

This thesis is organized as follows:

- Chapter 2 describes the implemented framework based on the ARM architecture and corresponding software stack used as development, testing and prototyping platform.
- Chapter 3 presents the micro-architectural extensions applied to the softcore processor, presented in the Chapter 2, to support hardware multithreading execution. We describe the main features of the flexible hardware multithreading support and its implementation details. Namely, we show how this implementation is configurable using a set of parameters, maintaining scalability and portability.
- Chapter 4 presents a novel hardware approach to solve the rate-monotonic problem found in many RTOS solutions. We describe how our approach unifies the priority space at the interrupt handling sub-system with enhancements on predictability and minimal hardware cost.
- Chapter 5 presents our portable architecture which provides a unified hardware-software scheduling, bringing the benefits of HW-MT to the RTOS domain. We show the benefits of our architecture in terms of performance and deter-

minism with very low area usage/performance overhead ratio.

- Chapter 6 presents our agnostic software stack based on a co-designed hardware-software transparent solution which enables current RTOS solutions to benefit from hardware acceleration without being intrusive to the software layer.
- Chapter 7 describes the integration of the whole system stack with the hardware multithreading support. We present our co-designed architecture and its impact on memory footprint.
- Chapter 8 concludes this thesis. We present the conclusions of our research and the limitations encountered during the development of our solution. To conclude, we suggest the future work towards fulfilling the aforementioned limitations.

Chapter 2

Research Platform

This chapter presents the implemented framework based on the ARM architecture and corresponding software stack used as a development, testing and prototyping platform. We describe the main reasons behind the choice of ARM architecture, Xilinx FPGAs and the software stack used.

The remainder of this chapter is organized as follows: Section 2.1 identifies the platform requirements to realize the proposed work; Section 2.1.1 presents the AT91SAM9XE SoC Clone deployed in a Xilinx FPGA platform and its implementation results in terms of occupied area; Section 2.1.2 concludes by describing the verification methodology used to validate the correctness of the platform.

2.1 Platform Requirements

To accomplish the work proposed by this thesis several platform requirements were identified in order to find the most suitable development platform. When developing an embedded system solution, several factors must be taken into consideration such as processor architecture, portability and flexibility in order to allow micro-architectural modifications. The following requirements were established:

1. **Processor architecture:** As one of most widely used architecture in the embedded world, ARM is selected as the ideal processor architecture. There is confidence in the ARM Instruction Set Architecture as they have been used in a myriad of embedded systems throughout the world and are very popular in today's embedded system market. As well known, ARM processors family

offers one of the best balance between processing performance and energy power consumption making it suitable for embedded systems. Furthermore, ARM already provides a completely established tool-chain which offers a collection of programming tools for developing applications and operating systems.

2. **Portability:** A portable platform will reduce the time-to-market of embedded systems solutions. Therefore, a commercial ARM-based SoC supporting several (Real-Time) Operating Systems ports is the adequate SoC platform.

3. **Flexibility:** In order to apply the micro-architectural extensions proposed by this thesis, the architecture must be editable. Hence, hardcore processors are out of the scope. This implies the use of an FPGA-based solution, promoting design re-use to improve productivity, prototyping of new solutions. Additionally, FPGA-based solutions provide an easy approach to perform the measurement of each new micro-architectural extension in terms of area, energy and performance.

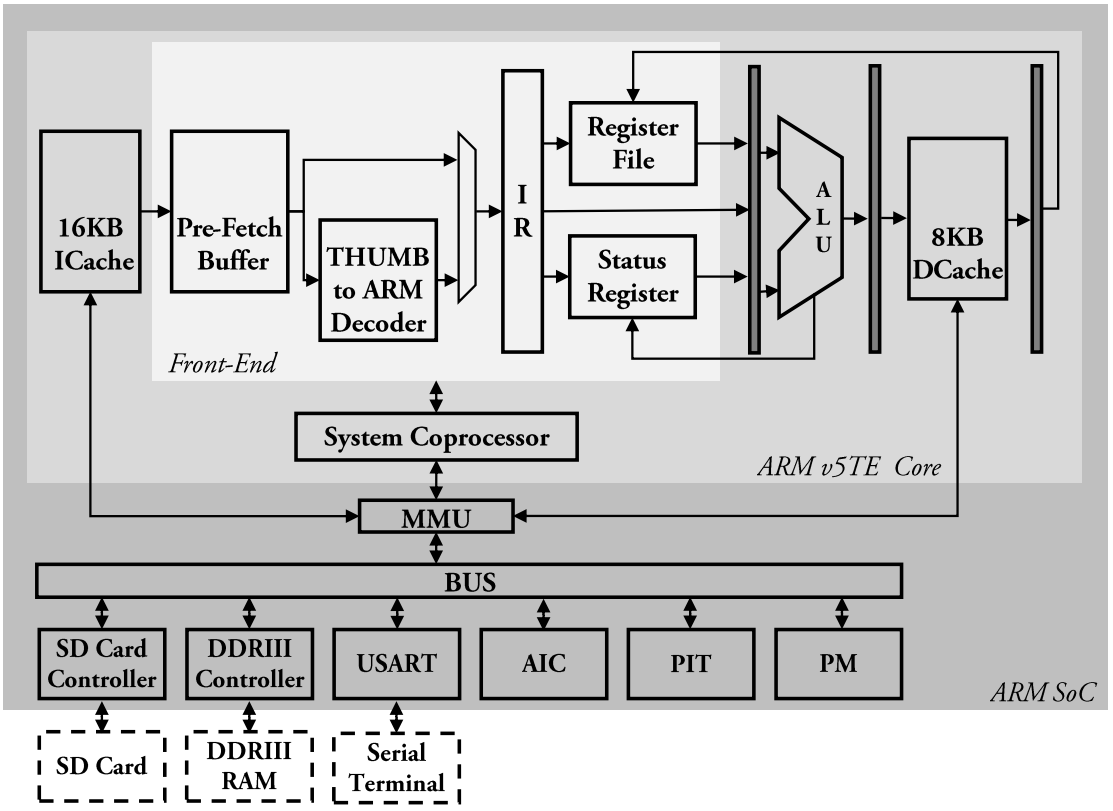


Figure 2.1: AT91SAM9XE SoC block diagram.

Considering the lack of existing solutions complying with the aforementioned requirements, we decided to develop an in-house softcore based on the ATMEL AT91SAM9XE SoC (Atmel, 2014). The decision of developing the SoC from scratch arose from the need to apply improvements at both core- and SoC-levels which would not be possible with an IP module. Due to the availability of FPGA platform, the first version of the SoC was deployed on a Xilinx Virtex 5 FPGA using Xilinx ISE Design Suite 14.5. However, the platform was later upgraded to a Xilinx Kintex 7, using Xilinx Vivado Design Suite 2014.4, which offers more logic cells to accommodate the new micro-architectural extensions.

2.1.1 AT91SAM9XE SoC Clone

This section describes the implemented ARM architecture. The developed architecture is based on the widely used commercial AT91SAM9XE SoC by Atmel (2014). This SoC implements the ARMv5TE instruction set and it is ready to be prototyped on a Xilinx Virtex 5 or Kintex 7 FPGA boards. Figure 2.1 depicts

Table 2.1: Detailed hardware utilization results on Xilinx Kintex 7 - XC7K325T.

Module	**Slice** (50950 available)	**Slice LUTs** (203800 available)	**Slice Registers** (407600 available)
⊟ ARM SoC	14808	47406	26829
⊟ ARM System	8103	26322	8602
⊟ ARM Core	7103	24919	4130
⊞ I. Cache	2358	7903	962
⊞ Front End	3193	10216	1693
Buffer Stage 2	893	1295	363
⊞ ALU	93	217	0
Buffer Stage 3	326	497	79
⊞ D. Cache	1486	4204	649
System CP	400	599	384
⊞ MMU	1501	1403	4472
⊞ Memory Controller	4152	12979	9379
PIT	49	21	136
PM	196	181	435
Remap	2	3	4
⊞ SD Card Controller	1164	3751	4478
System Bus	1552	2742	1455
⊞ AIC	1123	802	2212
⊞ USART	53	53	123

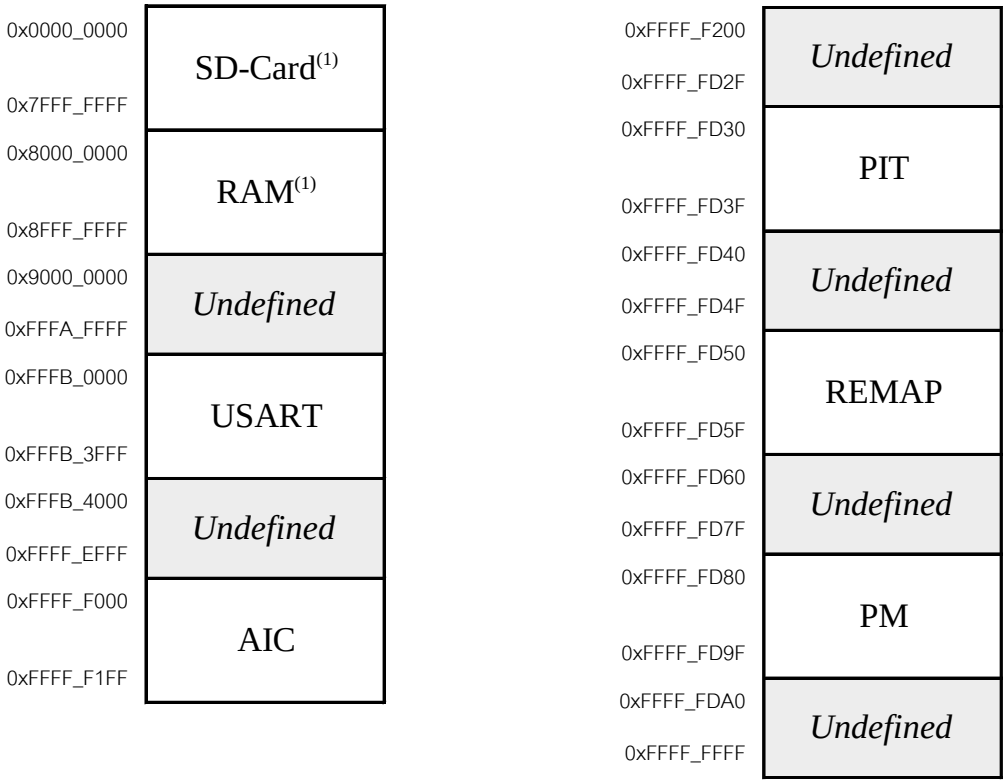
the top-level view of the implemented architecture. Table 2.1 shows the detailed hardware cost for each module when deployed on Kintex 7 FPGA Embedded Kit (XC7K325T) with default synthesis strategy. All the modules were developed in-house from scratch using Verilog Hardware Description Language (HDL) (Palnitkar, 2003), excluding the Double Data Rate (DDR) Memory Controller which is provided as an IP core from Xilinx (2014). The SoC in Figure 2.1 integrates our ARM-compliant processor core, a hardware-based page table walking Memory Management Unit (MMU) and a sub-set of peripheral devices required to fully test software-only Operating Systems. The implemented peripheral devices are a SD-Card Controller, a DDR Controller, a Universal Synchronous Asynchronous Receiver Transmitter (USART), an Advanced Interrupt Controller (AIC), a Periodic Interval Timer (PIT) and Performance Monitor (PM). All the modules, but the DDR Controller, are running at 33 MHz.

ARM-compliant Softcore Processor

The implemented ARM softcore processor is completely compliant with the ARMv5-TE instruction-set (Seal, 2000). Our implementation is based on the micro-architectural information of the ARM926EJ processor available in Atmel (2010). Figure 2.1 depicts the internal modules of the processor core which features a 5-stages pipelined core, a 8 KB 8-way set associative data cache, a 16 KB 16-way set associative instruction cache (both virtually-tagged/virtually-indexed) and a system coprocessor (ARM's CP15). We also implemented a Thumb to ARM decoder in order to support THUMB code (Goudge and Segars, 1996).

Clock Synchronization

Clock synchronizers were implemented to avoid metastability events that commonly arise from using multiple clock domains on the same chip. A synchronization scheme based on the Two-Flip-Flop Synchronizer (Ginosar, 2011) was used to synchronize the accesses between the memory controller and the remaining SoC modules.



(1) SD-Card or RAM depending on REMAP configuration

Figure 2.2: Address memory layout.

Performance Monitor

To allow a fine-grained measurement of the processor performance, a Performance Monitor module was developed and integrated as a peripheral device. This module was designed to count precisely how many clock cycles takes to execute either one or a set of instructions previously specified, ensuring high-level observability through the whole system performance. The PM counter is capable of counting the clock cycles of an instruction, from the time it is being issued by the processor from the memory until its last pipeline states (write-back) is accomplished. This module can be configured to provide an interrupt trigger to the AIC when a clock counting has finished. This way, the code being measured does not need to be modified, usually with a *start timer* and *stop timer* functions, and its counter value will be easily outputted by an Interrupt Service Routine (ISR). The main purpose of this module is to measure the latency of a set of instructions and also their execution jitter when executed more than once.

Memory Layout

The system memory is organized as depicted in Figure 2.2. On the reset state, the SD-Card memory is mapped at the address 0x0. This allows a bootloader to be loaded into the non-volatile memory and perform all the initializations before running an application. A REMAP feature was implemented in order to modify the memory layout by swapping the SD-Card and Random Access Memory (RAM) memory positions. This is useful for applications where the exception vector table is configured to be at the first memory address positions (and at RAM memory) thus allowing exceptions and interrupt requests to be quickly handled since the RAM memory interface is faster than the SD-Card interface. Usually, a bootloader loaded in the SD-Card memory can be booted, copying all software binary code to RAM memory, performing the remap command and finally branching to address 0x0 which is now RAM memory.

Cachability of Peripheral Devices

The ARM architecture reference manual specifies as *implementation defined* the possibility of allowing caches to be enabled while the MMU is disabled. These features are configured through the system coprocessor (CP15), while the MMU is responsible for managing whether an instruction fetch or a data access is treated as cachable or uncachable. Having the MMU disabled and the data cache enabled will result on all data accesses being treated as cachable accesses. This will restrict the access to memory mapped peripheral devices since any access to it will be cached, i.e., whether *read* or *write* operations will not be directly bypassed to the hardware.

In order to obtain the performance benefits from enabling the data cache in RTOSes, such as FreeRTOS, where the specific-port for this architecture does not use MMU, we have implemented a new register on the coprocessor CP15 enabling setting peripheral devices as uncachable when data cache is enabled and MMU is disabled. Hence, all data accesses will be treated as cachable while data accesses to memory mapped peripheral devices will be treated as uncachable. Therefore, RTOS applications may benefit from increased performance from enabled caches if necessary.

2.1.2 Verification

Module Verification

Each module is individually tested and verified to find logical errors. Module-specific testbenches are performed to validate the functionality of each component individually. If a module can be stand-alone tested in the hardware, (e.g., SD CARD reader, DDR3 Memory Controller), then it is also deployed on the platform and tested again to ensure correctness on the gate-level implementation.

Processor and SoC level Verification

All modules are integrated together after testing them individually. Usually, the initial tests to validate a processor design consist of short assembly applications targeting data and control flow of instructions executing through the pipelined stages (e.g., ALU instructions). For the first steps, these programs are loaded into the processor using a Register-Transfer Level (RTL) simulator. However, RTL simulators perform slowly large RTL designs, therefore, after extensive testing, more complex applications are loaded into the processor and executed on the hardware. Using on-chip debuggers such as Xilinx's ChipScope Pro (Xilinx, 2012) it is possible to detect potential faults on the design. To guarantee an error-free design, elaborated software applications must be used to build complex software contexts which will increase the design coverage. Operating systems, due to their complexity, are examples of complex software contexts used for verification purposes (Chen et al., 2013). Therefore, an RTOS or a General Purpose Operating Systems (GPOS) are good solutions to increase design coverage. In our case, we have used FreeRTOS (Barry, 2010), μCOSII (Labrosse, 2015) and Linux to prove the correctness of our design. After successfully running all the mentioned OSes on our platform we considered our design error-free for the moment, increasing the degree of confidence on the designed SoC.

Chapter 3

Hardware Multithreading Extensions

This chapter presents the micro-architectural extensions applied to the softcore processor, presented in the Chapter 2, to support hardware multithreading execution. We describe the main features of the flexible hardware multithreading support and its implementation details. Namely, we show how this implementation is configurable using a set of parameters while maintaining scalability and portability.

The remainder of this chapter is organized as follow: Section 3.1 presents the hardware extensions made to the single-threaded ARM core datapath to support hardware multithreading (i.e., SMP threading a.k.a Silicon SMP); Section 3.2 briefly presents the tightly-coupled hardware scheduler which will be more detailed in Chapter 5; Section 3.3 describes the modifications made to specific instructions required for multithreaded execution; Section 3.4 and Section 3.5 describe the hardware support required for delay and synchronization between threads; Section 3.6 concludes by presenting the coprocessor instructions used to manage the hardware multithreading support.

3.1 Hardware Multithreading Support

HW-MT is a processor-level optimization technique used to improve throughput which can be used to improve real-time responsiveness of real-time systems. This

feature ensures multiple thread contexts within the same core allowing multiple distinct thread contexts to be switched in hardware without saving and restoring them on software. HW-MT has been established as an architectural feature to maximize throughput in a processor. The idea is to maintain the processor running at maximum throughput by executing several hardware-supported threads. HW-MT can increase throughput by up to 25% (Koufaty and Marr, 2003) by hiding idle times such as memory latencies or branch penalties (Sodan et al., 2010). A HW-MT architecture typically encompasses per thread replication of the architectural units on a processor (e.g., register-file, status register, program counter, etc.) and a hardware scheduler in charge of managing threads execution flow. This approach has been proved to offer better processor utilization (throughput) when compared to single-threaded cores (Dimond et al., 2005). However, traditional HW-MT architectures are applied to Server/Desktop applications (Koufaty and Marr, 2003) and are not suitable for RTOSes commonly used in embedded systems. HW-MT schedulers use their own thread scheduling algorithm (e.g., blocking multithreading (BMT), interleaved multithreading (IMT) or simultaneous multithreading (SMT)) to explore chip utilization. This hardware level scheduling diverges from RTOS software scheduling, resulting in a hierarchical scheduling policy which breaks the expected RTOS execution flow and established static analysis methods; e.g., MIPS32 SMT-based HW-MT support (Oliveira et al., 2011) requires equivalent SMT software scheduler.

In our architecture, this is accomplished by replicating thread-specific registers in the processor datapath such as register-file and status-register. The main requirements for the HW-MT support are parameterization (i.e., configurable number of hardware-supported threads, number of synchronization mechanisms, etc) and OS agnosticism (i.e., the HW-MT support must be flexible enough to support different features from different RTOSes, therefore it should be RTOS-independent).

The datapath of the ARM softcore was extended to support multithreading execution. All the modifications made to the base single-threaded ARM core are controlled by software, i.e., by default (on reset state) the processor acts as a regular single-threaded core, the application may (or may not) use the HW-MT support. Depending on the application demands, the number of hardware-supported threads is configurable from 4, 8, 16, 32, 64 and 128 threads at design time, which are managed by a hardware thread scheduler described in Section 3.2. By default, ARM architecture uses overlapped banked registers which are visible depending on the current CPU mode (Seal, 2000). At any moment, 8 unbanked general-purpose

Table 3.1: ARM mode register view: banked registers are highlighted.

CPU Mode					
User/ System	**Supervisor**	**Abort**	**Undefined**	**Irq**	**Fiq**
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12_fiq
R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

registers (`R0-R7`), 7 general-purpose banked registers (`R8-R14`), one or two status registers and the program counter (`PC`) are available for usage. Table 3.1 illustrates the visible registers based on the CPU mode.

To ensure HW-MT support each thread must possess its own general-purpose registers. However, the replicated registers per thread must match the CPU mode in which RTOS threads execute, i.e., different RTOSes run their threads in different CPU modes. For instance, FreeRTOS's threads and embOS's threads run in sys-

Table 3.2: Multithreaded ARM mode register view: banked registers are highlighted.

THREAD								
TH #0						TH #1	TH #2	TH #n
CPU Mode								
User/ System	**Supervisor**	**Abort**	**Undefined**	**Irq**	**Fiq**	**Software Configurable**		
R0	R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_fiq	R8	R8	R8
R9	R9	R9	R9	R9	R9_fiq	R9	R9	R9
R10	R10	R10	R10	R10	R10_fiq	R10	R10	R10
R11	R11	R11	R11	R11	R11_fiq	R11	R11	R11
R12	R12	R12	R12	R12	R12_fiq	R12	R12	R12
R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	R13	R13	R13
R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq	R14	R14	R14
R15	R15	R15	R15	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq			

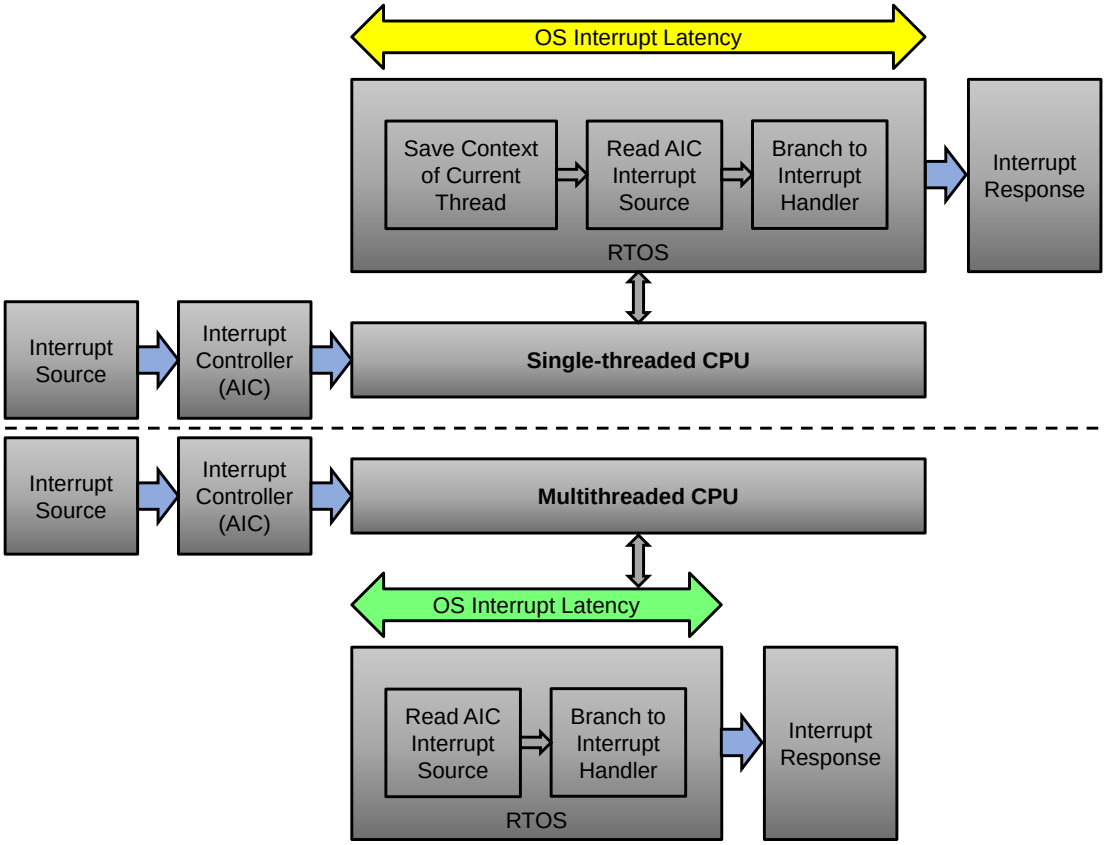


Figure 3.1: OS overhead to interrupt latency: Single-threaded CPU versus Multi-threaded CPU adapted from Sheikh and Driscoll (2011).

tem mode (Barry, 2010; Segger, 2008) while μCOSII's threads run in supervisor mode (Labrosse, 2015). In order to offer flexibility and compatibility with several RTOSes, the replicated registers mode can be configurable by software, matching RTOSes' specification. Table 3.2 depicts the visible registers on the HW-MT ARM architecture.

Another advantage of a HW-MT architecture is related to interrupt handling. Usually, each time an interrupt is triggered there is an associated OS interrupt latency overhead (Sheikh and Driscoll, 2011) as the RTOS is in charge of saving the context of the current thread, finding the interrupt source and starting the execution of the corresponding ISR. In our HW-MT architecture, a hardware-supported thread is dedicated to the kernel and so, it is assumed the RTOS kernel just needs one thread, which is true for FreeRTOS, μCOSII, embOS. Hence, when an interrupt is triggered, if a hardware-supported thread (excepting the kernel one) is running, the context of the thread does not need to be saved as it has its own dedicated registers, shortening the OS interrupt latency overhead. If the running thread is the one dedicated to the kernel, thus context-switch is performed by

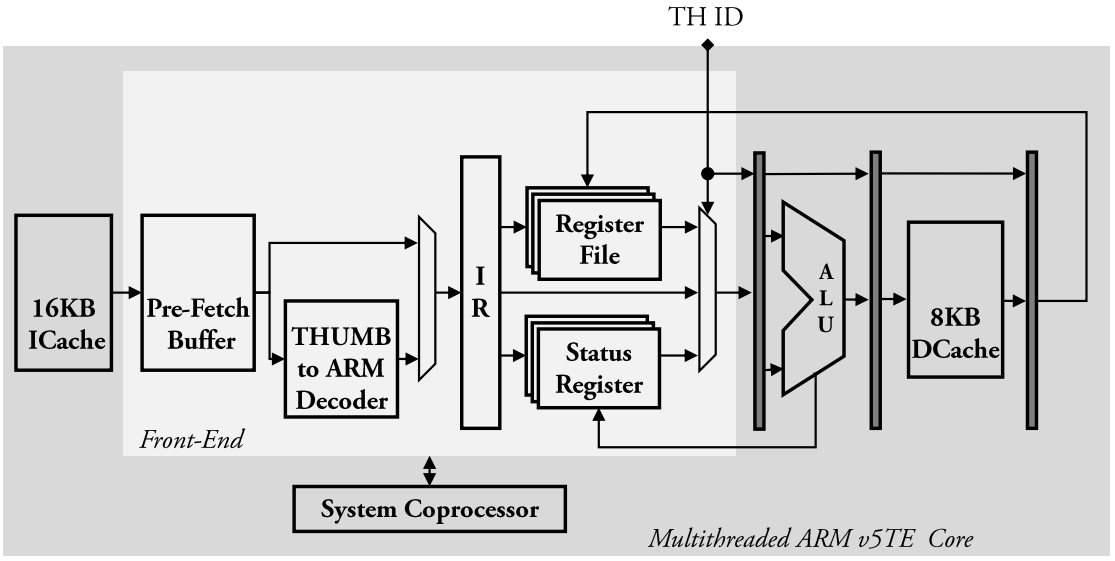


Figure 3.2: Multithreaded ARM core block diagram.

software as previously done by the RTOS. Figure 3.1 depicts the interrupt handling on a single-thread processor versus a multithreaded processor. Scalability is also guaranteed by this dedicated hardware-supported thread. For instance, in area-constrained platforms, where the RTOS application may create more threads than the hardware-supported ones, this dedicated thread can be used to run regular software threads, not limiting the number of created threads to the number of hardware-supported ones and so, keeping our architecture scalable.

A new signal containing the thread identification was added throughout the pipelined datapath in order to identify which is the thread being executed on each pipeline stage. The main usage of this signal is related to hazard detection (i.e., avoiding incorrect detection of hazards among different threads) and also to the saving of correct data on the corresponding write-back thread registers. Figure 3.2 presents a simplified block-diagram of the multithreaded ARM core datapath.

3.2 Tightly-Coupled Scheduler

Thread scheduling has been identified as one of the major performance bottleneck of RTOSes (Kohout et al., 2003). In order to alleviate the RTOS scheduling overhead we decided to offload the scheduler to FPGA fabric. Taking advantage of the HW-MT support, the tightly-coupled approach emerged as the best solution. A tightly-coupled scheduler can communicate with the processor core in a quick

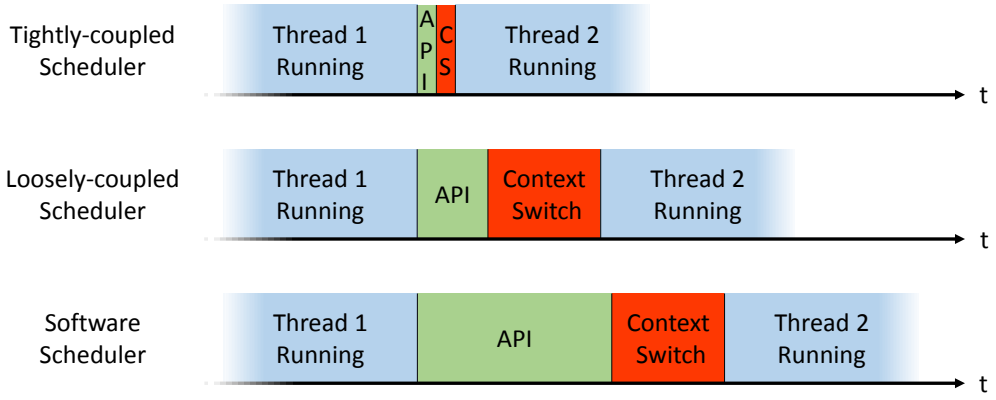


Figure 3.3: Tightly-coupled vs loosely-coupled vs software-only scheduler.

and predictable way. Figure 3.3 shows the benefits of a tightly-coupled scheduler over a loosely-coupled or a software-only scheduler.

Our hardware scheduler is implemented as a tightly-coupled ARM coprocessor. All the interface between the coprocessor and the core is provided through Move to Coprocessor from ARM Register (`MCR`) and Move to ARM Register from Coprocessor (`MRC`) magic instructions (Hameed et al., 2010). Magic instructions are instructions that typically interface with custom storage structures, such as designer-defined register-files (DDRFs), and perform hundreds of operations at once. These instructions can have a significant effect on both the energy and performance of an application by minimizing the communication bandwidth and power at all levels of the memory hierarchy, (registers, caches and memory) (Hameed et al., 2010). They alleviate the pressure of fetching multiple instructions (to perform the equivalent operation) as well as several translations and transactions between the processor and memory. Our hardware scheduler offers a high degree of configurability. Using software instructions, it is possible to configure: (1) the scheduling algorithm and the use of round-robin scheme; (2) configure the thread priority levels, i.e., if low or high priority values denote low or high priority threads; (3) read or modify the state of all the threads and (4) enable or disable the scheduling;

A thread is represented in the hardware by a DDRF which stores the priority, state, handler and stack pointer information of each thread. Figure 3.4 represents the state machine of a hardware-supported thread. In order to be flexible to any RTOS, a hardware-supported thread can be created with a configurable initial state (*ready* or *suspended* state) and with any priority. In addition, the thread's stack pointer can also be initialized as well as the thread's parameters.

On a priority-based scheduling algorithm it is necessary to find the highest priority

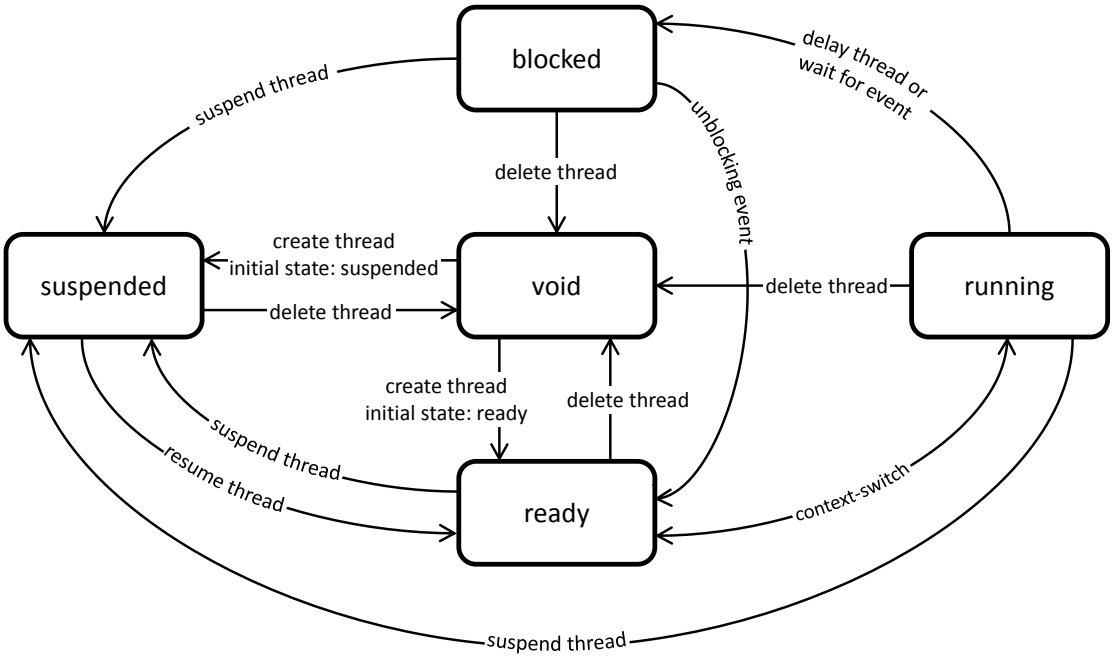


Figure 3.4: Hardware threads state machine.

thread. Our hardware scheduler implements a balanced binary tree to find the highest priority value from all the thread priorities. As explained before, the priority order is configurable. In RTOSes where a round-robin scheme is used as a background or tiebreaker scheduler on threads sharing the current highest priority value, a hardware arbiter with fairness scheme based on Altera (2011b) was implemented. Figure 3.5 depicts a simplified diagram of the hardware designed to find the next highest priority thread to be executed next. The balanced binary tree is in charge of finding the highest priority value which is then compared with the value of all thread priorities in *ready* state. The top-priority vector, containing one bit for each thread, indicates whether or not a thread has the highest priority and is in *ready* state. For instance, if the bit 1 and 3 of the top-priority vector is set to 1, it means that threads 1 and 3 are *ready* to execute and have the highest priority. The implemented arbiter with fairness algorithm is fed with the top-priority vector and with the running thread in order to output the *id* of the next thread to be executed. Following the example given on Figure 3.5, the arbiter will set thread 3 to be executed considering that thread 1 is running when a magic context-switch command/instruction is issued.

The currently implemented hardware-based scheduler is supported by the following hardware APIs (hAPIs):

- **Create Thread:** Initializes a new hardware-supported thread with an initial

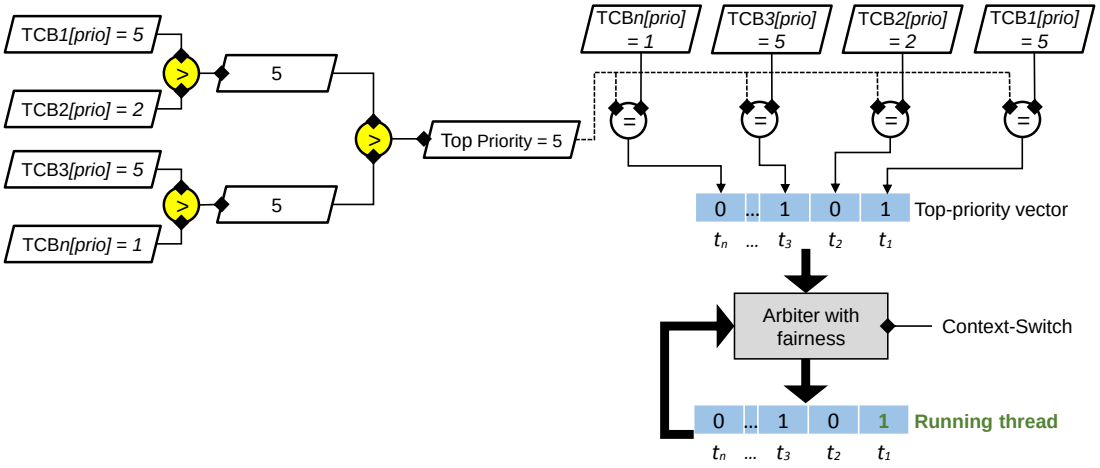


Figure 3.5: Priority-based scheduling logic.

state, priority, stack pointer and arguments;

- **Delete Thread:** Eliminates a hardware-supported thread, freeing a hardware-supported thread slot;
- **Resume Thread:** Sets the state of a thread as *ready*;
- **Suspend Thread:** Sets the state of a thread as *suspended*;
- **Set Priority:** Modifies the priority of a thread;
- **Get Priority:** Returns the priority of a thread;
- **Start Scheduler:** Starts the scheduling algorithm pre-configured;
- **Suspend Scheduler:** Suspends the scheduling algorithm;
- **Delay Thread:** Delays a thread by a specific number of OS ticks;
- **Create Mutex:** Initializes a hardware mutex slot;
- **Delete Mutex:** Empties a hardware mutex slot;
- **Take Mutex:** Thread holds a mutex, optionally thread can wait a specific time (in OS ticks) until mutex becomes available;;
- **Give Mutex:** Thread releases a mutex;
- **Create Semaphore:** Initializes a semaphore slot;
- **Delete Semaphore:** Empties a semaphore slot;

- **Take Semaphore:** Thread takes a semaphore, optionally thread can wait a specific time (in OS ticks) until semaphore becomes available;
- **Give Semaphore:** Thread gives a semaphore;

The thread execution flow is managed by a priority-based scheduling algorithm, i.e., the highest priority thread should always be the thread being currently executed. Different RTOSes use different priorities order, meaning that a thread with priority 0 may be the highest priority thread in some RTOSes or may be the lowest priority thread in other RTOSes. Our flexible hardware support allows this property to be software configurable by setting the priorities order accordingly. Our scheduler allows preemptive or cooperative mode. In preemptive mode, at each OS tick, the scheduler will preempt the currently running thread if there is a higher priority thread ready to run. In cooperative mode, the currently running thread (with the highest priority) executes until it is blocked by some event or it deliberately yields its execution, allowing another thread to be dispatched. Our scheduler allows the optional use of round-robin scheme between threads sharing the highest priority value, which is once again software configurable. This is useful to support different RTOSes; FreeRTOS uses round-robin scheme while μCOSII does not allow different threads to have the same priority value.

3.3 Modified Instruction Behavior

In order to accomplish the HW-MT support, the behavior of some ARM instructions has been modified. However, these modifications are not hardwired, they still execute as expected by ARM ISA specification and their behavior is only changed upon software configuration and can be restored back at run-time. The modified instructions are the Load Multiple (`LDM`) instruction and Store Multiple (`STM`) instruction.

- `LDM <Rn>,<register_list>^` : This form of `LDM` loads a sub-set registers from the CPU mode *user* when the processor is in a privileged mode. This instruction is commonly used by RTOSes for context switches.

The main use of this instruction is to allow the loading of multiple registers from consecutive memory locations. The modification made to this instruction relies on the registers list to be loaded from memory. The behavior of the `LDM` instruction can be modified using `MCR` instructions. These modifications are useful for two different

occasions: (1) initialize a thread-specific register-file while a new thread is created, specifically the thread argument (usually stored in register `R0`) and thread stack pointer (register `R13`); and (2) instead of loading to *user* mode registers as defined in ARM specification, the loading is done to the current thread-specific register-file independently of the configured thread CPU mode, as shown in Table 3.2.

- `STM <Rn>,<register_list>^` : This form of `STM` stores a sub-set of registers from the CPU mode *user* to sequential memory locations.

Usually, this instruction is used when the processor is in a privileged mode and it is necessary to store registers from user mode. If the processor is in a privileged mode, in our architecture means that the thread 0 is being used. So, the modification of this instruction allows the registers from the previously executed thread to be stored instead of the *user* mode registers from thread 0. For instance, this is useful when a context-switch is going to be performed and we need to store the return address of the previously executed thread in memory.

3.4 Delay Timers

One of the features present in many RTOSes is thread delay or thread sleep utilized to block the execution of a thread for a specific number of OS ticks. Blocking a hardware-supported thread implies specific hardware support in order to deterministically manage the thread state. Usually, this support implies a hardware timer for each thread; a thread-dedicated timer is loaded with the number of OS ticks to block a thread; at each OS tick the timer is decremented; when the timer

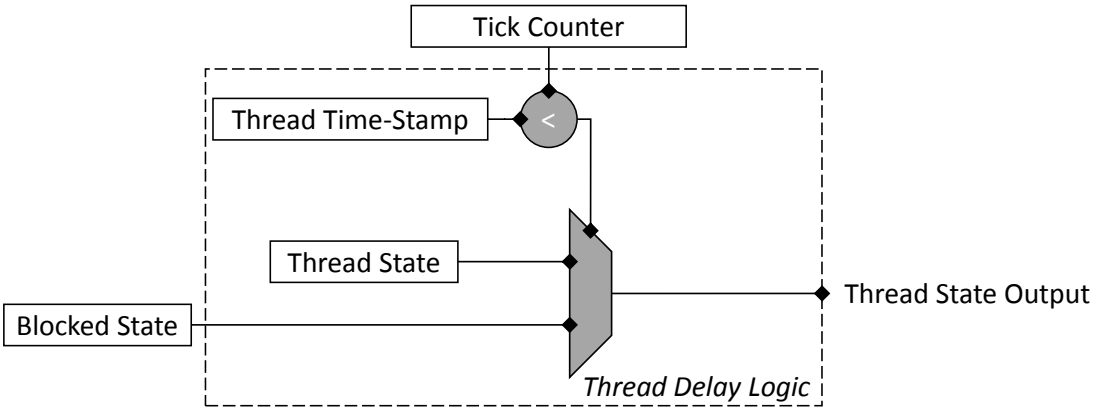
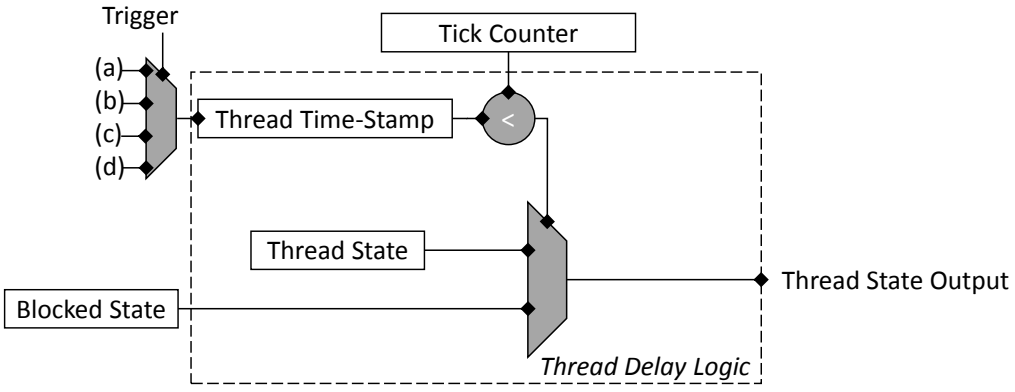


Figure 3.6: Delay logic for each thread.

expires the thread state changes to *ready*.

In order to optimize the hardware cost, we have implemented a different approach, shown in Figure 3.6. Instead of having a timer for each thread, there is only a single timer in charge of counting OS ticks. Then each thread possesses a time-stamp register containing the number of ticks for which the thread must block until the OS tick timer counter overpasses the time-stamp value. For instance, supposing that the tick timer counter has counted 20 ticks so far, and the thread A wants to delay for 5 OS ticks, then the time-stamp register of thread A is loaded with the value 25. As soon as the OS tick timer counter reaches 25, the thread A will change to *ready* state. This approach avoids the use of an array of timers which is expensive in terms of area for a great number of timers. This feature is also optional.



Trigger:
(a) Set thread delay
(b) Set mutex or semaphore blocking time
(c) Reset blocking time, if the mutex or semaphore was given by another thread
(d) Blocking time expired and thread does not want to wait for the mutex or semaphore anymore

Figure 3.7: Using delay logic of each thread to support mutex/semaphore blocking time.

3.5 Synchronization Mechanisms

Mutexes and semaphores are among OS primitives typically provided by RTOSes. Mutexes are used to ensure mutual exclusion to protect a shared resource while semaphores are mainly used for synchronization purposes. Usually, mutex and semaphore APIs allow threads to enter a blocking state while waiting for a specific event. For example, if a thread A wants to take a mutex already taken by a

thread B, thread A can set a specific time, in OS ticks, to wait for the mutex to become available. In order to take advantage of available hardware, the hardware used to manage thread delay (presented in Section 3.4) is also used to implement the blocking time of mutex and semaphore APIs. Figure 3.7 depicts the extra hardware added to thread delays logic to support mutexes and semaphores blocking time. Apart from this extra hardware, two bits are required for each mutex; one bit to identify if the mutex is created or not; and another bit to identify if the mutex is taken or not. Semaphores require two extra registers to hold the initial semaphore counter and the maximum count value. The number of hardware-supported mutexes and semaphores is also parameterisable.

3.6 CoProcessor Magic Instructions

Coprocessor instructions are used to configure all the RTOS hardware support. The `MCR` and `MRC` magic instructions offer several instruction fields allowing several different data to be obtained using the same instruction. The hardware scheduler is mapped on coprocessor number 14 (`CP14`) and contains several registers responsible for different features. Tables 3.3, 3.4, 3.5, 3.6, 3.7, 3.8 present all the available coprocessor registers and corresponding functionalities.

Table 3.3: Register `C0`: Hardware-supported threads initialization.

Writing to	`C0, op0`	indicates the user wants to initialize a new thread
Reading from	`C0, op0`	returns the number of created threads
Writing to	`C0, op1`	sets the priority of the new thread
Reading from	`C0, op1`	returns the number of suspended threads
Writing to	`C0, op2`	sets the handler of the new thread
Reading from	`C0, op2`	returns zero
Writing to	`C0, op3`	sets the top of stack of the new thread
Reading from	`C0, op3`	returns zero
Writing to	`C0, op5`	sets the initial top of stack of the new hw thread
Reading from	`C0, op5`	returns zero
Writing to	`C0, op7`	initialize a free thread with settings previously defined
Reading from	`C0, op7`	returns zero

Table 3.4: Register `C1`: Hardware-supported threads interface.

Writing to	`C1, op0`	selects the thread using its handler
Reading from	`C1, op0`	returns the handler of the selected thread; if `0xFFFFFFFF` is returned, no thread was and all actions below are ignored
Writing to	`C1, op1`	sets the priority of the selected thread
Reading from	`C1, op1`	returns the priority of the selected thread
Writing to	`C1, op2`	sets thread state as blocked, suspended, ready or running
Reading from	`C1, op2`	returns the current state of the selected thread
Writing to	`C1, op3`	sets the top of stack of the selected thread
Reading from	`C1, op3`	returns the top of stack of the selected thread
Writing to	`C1, op5`	sets a new handler for the selected thread
Reading from	`C1, op5`	returns zero
Writing to	`C1, op6`	returns the highest priority value from threads on ready state
Reading from	`C1, op6`	returns zero
Writing to	`C1, op7`	sets a thread delay for the running thread
Reading from	`C1, op7`	returns zero

Table 3.5: Register `C2`: Hardware scheduler interface.

Writing to	`C2, op0`	triggers a hardware context switch
Reading from	`C2, op0`	returns zero
Writing to	`C2, op1`	ignored
Reading from	`C2, op1`	returns the handler of the running thread
Writing to	`C2, op2`	ignored
Reading from	`C2, op2`	returns the priority of the running thread
Writing to	`C2, op5`	sets the new top of stack of the running thread
Reading from	`C2, op5`	returns the top of stack of the running thread
Writing to	`C2, op6`	increments the hardware tick counter
Reading from	`C2, op6`	returns zero
Writing to	`C2, op7`	sets the state of the scheduler as running or suspended
Reading from	`C2, op7`	returns the state of the scheduler

Table 3.6: Register C3: Configure behavior of LDM/STM instructions.

Writing to	C3, op0	modifies the behavior of the LDM/STM instructions
Reading from	C3, op0	returns 0 if instructions are not modified, returns 1 if instructions modified
Writing to	C3, op1	sets the mode of the replicated registers
Reading from	C3, op1	returns the mode of the replicated registers
Writing to	C3, op2	sets how a thread can be identified by itself (e.g., FreeRTOS uses NULL parameter, μCOSII uses OS_PRIO_SELF)
Reading from	C3, op2	returns how a thread is identified by itself
Writing to	C3, op3	Sets the order of priorities: 0: means that priority 0 is the highest priority 1: means that priority 0 is the lowest priority
Reading from	C3, op3	returns the order of priorities

Table 3.7: Register C4: Hardware-supported mutexes interface.

Writing to	C4, op0	selects a mutex
Reading from	C4, op0	returns the handler of the selected mutex, if 0xFFFFFFFF is returned, the mutex was not found
Writing to	C4, op1	trying to take a mutex
Reading from	C4, op1	returns true if the selected mutex is taken; false if selected mutex isn't taken nor is already created
Writing to	C4, op2	give a mutex
Reading from	C4, op2	returns zero
Writing to	C4, op5	sets the blocking time that the thread wants to wait for a mutex already taken
Reading from	C4, op5	returns zero
Writing to	C4, op6	creates a mutex
Reading from	C4, op6	returns zero
Writing to	C4, op7	deletes a mutex
Reading from	C4, op7	returns the next free mutex slot, if 0xFFFFFFFF is returned, all mutexes slots are being used

Table 3.8: Register C5: Hardware-supported semaphores interface.

Writing to	C5, op0	selects a semaphore
Reading from	C5, op0	returns the handler of the selected semaphore, if 0xFFFFFFFF is returned, the semaphore was not found
Writing to	C5, op1	trying to take a semaphore
Reading from	C5, op1	returns true if the selected semaphore is taken; false if selected semaphore is not taken nor is already created
Writing to	C5, op2	give a semaphore
Reading from	C5, op2	returns zero
Writing to	C5, op3	sets the initial semaphore count value
Reading from	C5, op3	returns zero
Writing to	C5, op4	sets the maximum semaphore count value
Reading from	C5, op4	returns zero
Writing to	C5, op5	sets the blocking time that the thread wants to wait for a semaphore already taken
Reading from	C5, op5	returns zero
Writing to	C5, op6	creates a semaphore
Reading from	C5, op6	returns zero
Writing to	C5, op7	deletes a semaphore
Reading from	C5, op7	returns the next free semaphore slot, if 0xFFFFFFFF is returned, all semaphores slots are being used

Chapter 4

Task-Aware Interrupt Controller: Priority Space Unification

In the development of real-time systems, predictability is often hindered by technological factors which break the timing abstractions offered by RTOSes; namely, the priority space separation between threads and interrupts induces the rate-monotonic problem. Software approaches have tackled this issue, attempting to unify the priority space with varying degrees of success.

This chapter presents a hardware approach to the problem by unifying the priority space at the interrupt handling sub-system while the predictability is greatly enhanced with minimum software modifications. Our solution provides the interrupt controller with awareness of the currently running task's priority, making the solution independent of the used Operating System. We show how our approach is minimally intrusive at hardware architecture level and provides benefits beyond the capabilities of previous approaches. Our technique shows a 0.05% run-time overhead if no interrupts occur, and run-time reduction proportional to the interrupt rate for rates higher than 5 per second, for a interrupt workload around 0.07 mili-seconds. The work presented in this chapter was published in (Gomes et al., 2015a).

The remainder of this chapter is organized as follows: Section 4.1 introduces the identified problem and presents our solution and its contributions; Section 4.2 describes the architecture details of our solution; Section 4.3 shows the obtained results; Section 4.4 concludes this chapter and presents the future work.

4.1 Introduction

Real-Time Operating Systems' software flow is typically divided in prioritized threads, managed by a synchronous scheduling algorithm, and Interrupt Service Routines, asynchronously triggered by an interrupt controller. Furthermore, interrupts can affect the CPU at any time and therefore they have an inherently higher priority than threads. However, the priority of an interrupt is not always a synonym of a real-time operation. Interrupt handlers owned by low priority threads may preempt threads of any priority, hindering real-time behavior. This breaks the deterministic execution of a priority-based scheduler and it is known as rate-monotonic priority inversion (Scheler et al., 2009).

First attempts at overcoming the dual priority space caused by the distinct priorities of threads and Interrupt Service Routines (ISRs) can be found in (Kleiman and Eykholt, 1995). Kleiman and Eykholt tried to handle interrupts as threads by unifying both into a single synchronization model. Although they use a low-overhead technique to map interrupts into threads, they still allow an ISR to interrupt a higher priority thread running on the CPU although for a short period of time. In (Leyva-del Foyo et al., 2006, 2012), a low-level interrupt handling mechanism was implemented for both tasks and interrupts. This allows an ISR, when there is a higher priority thread running, to promptly resume the execution of the corresponding thread and block itself. This solution partially unifies the priority space, at the cost of increased ISR handling latency. Another approach (Zhang and West, 2006) is reducing the impact of blocking the thread execution and the latency of interrupt handling by dividing the interrupt service in two parts. The first should quickly complete all the essential interrupt handling while the second half can be delayed when there are hard real-time tasks to be performed. The second part will then be scheduled with a priority consistent with the remaining active threads on the system. They show how their solution provides significant improvements in terms of predictability; however, it fails to provide a truly unified priority space.

Scheler et al. (2009) propose the use of a coprocessor to eliminate the likely priority inversion; however, due to the need for increased synchronization, there is a trade-off between processing overhead and predictability. Taking advantage of commodity off-the-shelf hardware, Hofer et al. (2009) implemented the SLOTH system. SLOTH uses the interrupt system already available on hardware platforms to implement a unified priority space. In contrast to Kleiman and Eykholt

(1995), SLOTH implements threads as ISRs by mapping threads into interrupt handlers. Hence, the RTOS scheduler responsibilities are migrated to the interrupt controller where uniform priority can be given to both threads and interrupts. However, the system presents some limitations using other RTOSes: only static and unique priorities can be assigned to tasks, which cannot be created at run-time nor be blocked. Hofer et al. (2009) later overcame some limitations. Pinto et al. (2014) proved the SLOTH's portability by implementing the SLOTH concept on another CPU architecture using FreeRTOS, facing the same limitations.

Our solution provides the interrupt controller with awareness of the currently running task's priority. Thus, the interrupt controller is able to mask lower-priority interrupts. By tackling this issue at the interrupt controller level this solution continues to be valid for hybrid implementations where parts of the RTOS are implemented in both software and hardware. It is also independent of the Operating System, i.e, the RTOS APIs do not need to be modified. At the cost of little added hardware, our solution can overcome all limitations presented in the related work.

The main contribution of this work is the implementation of a Task-Aware Interrupt Controller (TAIC) to cope with Real-Time applications by unifying the priority space. Our concept system presents the following advantages:

1. A unified priority space, allowing an assignment of same-space priorities to both threads and interrupts.

2. It completely avoids the disturbance caused by interrupts triggered by lower-priority events and thus no execution time is consumed while a high-priority thread is running.

3. Interrupt overload can be avoided by adequately balancing the priorities of interrupts and threads, statically or at run-time.

4. This implementation solves the rate-monotonic priority inversion problem, independently of the processor architecture and Operating System.

4.2 Architecture Description

Our solution is implemented on an ARM-compliant softcore as described previously on Chapter 2, prototyped on a Virtex-5 FPGA, running FreeRTOS v7.4.0.

4.2.1 ARM Advanced Interrupt Controller

The implemented AIC provides up to 32 interrupt sources which can be configured using an 8-level priority encoder. It supports nested interrupts and integrates an interrupt vector table where the addresses of the corresponding ISRs are stored.

4.2.2 Task Management in FreeRTOS

FreeRTOS's workload is typically split in several threads in order to perform a certain task. Each thread has an assigned priority on which the scheduler will rely to manage the threads' execution flow. Each thread is represented by a designer-defined register-file (DDRF) structure. The DDRF stores the priority of the thread, the address of the stack's start address as well other parameters which are used for thread management. FreeRTOS's scheduler is in charge of deciding which task should be executing at a specific time. It can be configured to use a preemptive or a cooperative scheduling algorithm. The latter is commonly used in non-real-time applications, where each thread has processor time until completion of the workload. After completing its work, each thread will relinquish the execution and allow the processor time to be given to another task based on its priority. In the preemptive mode, mostly used in real-time applications, the scheduler is able to suspend a thread at each OS tick and coordinate which is the next ready thread to run. Each time a thread stops executing, whether suspended, delayed or relinquished, a context switch is performed. If there are more than one ready thread with the highest-priority, the scheduler will apply a round-robin scheme.

4.2.3 Interrupt Handling

When an interrupt occurs, the execution of the currently running thread is halted; the return address and the previous context are saved, and restored only when the ISR has finished executing. For the purpose of this work the Fast Interrupt Request (FIQ interrupt) is ignored. FIQ interrupt is usually used for high-priority applications where, for instance, fast IO communications are required. Therefore, when configured it is supposed to be used as an interrupt that must be attended with minimum latency, so the CPU should always suspend its work and attend the interrupt.

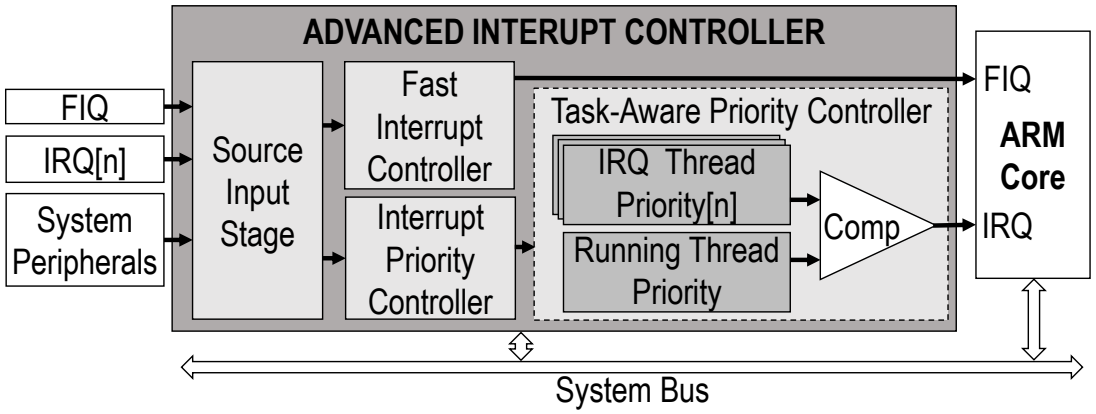


Figure 4.1: Task-Aware Interrupt Controller.

4.2.4 Task-Aware Interrupt Controller

The TAIC is a wrapper around ARM's AIC and differs from traditional Interrupt Controllers since it is supplied with information about the threads' priorities. This means that the interrupt controller is aware of the priority of code currently running on the CPU. Figure 4.1 illustrates a block diagram of the TAIC. For each interrupt source an extra 8-bit register, Interrupt-Thread Priority (`ITP`), is provided which holds the priority of the thread that configured the interrupt. Another 8-bit register, Running Thread Priority (`RTP`), which holds the priority of the currently running thread is also added to the interrupt controller. Therefore, when an interrupt occurs, TAIC compares the interrupt source priority recognized and forwarded by ARM's AIC against the priority of the running thread and decides whether to interrupt the CPU or not. Hence, the dual priority space is unified into a single priority scheme between threads and interrupts. The interface with these registers is explained as follows:

- When the `restore-context()` is issued, the FreeRTOS's `pxCurrentTCB` variable always points to the task control block (TCB) of the next thread. FreeRTOS uses `pxCurrentTCB` to retrieve the task's stack pointer. The corresponding thread's priority can be accessed from the memory position addressed by `pxCurrentTCB + 0x2C` which is where the priority of thread is stored.

- After getting the priority from the memory, the priority is stored in the `RTP` register of the TAIC. This register is memory mapped as any other AIC's registers and can be accessed using memory access instructions.

- After finishing the `restore-context()`, the CPU jumps to the thread's return

address and (re-)starts its execution. At this point, the TAIC is already aware of the priority of the currently running thread.

- Whenever a thread configures an interrupt, which encompasses configuring the interrupt mode and the handler before enabling it, the TAIC automatically copies the contents of the `RTP` register (which already holds the priority of the currently running thread) to the corresponding `ITP` register. Hence, the priority of the thread that configured the interrupt is saved.

- When an interrupt occurs, the TAIC compares the value of the corresponding `ITP` register with the `RTP` register: If the `ITP` register has a priority greater than or equal to the priority of the running thread, the interrupt is forwarded to the CPU which will handle it as usually; If the `ITP` register has a lower value than the priority of the running thread, the interrupt controller will not disturb the CPU until the priority of the running thread is decreased or another thread with lower priority starts the execution.

The `vTaskPrioritySet()` function sets the priority of any thread at run-time. Although this seems to imply an update of the thread priority in the `RTP` register, the update is not required because if a thread is setting its own priority down it means there might be other tasks of higher priority ready to execute and thus FreeRTOS forces a manual context-switch. This way, if there is no other threads to executed, the restore context of the same thread will occur and the `RTP` register will be updated as explained before during a `restore-context()`.

Table 4.1: Synthesis results obtained from Xilinx ISE 14.5.

Synthesis Results	Used (Utilization)			
	SoC (w/ AIC)	AIC	SoC (w/ TAIC)	TAIC
Slices	13306 (77%)	928 (5.4%)	13811 (79%)	1035 (6.0%)
Slice Registers	19233 (27%)	1948 (2.8%)	19497 (28%)	2212 (3.2%)
Slice LUTS	32064 (46%)	1999 (2.9%)	32158 (46%)	2092 (3.0%)
LUTRAM	3339 (18%)	0 (0%)	3339 (18%)	0 (0%)
Block RAM	4 (2%)	0 (0%)	4 (2%)	0 (0%)
Embedded DSPs	15 (23%)	0 (0%)	15 (23%)	0 (0%)
Clock Frequency	33MHz			

4.3 Results and Evaluation

Table 4.1 shows the synthesis results of the implemented ARM-based SoC, the AIC and the TAIC. Two different experiments were conducted using our ARM SoC running the FreeRTOS kernel. The experiment 4.3.1 validates the behavior of our single priority space. The experiment 4.3.2 measures the overhead of our implementation. For each experiment, tests were performed on the original system using the AIC, where a dual priority space exists, and on the unified priority space system, where the TAIC is used as Interrupt Controller. The hardware configuration for the following measurements are (1) CPU speed of 33 MHz, (2) caches disabled and (3) Periodic Interval Timer (PIT) timer frequency of 10 ms.

4.3.1 Behavior Evaluation

For the first experiment, we used the following system configuration:

- Four tasks, T1, T2, T3 and T4, each executing a benchmark application from the MiBench Suite (Guthaus et al., 2001), *basicmath_large*, *bitcount* (2000000 iterations), *qsort_large* and *stringsearch* respectively, all running with the highest priority on the system; thus, will run in a round-robin fashion. All threads start in a suspended state when the scheduler is started.
- A fifth task T5, with lower-priority (i.e., without real-time requirements), is configured to echo the data received through the serial port. After the scheduler is enabled, T5 will configure an ISR to signalize each time data is received in the serial input buffer. The ISR will inherit the priority of T5 which is lower than the others. The purpose of this interrupt is to analyze the impact of typical low-priority interrupts in the system.
- After configuring the ISR, T5 enables T1 which will start executing because of its higher priority. Then, each thread enables the next one sequentially.

For this experiment, the dual priority space problem exists, so whenever such interrupt occurs the execution of T1, T2, T3 or T4 is halted until the ISR handling is finished. The execution time of T1-4 reflects the time taken by the system to execute T1, T2, T3 and T4 plus the ISR handlers. Figure 4.2 depicts the execution time of the four threads for different interrupt rates. Using our TAIC, T1-4 are able to execute without disturbance from a low-priority ISR. The ISR is only attended

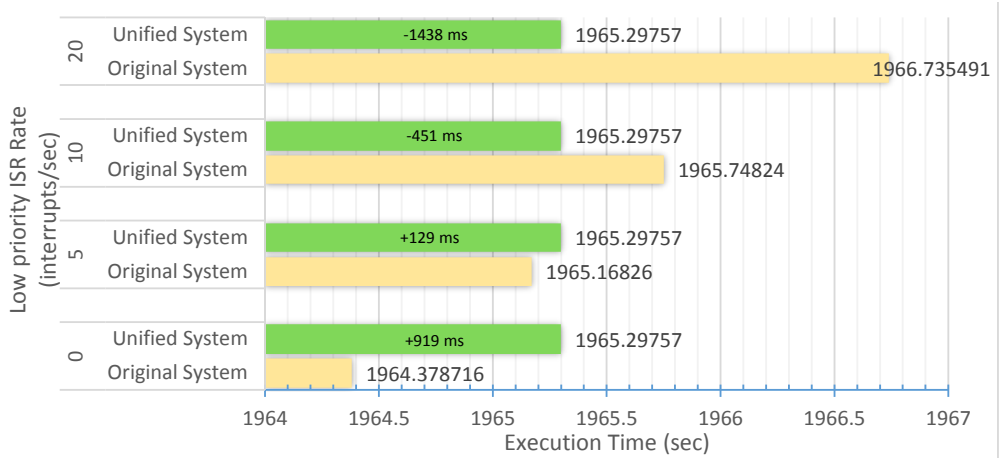


Figure 4.2: Execution time of T1+T2+T3+T4 with a low-priority interrupts ratio on the original vs unified priority systems.

at the end of T1-4's executions.

4.3.2 Overhead Evaluation

The only change required in the OS code to interface the TAIC was performed on the context-switch code, specifically on the `restore-context()` part. FreeRTOS has two sources which can trigger Context-Switch, whether by a periodic PIT interrupt or Software Interrupt (`SWI`) instruction used for manual context-switch. The context-switch code is divided in three main parts: (1) `save-context()` of the current thread, (2) find `highest-priority-thread()` to execute and (3) `restore-context()` of the thread. Moreover, for a PIT context-switch there is also the increment of the OS Tick. Our implementation adds four 32-bit instructions to the `restore-context()` code of both context-switch trigger sources resulting in a slight increase in memory footprint by 32 bytes. Table 4.1 highlights the minimal extra hardware required to support a unified priority space by hardware.

To measure the run-time overhead of our approach, we configured the Performance Monitor (PM) to measure the time of the original `restore-context()` code and the modified one (Figure 4.3) which interfaces with the Interrupt Controller. The PM peripheral was developed in order to allow measurements with cycle-accurate granularity of RTOS's APIs allowing analysis in terms of determinism, latency and jitter. The measurement was performed for the WCET which occurs when both caches are disabled. The original `restore-context()` code takes on average 738 ± 3 clock cycles (22.4 ± 0.08 μs) to execute while the modified one takes on

```
; Restore the context of the selected task.
; Set the LR to the task stack.
LDR     R1, =pxCurrentTCB
LDR     R0, [R1]
LDR     LR, [R0]
; Set the priority of the task in the Interrupt Controller
LDR     R2, [R0, #0x2C] ; get thread's priority
MOV     R3, #0xFFFFFF4C
BIC     R3, R3, #0xE00  ; set AIC_RTP address (0xFFFFF14C)
STR     R2, [R3]        ; store thread's priority
; The critical nesting depth is the first item on the stack.
; Load it into the ulCriticalNesting variable.
LDR     R0, =ulCriticalNesting
LDMFD   LR!, {R1}
STR     R1, [R0]
; Get the SPSR from the stack.
LDMFD   LR!, {R0}
MSR     SPSR_cxsf, R0
; Restore all system mode registers for the task.
LDMFD   LR, {R0-R14}^
NOP
; Restore the return address.
LDR     LR, [LR, #+60]
; And return - correcting the offset in the LR to obtain
; the correct address.
SUBS    PC, LR, #4
```

Figure 4.3: Restore context code - added instructions are highlighted.

average 851 ± 3 clock cycles (25.8 ± 0.09 μs) which shows an increase of 15% of run-time execution for each `restore-context()` performed by the FreeRTOS. On a PIT context-switch this only represents a context-switch overhead of 2%.

To obtain the impact of the modified `restore-context()` in a real application, we setup only T1-4 to execute with same priority, i.e., there is no ISRs occurring during the execution of T1-4. As shown in Figure 4.2, for an ISR rate equal to 0, the context-switch overhead has negligible impact in a real application when there is no low priority ISRs; there is only an increase of 0.047% (for almost 33 minutes of execution) in the time required to executed the tasks due to the extra four instructions on `restore-context()` code. Nevertheless, for certain conditions (e.g., different interrupt rates or interrupt workload), running T1-4 on the unified systems is even faster than running T1-4 with preemptions as shown in Figure 4.2. The execution time of the unified system, for different low-priority ISR rates, is always the same since no low-priority preemption occurs. However, the execution time of the original system varies with the ISR rate due to the ISR's latency. In fact, for an ISR rate greater than 10 the execution time of the original system becomes greater than the unified system. Hence, by solving the priority inversion, the performance of the real-time tasks is inherently improved, since we avoid blocking high-priority threads by low-priority events.

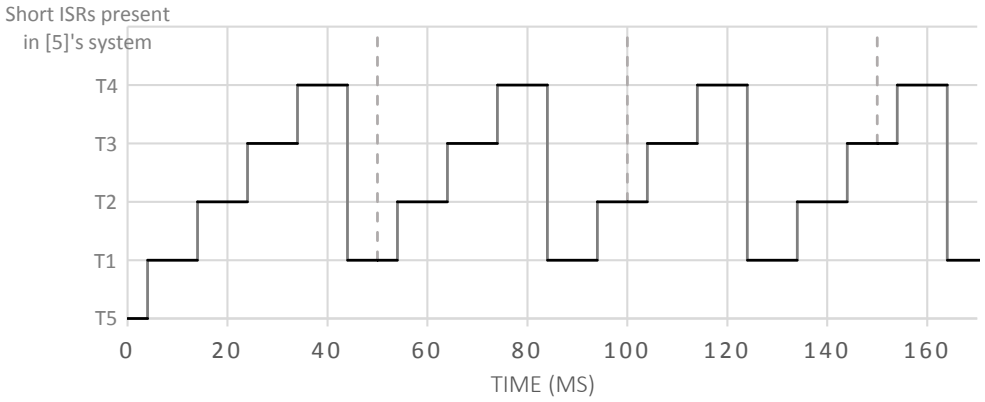


Figure 4.4: Unified priority space vs integrated priority space [5].

4.4 Conclusions

This chapter described micro-architectural enhancements to an Interrupt Controller to implement a unified priority space. Making the interrupt controller aware of the priority of what is being executed in the CPU, we successfully showed how a Task-Aware Interrupt Controller can be used to unify the priority of threads and interrupts. Therefore, we solve the rate-monotonic priority inversion issue by giving the Interrupt Controller the ability to only interrupt the CPU if the priority of the interrupt is greater or equal than the priority of the currently running thread. This solution is ideal for applications where only low-priority interrupts will occur. However, for applications where interrupts in a system are high priority, the user can decide not to configure the TAIC and thus avoid the associated overhead. Our solution presents negligible hardware and memory usage costs. Run-time varies, depending on interrupt-rate and interrupt workload; in our experiments, run-time was effectively decreased for interrupt rates higher than 5 per second. This approach is able to surpass related work by avoiding the limitations their solutions present to the rate-monotonic priority inversion problem. Compared with SLOTH approach our solution demands no engineering effort, i.e., the RTOS's APIs remain intact. Figure 4.4 depicts the behavior exhibited by our system and (Leyva-del Foyo et al., 2006); our system does not exhibit the periodic ISR entry observed in (Leyva-del Foyo et al., 2006); basically, our system behaves

Table 4.2: Context-switch cost of our approach using FreeRTOS vs approach used in Leyva-del Foyo et al. (2006) using other RTOSes. Caches are enabled.

	Hyperkernel 4.3	INTime 1.20	Windows CE.NET	QNX 6.1	FreeRTOS
Cost (%)	1.0	11.6	7.2	23.3	3.4

in the same way SLOTH would behave, without the inherent engineering effort limitations. The cost of our approach compared with (Leyva-del Foyo et al., 2006) is depicted on Table 4.2.

Although, in a real-time system, some peripheral device is dedicated most of the time to a specific thread according to the application design, future work will focus on allowing a peripheral interrupt to be shared by more than one thread. In doing so, TAIC will be extended with more resources to register thread *ids*, and priorities on shared interrupt lines as well as a mechanisms to de-multiplex shared interrupts. Moreover, some kind of message-centric approach will be followed to identify the thread destination as raw data does not apply in a multiplexed interrupt environment. Finally, future work will focus on improving the Context-Switch of RTOSes by implementing a new, scalable micro-architectural feature in our ARM SoC in order to perform the Context-Switch more quickly and more deterministically.

Chapter 5

Hardware Multithreading Applied to the Real-Time Domain

The emergence of hardware multithreading (HW-MT) architectures increased the performance of MT applications. However, traditional HW-MT architectures are not suitable for Real-Time Operating Systems as their performance-oriented scheduling algorithm may conflict with RTOS software scheduling.

This chapter presents **R**eal-**T**ime **S**ystem **H**ardware for **A**gnostic and **D**eterministic **O**Ses **W**ithin **S**oftcore (RT-SHADOWS), a portable architecture which provides a unified hardware-software scheduling, bringing the benefits of HW-MT to the RTOS domain. We show that tightly-coupled real-time compliant hardware integration achieves throughput benefits, maintaining the RTOS scheduling policy intact while increasing the predictability of RTOSes. Our solution shows, on average, speed-ups between 3 and 4 times over the native versions with very low area usage/performance overhead ratio, due to its minimal cost (2% of extra slices per hardware-supported thread). This work surpasses related work by providing a complete and agnostic hardware solution which is independent of any specific RTOS. The work presented in this chapter was published in (Gomes et al., 2015b).

The remainder of this chapter is organized as follows: Section 5.1 introduces the use of HW-MT support in real-time systems; Section 5.2 presents the problem associated with applying current HW-MT architectures on existent RTOS solutions; Section 5.3 describes RT-SHADOWS architecture. Section 5.4 presents the results and evaluation; Section 5.5 concludes the chapter and presents future work.

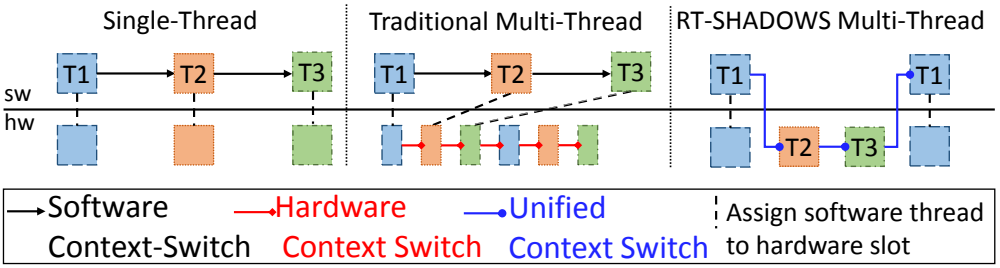


Figure 5.1: Scheduling on single-threaded, traditional multithreaded and RT-SHADOWS architectures.

5.1 Introduction

Real-Time Operating Systems (RTOSes) aid designers in simplifying and expediting the development of multithreaded applications by providing several APIs at the cost of performance overhead (Sheikh and Driscoll, 2011). Research towards alleviating this overhead by migrating different functionalities (mainly RTOS schedulers) into hardware has been performed (Naotaka et al., 2014; Labrosse, 2014; Oliveira et al., 2011). Usually, a hardware scheduler is implemented in FPGA-fabric and loosely-coupled to the processor using a commercially available bus (Naotaka et al., 2014; Labrosse, 2014; Bahri et al., 2012). Although these projects focus on hardware acceleration, there is no concern about portability to other RTOSes, which limits legacy software re-use. These projects fail to get the attention of the industrial community (Ong et al., 2013), either because these solutions do not cover a wide spectrum of RTOSes or are too complex and RTOS-dependent, which requires in-depth knowledge of the RTOS architecture from software developers. MAPUSOFT (Craig, 2015) is a software-based solution to provide agnosticism between applications and the OSes, while SEOS (Ong et al., 2013) is a hardware solution focusing on adaptability for other RTOSes; however, acceleration is only based on a hardware scheduler. To the best of the authors' knowledge, no research has applied hardware multithreading (HW-MT) to existent RTOSes, allowing legacy applications to benefit from shorter and deterministic context-switch and interrupt handling. Section 3.1 explains in more detail the scheduling algorithms used by current state of art HW-MT architectures. Figure 5.1 depicts the two levels of hierarchical scheduling found in traditional HW-MT architectures.

We present a HW-MT solution which provides throughput benefits, maintaining the RTOS scheduling policy intact and increasing the predictability of RTOSes. To accomplish this, we have developed RT-SHADOWS, a highly-portable multi-

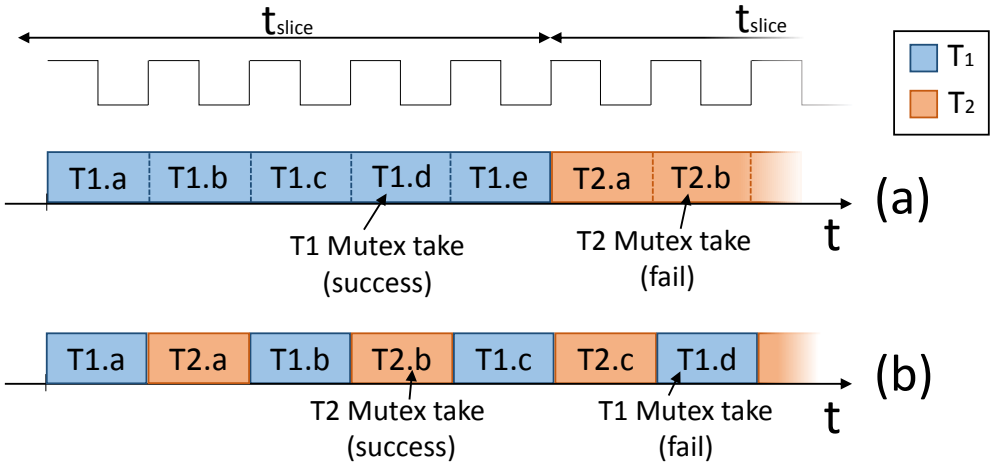


Figure 5.2: Scheduling conflict; (a) RTOS scheduling policy (b) IMT scheduling policy.

threaded softcore processor which offers hardware multithreading support portable across RTOSes, as well as higher performance predictable behavior. Our approach unifies RTOS scheduling and hardware-based thread scheduling, implementing a holistic multithreaded scheduling which leverages the advantages of HW-MT to the real-time world. By integrating the RTOS scheduler with the processor scheduler in a tightly-coupled fashion, the performance advantages of HW-MT are achieved without sacrificing (often improving) deterministic execution nor invalidating established static analysis methods.

The main contribution of this chapter is the implementation of a hardware multithreading architecture to cope with real-time applications. The main features of our system architecture are: (1) Unified HW/SW Multithreading Support; (2) Deterministic Tightly-Coupled Processor Scheduler and (3) Short and Deterministic Interrupt Handling (Gomes et al., 2015a); Chapter 6 will address with more detail two other features: (4) APIs Agnosticism and (5) High Portability.

5.2 Problem Description

Traditional HW-MT architectures apply their own scheduling algorithm (e.g., BMT, IMT or SMT) to manage threads execution flow in order to achieve maximum performance. Furthermore, RTOSes also apply their own scheduling algorithm which results in a hierarchical scheduling conflict as shown in Figure 5.2, which depicts an example where the IMT scheduling algorithm would change the expected behavior of an RTOS execution flow. Running 2 threads in round-robin

scheme (i.e., both have the same priority), it is expected that T1 starts executing first and takes the mutex (Figure 5.2 (a)). After the time-slice assigned to T1 is finished, T2 starts executing. As the mutex is already taken by T1, T2 will fail to take it. In Figure 5.2 (b) we are applying a round-robin scheme but using an IMT policy at hardware level (i.e., at each clock cycle a different thread is dispatched). With IMT scheduling, T2 may take the mutex before T1 (Figure 5.2 (b)) which would modify the expected execution flow. RT-SHADOWS takes advantages of HW-MT architectures using current RTOS solutions by unifying the two scheduling strategies.

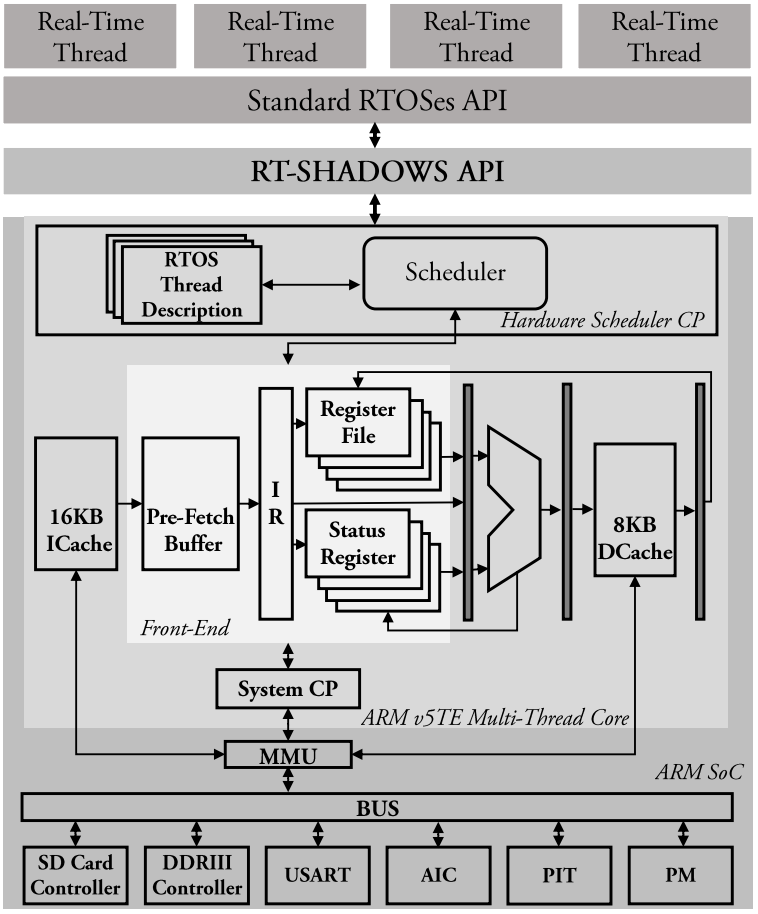


Figure 5.3: RT-SHADOWS top-level architecture.

5.3 RT-SHADOWS Architecture Description

An in-house ARMv5-compliant softcore was extended with new micro-architectural features to provide parameterisable, deterministic and agnostic HW-MT support. Figure 5.3 depicts RT-SHADOWS architecture. The number of hardware-supported

threads is configurable up to 128 threads, depending on the application demands. For area-constrained platforms, RT-SHADOWS allows the use of regular software threads if the number of application's threads is greater than the number of hardware-supported ones in order to ensure scalability. Also, the use of delay timers and synchronization methods (e.g., mutexes and semaphores) is optional. RT-SHADOWS offers a set of thread management and synchronization APIs commonly used in RTOSes. These are application-transparent, i.e., applications use the standard RTOSes APIs, wrapped into RT-SHADOWS APIs, in order to interface with the HW-MT support. In summary, only OS port-specific files are modified and no modifications are required on the OS kernel source, ensuring all the standard APIs remain intact.

5.3.1 Hardware Multithreading Support

Interrupt processing and RTOS services are the most important aspects that define RTOS performance. Each hardware-supported thread has its own registers, allowing short and deterministic switching of multiple contexts within the core. The ARM architecture supports multiple execution modes (e.g., Interrupt Request (IRQ), Supervisor (SVC), User (USR), etc) which different RTOSes can leverage. In order to speed-up exception handling, ARM uses banked registers for each mode. To support multiple RTOSes, our architecture enables banked registers mode to be software configurable; e.g., FreeRTOS's threads run in system (SYS) mode while μCOSII's threads run in SVC mode, with banked registers' mode matching RTOS's specification. In order to ensure a short and predictable interrupt response time, a hardware-supported thread is dedicated to the kernel. Hence, the RTOS interrupt latency overhead is decreased as no context of the currently running thread must be saved. Additionally, our architecture is able to solve the rate-monotonic priority inversion found in many RTOSes using our task-aware interrupt controller presented in (Gomes et al., 2015a).

5.3.2 Unified Scheduler

The unified processor scheduler is implemented as a tightly-coupled ARM coprocessor. In contrast to loosely-coupled schedulers, usually connected to a bus where several peripheral devices may compete for access, communication between the core and the coprocessor is performed using `MCR`/`MRC` instructions, ensuring a

short and deterministic communication link. This processor scheduler offers a high level of software configurability: (1) configurable scheduling algorithm (optional round-robin scheme); (2) configurable order of thread priorities (e.g., ascending or descending), enabling RTOSes to configure if low or high priority numbers denote low or high priority threads; and (3) ARM mode of the banked registers. A compact designer-defined register-file is used to store the thread's information such as its priority, current state, handler and stack pointer.

5.4 Results and Evaluation

To evaluate our solution in terms of performance and determinism two different experiments were conducted. Experiment 5.4.1 assesses APIs and shows the benefits of the hardware multithreaded extensions over the native RTOS execution. Experiment 5.4.2 runs Thread-Metric Benchmark Suite in order to evaluate how RT-SHADOWS alleviates RTOS overhead. Both experiments were validated on a Kintex-7 FPGA Embedded Kit (XC7K325T). Figure 5.4 shows the hardware cost of our approach. On our architecture, each extra hardware-supported thread requires 2 percent more chip space, compared to the single-thread version.

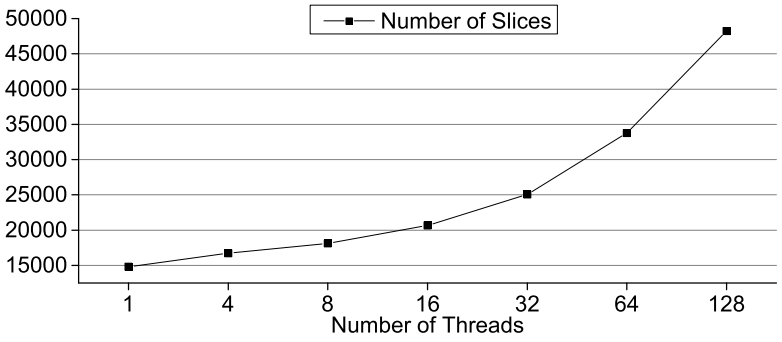


Figure 5.4: RT-SHADOWS hardware cost for a different number of hardware-supported threads over the single-threaded version.

5.4.1 API Evaluation

Several measurements were conducted in order to assess the minimum execution time (Min) and latency variance (Variation), of the most common APIs. Figure 5.5 presents the results on each architecture: (a) FreeRTOS native version; (b) FreeRTOS with multithreading extensions (c) μCOSII native version; and (d) μCOSII

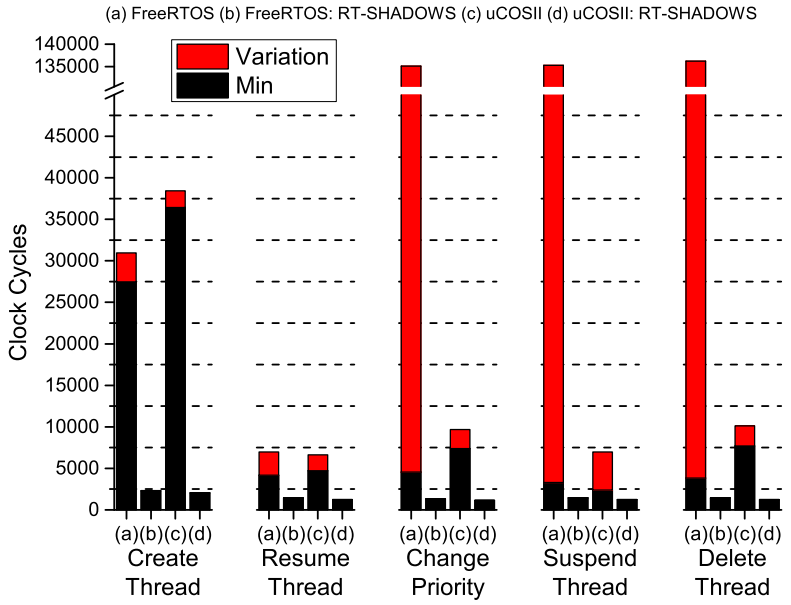


Figure 5.5: Comparison between the performance and jitter results in clock cycles for each architecture.

with multithreading extensions. A particular API may have different outcomes depending on multiple parameters such as the current threads' states or priorities. Hence, these results translate the values obtained from the variations of these different parameters: (1) number of threads; (2) thread's priority; (3) consecutive threads' priority gap and (4) whether the API triggers a context-switch. These variations encompassed several corner cases. As depicted, configurations with multithreading extensions outperform the native versions. Both performance and determinism are significantly increased. RT-SHADOWS reduces the overhead from 56% up to 98%. Also, RT-SHADOWS shows that the variation of the aforementioned parameters does not interfere with hardware-based APIs' execution time, i.e., the hardware latency is constant no matter how many threads are hardware-supported or created and their priorities. It is also noticeable that μCOSII natively

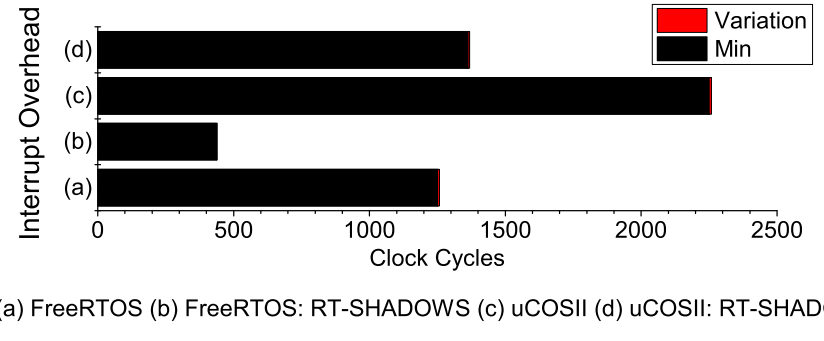


Figure 5.6: RTOS interrupt overhead in clock cycles for each architecture.

has better determinism than FreeRTOS. The dispatching of threads with a huge gap between their priorities is the main factor behind FreeRTOS indeterminism. Figure 5.6 depicts OS interrupt overhead of each configuration. This overhead is measured as the time between CPU interruption until the first instruction of the corresponding interrupt service routine is issued from memory (Sheikh and Driscoll, 2011). RT-SHADOWS is able to attend an interrupt request in shorter time than the native versions. The variation in Figure 5.6 is unnoticeable as all configurations present very low variation values (maximum 6 clock cycles).

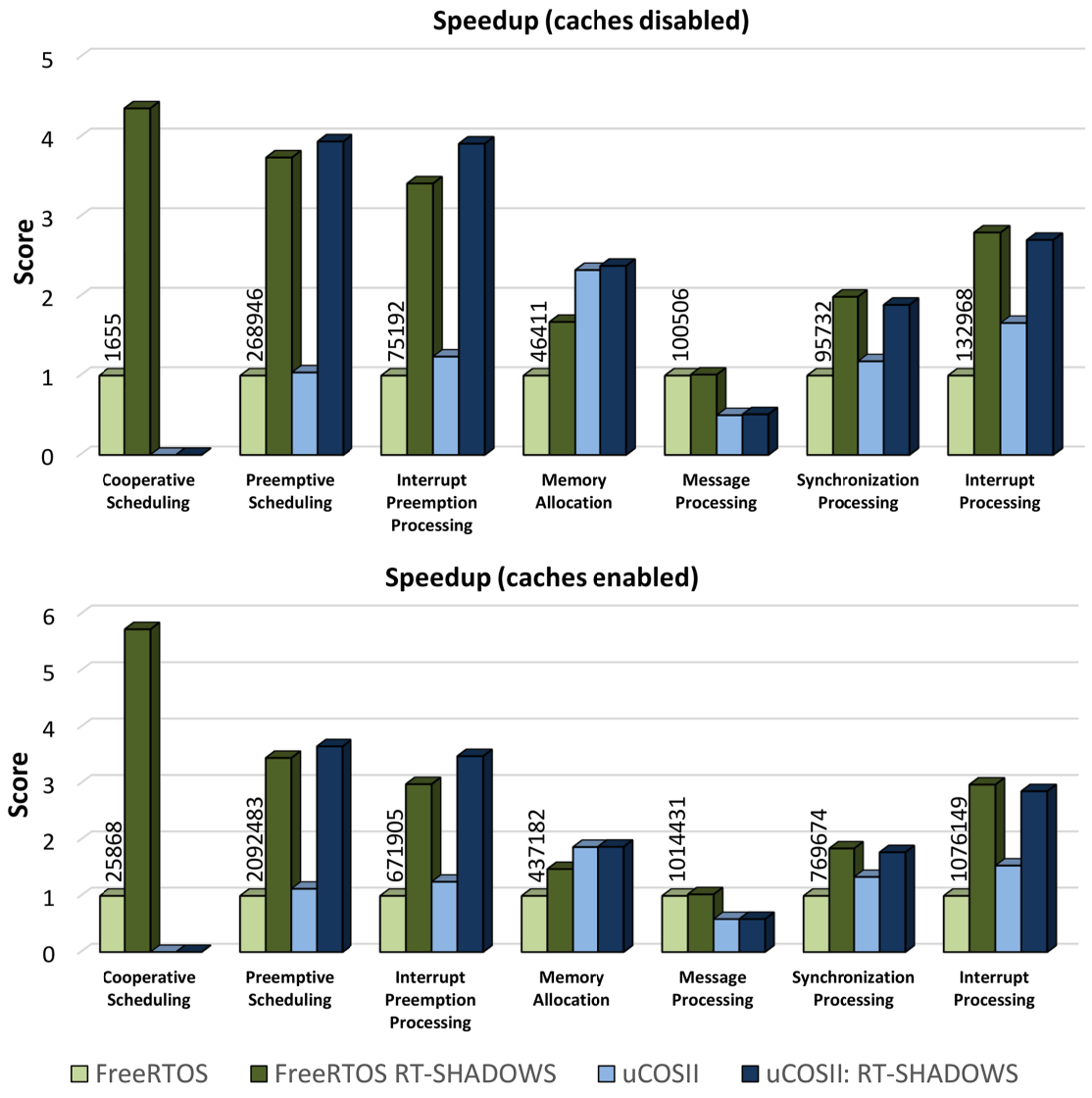


Figure 5.7: Speed-up results running Thread-Metric Benchmark with caches enabled.

5.4.2 Thread-Metric Evaluation

The Thread-Metric Benchmark Suite is a benchmark that measures RTOS real-time performance developed by Express Logic Inc. (Express Logic, 2015). The suite consists of 7 benchmarks, each evaluating interrupt processing and RTOS services. Each benchmark's score represents the RTOS impact on the running application, i.e., the greater the score the smaller the impact. These experiments were conducted with a clock frequency of 33 MHz, 10 ms periodic timer, with and without caches enabled and the IAR compiler with no optimization. We executed the benchmark on the FreeRTOS and μCOSII native versions and compared them over RT-SHADOWS architecture. Figure 5.7 shows how our architecture outperforms the native versions. Especially on benchmarks where context-switch and interrupt handling are exacerbated, RT-SHADOWS is able to show speed-ups between 3 and 4 times. On the memory and message specific benchmarks, RT-SHADOWS can still present speed-ups due to the gain obtained on the periodic context-switch. There is no results for μCOSII running the cooperative scheduling since this algorithm is not supported. Our system outperforms the native version with and without caches enabled.

5.5 Conclusions

This chapter described RT-SHADOWS, a co-designed hardware-software architecture which implements a holistic HW-MT solution, promoting configurability, determinism, performance and portability. We showed how such a holistic HW-MT approach can be applied to RTOSes solutions without the need for refactoring legacy-software. Our solution outperforms native solutions in terms of performance and determinism. RT-SHADOWS presents very low area usage/performance overhead ratio, due to its minimal cost (2% extra slices per hardware-supported thread). This work surpasses related work by providing a complete and agnostic hardware solution which is also RTOS-agnostic. Future work will encompass the development of new features and refactoring RT-SHADOWS to allow fine-grained configurations/customizations. Furthermore, future work will also focus on measuring the energy-efficiency of our approach. The ultimate goal will be to develop a profiling tool, that through a hardware-software co-design methodology, explores the migration of software threads to hardware according to the application and hardware platform demands and constraints.

Chapter 6

System Stack Agnosticism

The ability to build tailored processor systems exploiting softcores has become more realistic in applications that can be implemented on FPGAs. To ease this development process, several electronic design automation (EDA) tools are available. However, the efficiency of such tools can be limited by static processor architectures or architectures with insufficient customization capabilities hindering the development of tailored solutions (e.g., RTOS solutions) to tackle a particular metric. In order to fulfill such metrics, configurability of architectural features became an important aspect to be taken into account.

This chapter presents our agnostic system stack based on a co-designed hardware-software transparent solution which enables current RTOS solutions to benefit from hardware acceleration with null portability effort at software application and RTOS layer. Section 6.1 introduces the associated issue of hardware-based RTOS solutions regarding adaptability. Section 6.2 describes the agnostic stack and how transparency at the RTOS- and Application-level is ensured. Also, a feature diagram of the customizable hotspots (i.e., variability points) is presented. Section 6.3 concludes this chapter.

6.1 Introduction

The use of Real-Time Operating Systems is an established trend in most real-time embedded systems for resource management. However, RTOSes induce undesired overhead and latency into the system. Several researches proved that offloading

RTOSes to the hardware layer can bring significant performance and energy improvements. But these research outcomes failed to get the attention of industrial community as RTOS industry shows little interest in hardware-based RTOS solutions due to the high level of difficulty in adaptation process (Ong et al., 2013). Craig (2015) presented MAPUSOFT OS abstractor, a software-based approach that promotes re-use through OS agnosticism, allowing applications deployed on a specific OS to be easily moved to another OS. Ong et al. (2013) described SEOS, a hardware approach focusing on adaptability for RTOS which eases the hardware RTOS adaptation. However, MAPUSOFT and SEOS do not offer any hardware acceleration features, such as HW-MT and tightly-coupled hardware scheduler, limiting the maximum efficiency (in terms of performance, real-time execution and power) that can be obtained in systems where these features can be applied.

This work surpasses related work by providing a complete and agnostic system stack solution which is independent of any specific RTOS. Furthermore, as customization is a key to build tailored solutions, our solution leverages customization capabilities to build parameterisable hardware-based RTOS solutions depending on the application demands, e.g., custom number of hardware-supported threads, synchronization mechanisms, etc.

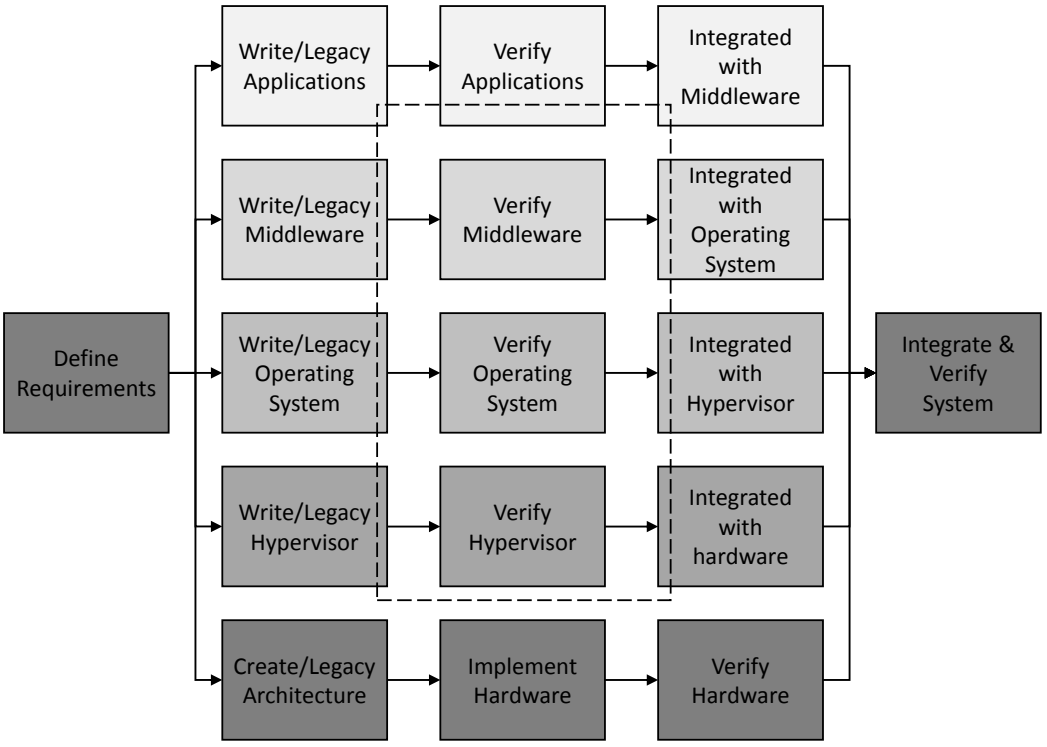


Figure 6.1: Our envisioned system stack design workflow; In this project, the middleware and virtualization agnosticism were not addressed.

6.2 Agnostic Software Stack

The system stack of a real-time embedded system is usually composed by three main layers: application layer composed by several real-time threads, operating system layer with real-time characteristics and the hardware platform. However, some middle- or high-end embedded systems can come with many more layers, as shown for instance in Figure 6.1. Furthermore, it is possible to have bare-metal stacks with only two layers (i.e., application and hardware layers). Our agnostic solution is able to introduce hardware acceleration without affecting the dependencies among the three main layers. We offer two levels of APP-OS-HW transparency. (1) The first one is between the application level and RTOS level. Applications are not aware of whether RTOS APIs are implemented in hardware or software. The standard RTOS APIs are wrapped in order to interface with the hardware support, allowing no changes in the OS kernel code. (2) The second one is between the RTOS and hardware levels. Figure 6.2 shows how our approach affect the three main layers. Our multithreaded hardware support is portable to different RTOSes, i.e., it is non-intrusive and independent of the RTOS, allowing only port-specific files of the RTOS to be automatically modified. This level of

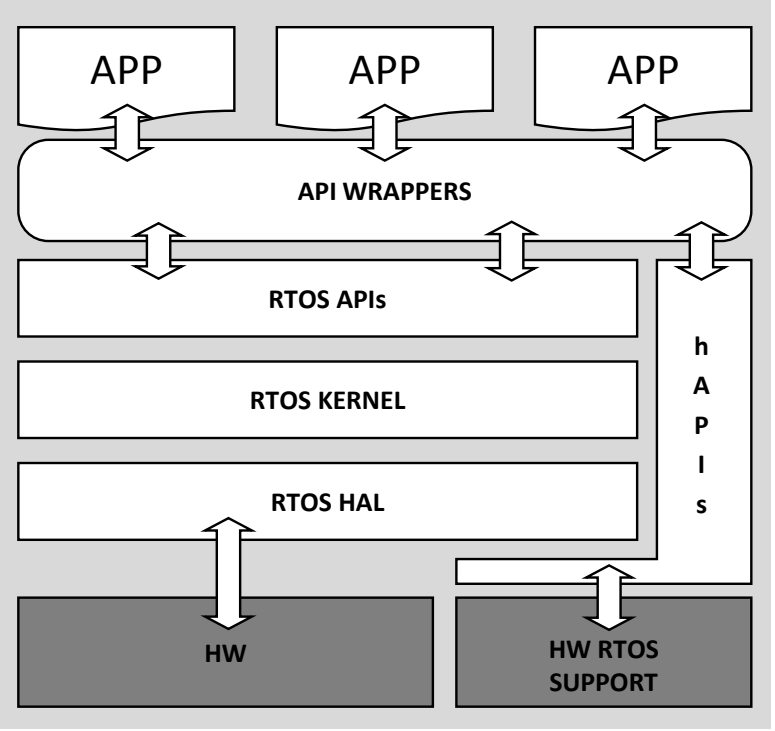


Figure 6.2: Agnostic system stack.

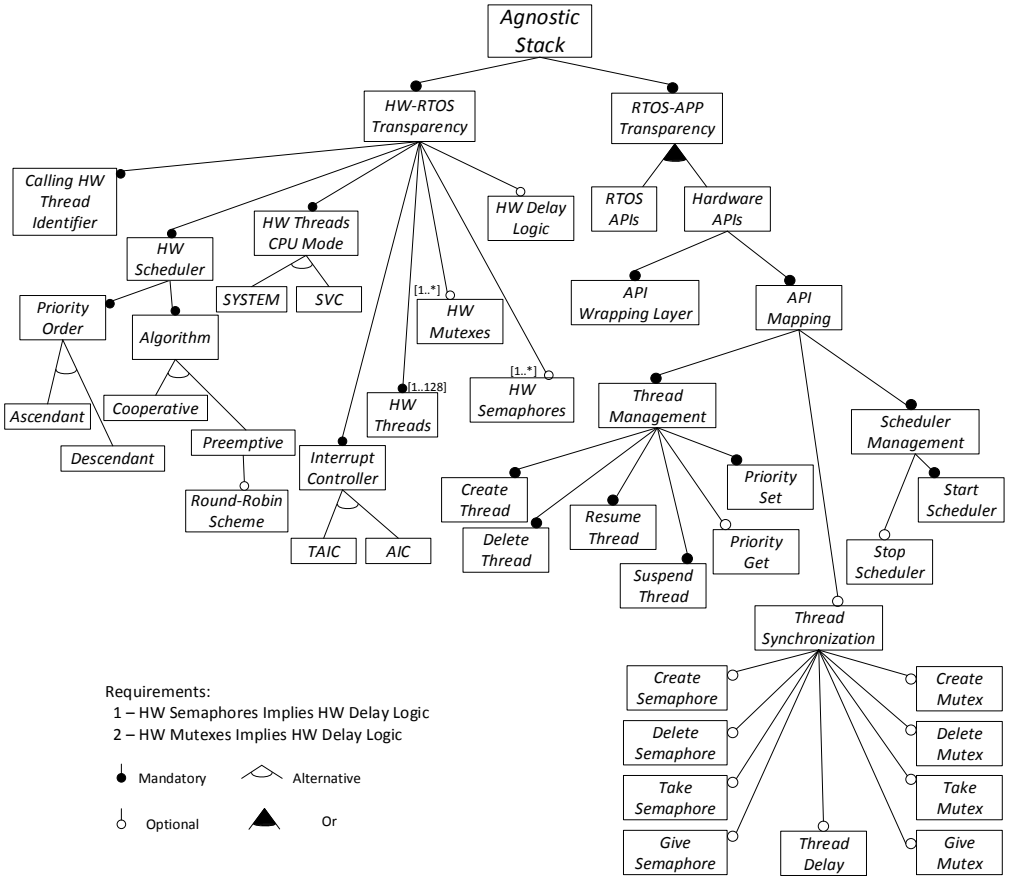


Figure 6.3: Agnostic stack feature diagram.

transparency allows legacy-software source code to run on our architecture without any direct modification from the programmer-side, with better performance and determinism. Furthermore, RTOS software developers can develop their RTOS applications without in-depth knowledge of the hardware, following the RTOS API specification.

To ensure this agnosticism, our hardware support is configurable in order to be able to match different RTOS specifications. Figure 6.3 depicts a feature diagram of our architecture, describing the customization hotspots. Our architecture provides two levels of agnosticism: (1) between the hardware and the RTOS and (2) between the RTOS and the application.

HW-RTOS Transparency

- **HW Threads:** identifies the number of hardware-supported threads.
- **HW Delay Logic:** optional feature to support thread delay/sleep by hardware.

- **HW Mutexes:** optional feature which identifies the number of hardware-supported mutexes if desired. This feature implies the use of HW Delay Logic.
- **HW Semaphores:** optional feature which identifies the number of hardware-supported semaphores if desired. This feature implies the use of HW Delay Logic.
- **Interrupt Controller:** selects which interrupt controller to be used. The default ARM AIC or the TAIC presented in Chapter 4.
- **HW Threads CPU Mode:** sets the the CPU mode of replicated register file for each thread.
- **HW Scheduler:** sets the thread's priority order either ascendant or descendant. It also sets the HW scheduler algorithm as cooperative or preemptive with optional round-robin scheme.
- **Calling HW Thread Identifier:** Some RTOSes allow the calling thread to use a specific argument to identify itself through an API parameter. On FreeRTOS, passing `NULL` as the argument for the thread handler API parameter will identify the calling thread while in μCOSII the equivalent identifier is given by the macro `OS_PRIO_SELF`.

RTOS-APP Transparency

- **RTOS APIs:** Set of the standard RTOS depending on the RTOS used.
- **Hardware APIs:** Set of hardware-based APIs specific for the RTOS used.

The following XMLized snippets represent metadata models for both transparency levels. These metadata models leverage the automation of system stack design workflow and ease the porting of new RTOSes to our proposed solution. Source-level code patching, implemented as XSLT transformer and Perl scripts, has been experimented to be later embedded into the elaboration time according to our envisioned system stack design workflow. Specifically at the hardware level, during the elaboration time, verilog directives (e.g., `` `define `` or `parameter`) are generated to control conditionalities (e.g., hAPIs) defined with `` `ifdef... `` `` `endif `` directives as well as general RTL parameters (e.g., `parameter data_width = 32;`). In so doing, all customizable hotspots will be automatically configured based on inputs provided by both system (system software and hardware) and applications designers.

HW-RTOS Transparency XML:

```
<feature name="HW-RTOS Transparency" type="Mandatory">
  <feature name="HW Scheduler" type="Mandatory">
    <feature name="Priority Order" type="Mandatory">
      <featureGroup type="Alternative">
        <feature name="Ascendant"/>
        <feature name="Descendant"/>
      </featureGroup>
    </feature>
    <feature name="Algorithm" type="Mandatory">
      <featureGroup type="Alternative">
        <feature name="Cooperative"/>
        <feature name="Preemptive">
          <feature name="Round-Robin Scheme" type="Optional"/>
        </feature>
      </featureGroup>
    </feature>
  </feature>
  <feature name="HW Threads CPU Mode" type="Mandatory">
    <featureGroup type="Alternative">
      <feature name="SYSTEM"/>
      <feature name="SVC"/>
    </featureGroup>
  </feature>
  <feature name="Interrupt Controller" type="Mandatory">
    <featureGroup type="Alternative">
      <feature name="TAIC" type="Mandatory"/>
      <feature name="AIC" type="Mandatory"/>
    </featureGroup>
  </feature>

  <feature min="1" max="*" name="HW Threads" type="Mandatory"/>
  <feature min="0" max="*" name="HW Delay Logic" type="Optional"/>
  <feature min="0" max="*" name="HW Semaphores" type="Optional"/>
  <feature min="0" max="*" name="HW Mutexes" type="Optional"/>
</feature>
```

RTOS-APP Transparency XML:

```
<feature name="RTOS-APP Transparency" type="Mandatory">
  <featureGroup type="Or">
    <feature name="RTOS APIs"/>
    <feature name="Hardware APIs">
      <feature name="API Wrapping Layer" type="Mandatory"/>
      <feature name="API Mapping" type="Mandatory">
        <xi:include href="API_Mapping.xml"/>
      </feature>
    </feature>
  </featureGroup>
</feature>
```

API Mapping XML:

```
<feature name="API Mapping" type="Mandatory">
  <feature name="Thread Management" type="Mandatory">
    <feature name="HW_Create_Thread" type="Mandatory"/>
    <feature name="HW_Delete_Thread" type="Mandatory"/>
    <feature name="HW_Resume_Thread" type="Mandatory"/>
    <feature name="HW_Suspend_Thread" type="Mandatory"/>
    <feature name="HW_Priority_Set" type="Mandatory"/>
    <feature name="HW_Priority_Get" type="Optional"/>
  </feature>
  <feature name="Thread Synchronization" type="Optional">
    <feature name="HW_Thread_Delay" type="Optional"/>
    <feature name="HW_Create_Mutex" type="Optional"/>
    <feature name="HW_Delete_Mutex" type="Optional"/>
    <feature name="HW_Take_Mutex" type="Optional"/>
    <feature name="HW_Give_Mutex" type="Optional"/>
    <feature name="HW_Create_Semaphore" type="Optional"/>
    <feature name="HW_Delete_Semaphore" type="Optional"/>
    <feature name="HW_Take_Semaphore" type="Optional"/>
    <feature name="HW_Give_Semaphore" type="Optional"/>
  </feature>
  <feature name="Scheduler Management" type="Optional">
    <feature name="HW_Start_Scheduler" type="Mandatory"/>
    <feature name="HW_Stop_Scheduler" type="Optional"/>
  </feature>
</feature>
```

Example of FreeRTOS API Mapping XML:

```
<feature name="FreeRTOS API Mapping" type="Mandatory">
  <feature name="Thread Management" type="Mandatory">
    <feature HW_Create_Thread="xTaskGenericCreate"/>
    <feature HW_Delete_Thread="vTaskDelete"/>
    <feature HW_Resume_Thread="vTaskResume"/>
    <feature HW_Suspend_Thread="vTaskSuspend"/>
    <feature HW_Priority_Set="vTaskPrioritySet"/>
    <feature HW_Priority_Get="uxTaskPriorityGet"/>
  </feature>
  <feature name="Thread Synchronization">
    <feature HW_Thread_Delay="vTaskDelay"/>
    <feature HW_Create_Mutex="xSemaphoreCreateMutex"/>
    <feature HW_Delete_Mutex="vSemaphoreDelete"/>
    <feature HW_Take_Mutex="xSemaphoreTake"/>
    <feature HW_Give_Mutex="xSemaphoreGive"/>
    <feature HW_Create_Semaphore="xSemaphoreCreateCounting"/>
    <feature HW_Delete_Semaphore="vSemaphoreDelete"/>
    <feature HW_Take_Semaphore="xSemaphoreTake"/>
    <feature HW_Give_Semaphore="xSemaphoreGive"/>
  </feature>
  <feature name="Scheduler Management">
    <feature HW_Start_Scheduler="vTaskStartScheduler"/>
    <feature HW_Stop_Scheduler="vTaskEndScheduler"/>
```

```
  </feature>
</feature>
```

Example of μCOSII API Mapping XML:

```
<feature name="uCOSII API Mapping" type="Mandatory">
  <feature name="Thread Management" type="Mandatory">
    <feature HW_Create_Thread="OSTaskCreateExt"/>
    <feature HW_Delete_Thread="OSTaskDel"/>
    <feature HW_Resume_Thread="OSTaskResume"/>
    <feature HW_Suspend_Thread="OSTaskSuspend"/>
    <feature HW_Priority_Set="OSTaskChangePrio"/>
  </feature>
  <feature name="Thread Synchronization">
    <feature HW_Thread_Delay="OSTimeDly"/>
    <feature HW_Create_Mutex="OSMutexCreate"/>
    <feature HW_Delete_Mutex="OSMutexDel"/>
    <feature HW_Take_Mutex="OSMutexPend"/>
    <feature HW_Give_Mutex="OSMutexPost"/>
    <feature HW_Create_Semaphore="OSSemCreate"/>
    <feature HW_Delete_Semaphore="OSSemDel"/>
    <feature HW_Take_Semaphore="OSSemPend"/>
    <feature HW_Give_Semaphore="OSSemPost"/>
  </feature>
  <feature name="Scheduler Management">
    <feature HW_Start_Scheduler="OSStart"/>
  </feature>
</feature>
```

6.3 Conclusions

The opportunity to develop an agnostic hardware-support RTOS solution allows the acceleration of legacy applications, previously running in software-only RTOS solutions, with a seamless transition from one platform to another. This chapter presented our configurable architecture and its configurable hotspots that can be used to create different solutions depending on the application requirements. Also, representing the customization hotspots as XMLized models provides higher flexibility and customizability for later re-use as well as enabling system stack design and integration automation. Source-level code patching, implemented as XSLT transformer and Perl scripts, has been experimented to be later embedded into the elaboration time according to our envisioned system stack design workflow. Future work will encompass more research into code transformers to allow seamless integration and generation of the system stack architecture during the elaboration time of our envisioned design workflow.

Chapter 7

System Integration

This chapter describes the integration of the whole system stack with the hardware multithreading support. We present our co-designed architecture and its impact on memory footprint. Section 7.1 introduces the integration of all sub-systems and how software communicates with the hardware support. Section 7.2 presents the impact of our approach on memory footprint. Section 7.3 concludes this chapter.

7.1 Introduction

In order to benefit from hardware acceleration without being intrusive to the software layer we present a co-designed hardware-software transparent solution. Figure 7.1 depicts the complete system stack encompassing a configuration tool responsible for specifying the optimal software-hardware solution for a particular application, i.e., number of hardware-supported threads depending on available area, hardware synchronization support, etc. Although the configuration tool is still under development, all the software and hardware platform are parameterisable allowing easy custom configurations to be made by a configuration tool.

7.1.1 System Boot

We have developed our own bootloader code to initialize our ARM softcore processor. The startup sequence is very dependent on our ARM architecture and memory mapping, therefore it is implemented in assembly. We followed a simple

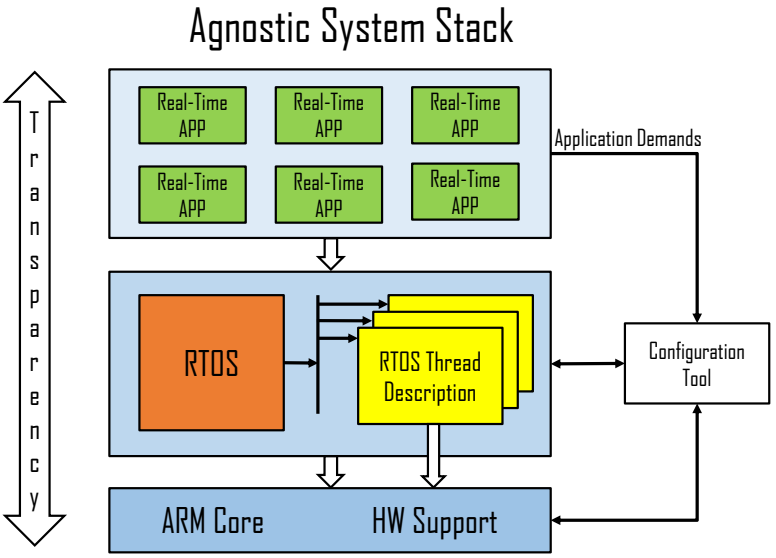


Figure 7.1: Complete agnostic system stack.

layout by locating a non-volatile memory (SD-Card) at address 0x0 and volatile memory (DDR RAM) upwards. The memory mapping can be accomplished by performing the remap command, allowing RAM, which is normally faster, to be located at address 0x0 to speed up the handling of processor exceptions and interrupts through the vector table. Figure 7.2 depicts the memory layout before and after the remap command. Our bootloader performs the following steps:

1. Enable instruction cache to speedup the kernel code copying process.
2. Initialize USART peripheral for debug purposes.
3. Copy kernel code from SD-CARD to RAM.

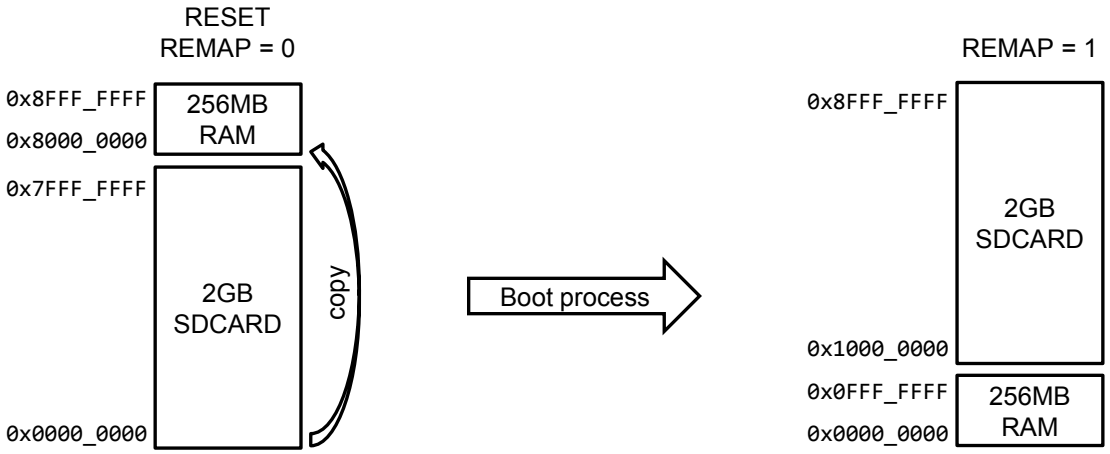


Figure 7.2: Memory remap feature on booting.

4. Disable interrupts (FIQ and IRQ) and enable supervisor CPU mode.

5. Disable MMU and both caches followed by flushing both of them.

6. Perform remap command using the following assembly code snippet:

```
1  ; remap memory
2  MOV R0, #0xFFFFFF50
3  BIC R0, R0, #0x200 ; r0 = 0xFFFFFD50
4  STR R0, [R0] ; to enable remap, perform a dummy store to addr 0xFFFFFD50
5  MOV PC, #0x0 ; Call kernel: jump to addr 0x0 which is now the first memory address of RAM
```

7. Jump to address 0x0 which is now on RAM and start the execution of the kernel.

Figure 7.3 depicts how the remap is performed on our architecture. Our pipelined processor implementation assures that the instruction `MOV PC, #0x0` will be read from SD-CARD memory before the remap command is executed. When such instruction is decoded and executed, all pipeline will be flushed as it performs an unconditional jump. As the remap was already performed at this point, the instruction at address 0x0 will be fetched from RAM memory.

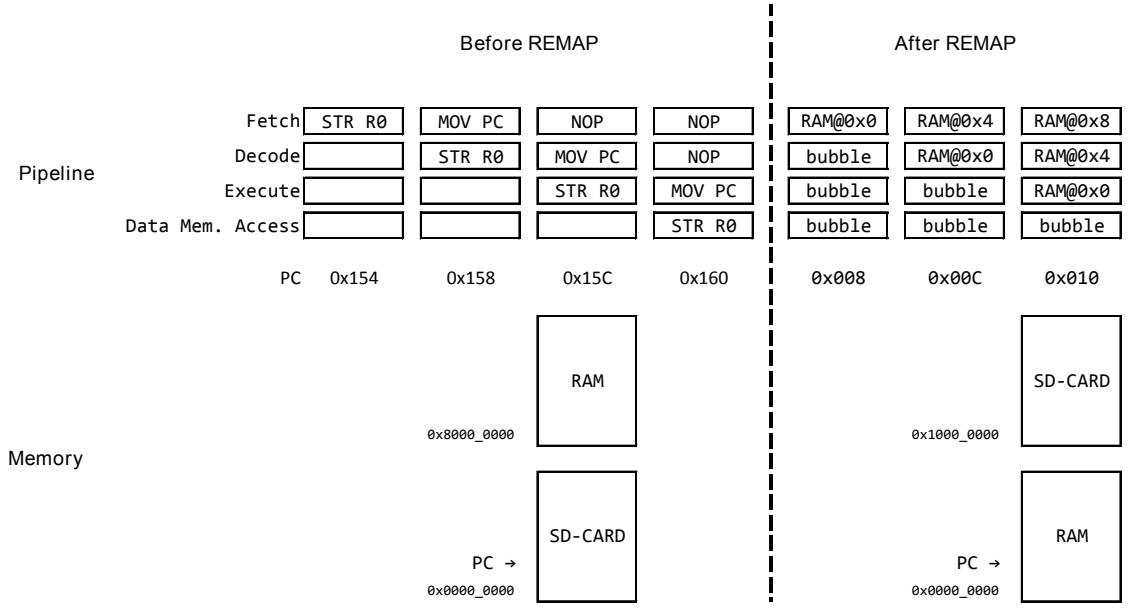


Figure 7.3: Pipelined processor during memory remap feature on booting.

7.1.2 RTOS-level Transparency

Upon RTOS boot, several peripheral devices are configured such as the AIC, PIT, etc. At this point, the RTOS specificities must be informed to the hardware multithreading support. The following features must be initialized before the RTOS starts executing any thread code:

- The behavior of the `LDM` and `STM` instructions, as explained in Section 3.3;
- The register mode of the hardware-supported threads, as explained in Section 3.1;
- How does the RTOS identify a caller thread without knowing its *id*, as explained in Section 3.2;
- Set how does low or high priority values denote lower or higher priorities, as explained in Section 3.2;

The following assembly code snippets exemplify how FreeRTOS and μCOSII initialize all of the aforementioned features. The former was included into the FreeRTOS' `LowLevelInit()` located in the *board_lowlevel.c* file:

```
1  asm volatile ("PUSH {R0}");
2  asm volatile ("MOV  R0, #1");
3  // change the behaviour of LDM and STM
4  asm volatile ("MCR  p14, 0, R0, c3, c0, 0");
5  // freeRTOS threads run in SYSTEM mode
6  asm volatile ("MOV R0, #0x1F");
7  // set the mode of the replicated registers (i.e., multithreading registers)
8  asm volatile ("MCR  p14, 0, R0, c3, c0, 1");
9  // FreeRTOS uses NULL parameter to identify the running/caller thread
10 asm volatile ("MOV  R0, #0");
11 //set how the hw identify the running/caller thread without knowing its ID
12 asm volatile ("MCR  p14, 0, R0, c3, c0, 2");
13 // sets the order of priorities;
14 asm volatile ("MOV  R0, #1");
15 //"0" -> means that priority 0 is the highest
16 //"1" -> means that priority 0 is the lowest priority
17 asm volatile ("MCR  p14, 0, R0, c3, c0, 3"); // sets the priority order
18 asm volatile ("POP  {R0}");
```

The latter is the equivalent code snippet for μCOSII and it was included in the `BSP_Init()` function located in the *bsp.c* file:

```
1  asm volatile ("PUSH  {R0}");
2  asm volatile ("MOV   R0, #1");
```

```
3  // change the behaviour of LDM and STM
4  asm volatile ("MCR  p14, 0, R0, c3, c0, 0");
5  // ucOSII threads run in SVC mode
6  asm volatile ("MOV  R0, #0x13");
7  // set the mode of the replicated registers (i.e., multithreading registers)
8  asm volatile ("MCR  p14, 0, R0, c3, c0, 1");
9  // ucOSII uses OS_PRIO_SELF (0xFF) parameter to identify the running/caller thread
10 asm volatile ("MOV  R0, #0xFF");
11 //set how the hw identify the running/caller thread without knowing its ID
12 asm volatile ("MCR  p14, 0, R0, c3, c0, 2");
13 // sets the order of priorities;
14 asm volatile ("MOV  R0, #0");
15 //"0" -> means that priority 0 is the highest
16 //"1" -> means that priority 0 is the lowest priority
17 asm volatile ("MCR  p14, 0, R0, c3, c0, 3"); // sets the priority order
18 asm volatile ("POP  {R0}");
```

7.1.3 Application-level Transparency

In order to maintain API transparency each standard RTOS API is wrapped into a hardware-based API. Each RTOS has its own set of hardware-based APIs. Although they are very similar, since they all use the same hardware, there are minor differences between hAPIs for different RTOSes (e.g., in terms of each API function prototype or signature).

These are the generic hAPIs; When porting this hAPIs for each RTOS, the header must be adapted to match to the corresponding standard RTOS API signature or prototype.

Create Thread: Creates a new thread. It checks for a free hardware slot for the new thread (line 6), if it fails the equivalent standard RTOS API should be called and a regular software thread will be created (line 10). If a hardware-supported thread is created, then the new thread's stack is initialized (line 16-26), and the thread's hardware features are configured, such as thread priority, thread *id* or handler, initial thread state, thread parameters and thread stack pointer (line 28-41). Lastly and as a scheduling point is reached, it verifies if a context-switch is required since the new thread could currently be the highest priority thread ready to run and should run immediately (line 48).

```
1  unsigned int HW_Create_Thread( void* thread_code, void *thread_parameters, unsigned int
       thread_priority, unsigned int *created_thread)
2  {
```

```
3   unsigned int thread_id;
4   unsigned int scheduler_running;
5   // check if there are hardware-supported threads available
6   asm volatile ("MRC  p14, 0, %[value], c2, c0, 6": [value]"+r" (thread_id));
7   // if thread_id is equal to zero all hardware-supported threads are full
8   if(thread_id == 0)
9   { // Create regular software thread
10    /* call RTOS API here */
11    return 0;
12  }
13  else
14  { // return thread_id to application
15    *created_thread = thread_id;
16    // top of the stack
17    unsigned int original_top_of_stack = (unsigned int)&ThreadStack[thread_id][
          THREAD_STACK_SIZE-1];
18    unsigned int *top_of_stack = (unsigned int *) original_top_of_stack;
19    // place the return address on the stack, offset is added since returning from ISR
          we perform SUB pc, lr, #4
20    *top_of_stack = ( void * ) thread_code + 0x4;
21    top_of_stack--;
22    // used later to load the stack value to R13 register of the thread being created
23    *top_of_stack = original_top_of_stack;
24    top_of_stack--;
25    // used later to load the parameter to R0 register of the thread being created
26    *top_of_stack = (int)thread_parameters;
27    ENTER_CRITICAL();
28    // opcode = 0: init the process of creating hardware-supported thread
29    asm volatile ("MCR  p14, 0, R0, c0, c0, 0");
30    // opcode = 1: set the new hardware-supported thread's priority
31    asm volatile ("MCR  p14, 0, %[value], c0, c0, 1" : [value]"+r" (thread_priority));
32    // opcode = 2: set the new hardware-supported thread's handler
33    asm volatile ("MCR  p14, 0, %[value], c0, c0, 2": [value]"+r" (*created_thread ));
34    // opcode = 3: set the new hardware-supported thread's state
35    asm volatile ("PUSH  {R0}");
36    asm volatile ("MOV  R0, #4");
37    asm volatile ("MCR  p14, 0, R0, c0, c0, 3");
38    // load thread's R0 and R13 registers
39    asm volatile ("MOV  R0, %[value]": [value]"+r" (top_of_stack));
40    asm volatile ("LDMFD R0, {R0,R13}^");
41    asm volatile ("POP  {R0}");
42    *top_of_stack++;            // remove the entry for the R0 register
43    *top_of_stack++;            // remove the entry for the R13 register
44    // opcode = 5: set the new hardware-supported thread's current top of stack
45    asm volatile ("MCR  p14, 0, %[value], c0, c0, 5": [value]"+r" (top_of_stack));
46    // opcode = 7: initialize a free hardware slot with a new hardware-supported thread
47    asm volatile ("MCR  p14, 0, R0, c0, c0, 7");
48    // check if scheduler is running
49    asm volatile ("MRC  p14, 0, %[value], c2, c0, 7": [value]"+r" (scheduler_running));
50    if( scheduler_running )
51    {
52      EXIT_CRITICAL();   // if scheduler is running, trigger a context switch
53      asm volatile ("SWI 0");  // to find the thread with highest priority
54    }
55    else
56      EXIT_CRITICAL();
```

```
57        return 1;
58      }
59    }
```

Delete Thread: Deletes a thread. It checks if the thread to be deleted is a software or a hardware-supported one (line 6-10). If hardware-supported, it selects the thread and change its state to *deleted* and its hardware slot becomes available for a new thread; then, it analyzes this scheduling point by selecting the next higher priority thread ready to run (line 13-26). If software-supported, the equivalent standard RTOS API should be called instead (line 31).

```
1   void HW_Delete_Thread( unsigned int thread_to_delete )
2   {
3     unsigned int   HW_thread_id;
4     unsigned int   scheduler_running;
5     ENTER_CRITICAL();
6     // check if thread is a hardware-supported one
7     asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (thread_to_delete));
8     // if a thread is found on hardware, HW_thread_id will have the thread handler
9     asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
10    if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
11    { // thread is a hardware_supported one
12      // set the thread state as deleted
13      asm volatile ("PUSH {R0}");
14      asm volatile ("MOV  R0, #0x00");
15      asm volatile ("MCR  p14, 0, R0, c1, c0, 2");
16      asm volatile ("POP  {R0}");
17      // check if scheduler is running
18      asm volatile ("MRC  p14, 0, %[value], c2, c0, 7": [value]"+r" (scheduler_running));
19      if( scheduler_running )
20      {
21        EXIT_CRITICAL();   // trigger a context switch
22        asm volatile ("SWI 0");  // to find the thread with highest priority
23      }
24      else
25        EXIT_CRITICAL();
26    }
27    else
28    { // thread is not a hardware_supported one
29      EXIT_CRITICAL();
30      // delete a software thread
31      /* call RTOS API here */
32    }
33  }
```

Resume Thread: Resumes a thread. It checks if the thread to be resumed is a software or a hardware-supported one (line 6-10). If hardware-supported, it selects the thread and change its state to *ready* and then, it analyzes this new

scheduling point (line 13-26). If software-supported, the equivalent standard RTOS API should be called instead (line 31).

```
1  void HW_Resume_Thread( unsigned int thread_to_resume )
2  {
3    unsigned int   HW_thread_id;
4    unsigned int   scheduler_running;
5    ENTER_CRITICAL();
6    // check if thread is a hardware-supported one
7    asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (thread_to_resume));
8    // if a thread is found on hardware, HW_thread_id will have the thread handler
9    asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
10   if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
11   { // thread is a hardware_supported one
12     // set the thread state as ready
13     asm volatile ("PUSH {R0}");
14     asm volatile ("MOV  R0, #0x04");
15     asm volatile ("MCR  p14, 0, R0, c1, c0, 2");
16     asm volatile ("POP  {R0}")
17     // check if scheduler is running
18     asm volatile ("MRC  p14, 0, %[value], c2, c0, 7": [value]"+r" (scheduler_running));
19     if( scheduler_running )
20     {
21       EXIT_CRITICAL();    // trigger a context switch
22       asm volatile ("SWI 0");   // to find the thread with highest priority
23     }
24     else
25       EXIT_CRITICAL();
26   }
27   else
28   { // thread is not a hardware_supported one
29     EXIT_CRITICAL();
30     // resume a software thread
31     /* call RTOS API here */
32   }
33 }
```

Suspend Thread: Suspends a thread. It checks if the thread to be suspended is a software or a hardware-supported one (line 6-10). If hardware-supported, the thread's state is changed to *suspended* and then, it evaluates this new scheduling point (line 13-26). If software-supported, the equivalent standard RTOS API should be called instead (line 31).

```
1  void HW_Suspend_Thread( unsigned int thread_to_suspend )
2  {
3    unsigned int   HW_thread_id;
4    unsigned int   scheduler_running;
5    ENTER_CRITICAL();
6    // check if thread is a hardware-supported one
7    asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (thread_to_suspend));
8    // if a thread is found on hardware, HW_thread_id will have the thread handler
```

```
9     asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
10    if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
11    { // thread is a hardware_supported one
12      // set the thread state as suspended
13      asm volatile ("PUSH {R0}");
14      asm volatile ("MOV  R0, #0x02");
15      asm volatile ("MCR  p14, 0, R0, c1, c0, 2");
16      asm volatile ("POP  {R0}");
17      // check if scheduler is running
18      asm volatile ("MRC  p14, 0, %[value], c2, c0, 7": [value]"+r" (scheduler_running));
19      if( scheduler_running )
20      {
21        EXIT_CRITICAL();    // trigger a context switch
22        asm volatile ("SWI 0");   // to find the thread with highest priority
23      }
24      else
25        EXIT_CRITICAL();
26    }
27    else
28    { // thread is not a hardware_supported one
29      EXIT_CRITICAL();
30      // suspend a software thread
31      /* call RTOS API here */
32    }
33  }
```

Priority Set: Modifies the priority of a thread. It checks if the thread to change the priority is a software or a hardware-supported one (line 6-10). If hardware-supported, it selects the thread and sets its new priority and then, it analyzes this new scheduling point since a thread may become the highest priority thread among all those ready to run (line 13-23). If software-supported, the equivalent standard RTOS API should be called instead (line 28).

```
1   void HW_Priority_Set( unsigned int thread_id, unsigned int new_priority )
2   {
3     unsigned int HW_thread_id;
4     unsigned int scheduler_running;
5     ENTER_CRITICAL();
6     // check if thread is a hardware-supported one
7     asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (thread_id));
8     // if a thread is found on hardware, HW_thread_id will have the thread handler
9     asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
10    if(HW_thread_id != (unsigned int)0xFFFFFFFF)
11    {  // thread is a hardware_supported one
12      // set the new priority
13      asm volatile ("MCR  p14, 0, %[value], c1, c0, 1": [value]"+r" (new_priority));
14      // check if scheduler is running
15      asm volatile ("MRC  p14, 0, %[value], c2, c0, 7": [value]"+r" (scheduler_running));
16      if( scheduler_running )
17      {
18        EXIT_CRITICAL();  // trigger a context switch
19        asm volatile ("SWI 0"); // to find the thread with highest priority
```

```
20      }
21      else
22        EXIT_CRITICAL();
23    }
24    else
25    { // thread is not a hardware_supported one
26      EXIT_CRITICAL();
27      // change software thread priority
28      /* call RTOS API here */
29    }
30  }
```

Priority Get: Obtains the priority of a thread. It checks if the thread to get the priority is a software or a hardware-supported one (line 6-10). If hardware-supported, it selects the thread and gets its priority (line 13-15). If software-supported, the equivalent standard RTOS API should be called instead. (line 20).

```
1   unsigned int HW_Priority_Get( unsigned int thread_to_get_prio )
2   {
3     unsigned int   HW_thread_id;
4     unsigned int    prio_return;
5     ENTER_CRITICAL();
6     // check if thread is a hardware-supported one
7     asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (thread_to_get_prio));
8     // if a thread is found on hardware, HW_thread_id will have the thread handler
9     asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
10    if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
11    { // thread is a hardware_supported one
12      // get thread's priority
13      asm volatile ("MRC  p14, 0, %[value], c1, c0, 1": [value]"+r" (prio_return));
14      EXIT_CRITICAL();
15    }
16    else
17    { // thread is not a hardware_supported one
18      EXIT_CRITICAL();
19      // get software thread priority
20      /* call RTOS API here */
21      prio_return = 0;
22    }
23    return prio_return;
24  }
```

Delay Thread: Delays the calling thread by a specific number of OS ticks. It checks if the thread delaying itself is a software or a hardware-supported one (line 8-12). If hardware-supported, sets its delay time (line 14-24); If the number of specified ticks is zero (line 14-17), then an action must be performed accordingly with the RTOS specification, otherwise it evaluates

this new scheduling point. If software-supported, the equivalent standard RTOS API should be called instead (line 29).

```
1  void HW_Delay_Thread( unsigned int ticks_to_delay )
2  {
3    // Only the running thread can call this function
4    // NULL for FreeRTOS; OS_PRIO_SELF for uCOSII to get the id of calling thread
5    unsigned int thread_to_sleep = NULL;
6    unsigned int HW_thread_id;
7    ENTER_CRITICAL();
8    // check if thread is a hardware-supported one
9    asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (thread_to_sleep));
10   // if a thread is found on hardware, HW_thread_id will have the thread handler
11   asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
12   if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
13   { // thread is a hardware_supported one
14     if(ticks_to_delay == 0)
15       // RTOS specific
16       // FreeRTOS uses asm volatile ("SWI 0");
17       // uCOSII uses return;
18     else
19     { // set the delay time
20       asm volatile ("MCR  p14, 0, %[value], c1, c0, 7": [value]"+r" (ticks_to_delay));
21       EXIT_CRITICAL();
22       asm volatile ("SWI 0");
23     }
24   }
25   else
26   { // thread is not a hardware_supported one
27     EXIT_CRITICAL();
28     // delay a software thread
29     /* call RTOS API here */
30   }
31 }
```

Start Scheduler: Starts the hardware scheduler (line 6) and triggers the execution of the highest priority thread (line 8-11).

```
1  void HW_Start_Scheduler( void )
2  {
3    asm volatile("PUSH  {R0}");
4    asm volatile("MOV  R0, #1");
5    // Start scheduler
6    asm volatile("MCR  p14, 0, R0, c2, c0, 7");
7    // Updates the highest priority thread
8    asm volatile("MCR  p14, 0, R0, c2, c0, 0");
9    asm volatile("POP  {R0}");
10   // trigger a context-switch
11   asm volatile ("SWI 0");
12 }
```

Stop Scheduler: Suspends the hardware scheduler (line 7), any triggered context-switch will not have any effect.

```
1  void HW_Stop_Scheduler( void )
2  {
3    asm volatile("PUSH  {R0}");
4    asm volatile("MOV  R0, #0");
5    // Stop scheduling
6    asm volatile("MCR  p14, 0, R0, c2, c0, 7");
7    asm volatile("POP  {R0}");
8  }
```

The following hAPIs are related with synchronization mechanisms. There is a limitation on the use of these mechanisms. These hardware-supported synchronization mechanisms should only be used by hardware-supported threads and the regular software synchronization mechanisms should only be used by regular software threads. Therefore, a hardware-supported thread cannot dispute a mutex or semaphore with a regular software thread. This limitation only affects the system if more threads are created than the number of threads supported in hardware. Also, at this moment, hardware-supported mutexes do not implement the priority inheritance mechanism.

Create Mutex: Creates a mutex. If the calling thread is a hardware-supported one (line 9-11), then if there are available mutexes in hardware (line 15-17) a mutex is created and its handler is returned. If the calling thread is a regular software thread, then the equivalent standard RTOS API should be called instead (line 33).

```
1  void* HW_Create_Mutex()
2  {
3    int mutex_id;
4    ENTER_CRITICAL();
5    // only hw-supported threads can use hw-supported mutexes
6    // NULL for FreeRTOS; OS_PRIO_SELF for uCOSII to get the id of calling thread
7    unsigned int calling_thread = NULL;
8    // check if calling thread is a hardware-supported one
9    asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (calling_thread));
10   // if a thread is found on hardware, HW_thread_id will have the thread handler
11   asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
12   if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
13   { // thread is a hardware_supported one
14     // check if there are hardware-supported mutexes available
15     asm volatile ("MRC  p14, 0, %[value], c4, c0, 7": [value]"+r" (mutex_id));
16     // if mutex_id is equal to 0xFFFF_FFFF all hardware-supported mutexes are full
17     if(mutex_id == 0xFFFFFFFF)
18     {
```

```
19          EXIT_CRITICAL();
20          // no mutexes available, fail to create mutex
21          return (void *)0;
22        }
23        else
24        { // create mutex
25          asm volatile ("MCR  p14, 0, R0, c4, c0, 6");
26          EXIT_CRITICAL();
27          return (void *)(mutex_id);
28        }
29      }else
30      { // thread is not a hardware_supported one
31        EXIT_CRITICAL();
32        // create a software mutex
33        /* call RTOS API here */
34        return (void *)0;
35      }
36    }
```

Delete Mutex: Deletes a mutex. It checks if the mutex to delete is a software or a hardware-supported one (line 6-10). If hardware-supported, it selects the mutex and change its state to *deleted* and its hardware slot becomes available for a new mutex (line 16). There are two implementations/configurations of this API depending on the OS used, configuration (1) the deletion is canceled if there are pending threads waiting for this mutex and (2) mutex is deleted and if there are pending threads waiting on it then their states are changed to *ready*. If software-supported, the equivalent standard RTOS API should be called (line 12).

```
1   void HW_Delete_Mutex( unsigned int Mutex )
2   {
3     unsigned int  mutex_id = (unsigned int)Mutex;
4     ENTER_CRITICAL();
5     // check if mutex is a hardware-supported one
6     asm volatile ("MCR  p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
7     // if a mutex is found on hardware, mutex_id will have the mutex handler
8     asm volatile ("MRC  p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
9     if(mutex_id == 0xFFFFFFFF)
10    { // mutex is not a hardware_supported one
11      EXIT_CRITICAL();
12      /* call RTOS API here */
13    }
14    else
15    { // mutex is a hardware_supported one
16
17      // configuration 1 - check if it is taken by any task
18      // unsigned int mutex_is_taken;
19      // asm volatile ("MRC  p14, 0, %[value], c4, c0, 1": [value]"+r" (mutex_is_taken));
20      // if(mutex_is_taken)
21      // {
```

```
22      //  EXIT_CRITICAL();
23      //  return; // there are pending threads, do not delete mutex
24      // }
25      // else
26      // {
27      //  asm volatile ("MCR  p14, 0, R0, c4, c0, 7"); // delete mutex
28      //  EXIT_CRITICAL();
29      // }
30
31      // configuration 2
32      // by hardware, if a mutex is deleted then all pending threads
33      // change to ready state
34      asm volatile ("MCR  p14, 0, R0, c4, c0, 7"); // delete mutex
35      EXIT_CRITICAL();
36    }
37  }
```

Take Mutex: Takes a mutex. It checks if the accessed mutex is a software or a hardware-supported one (line 6-10). If software-supported, the equivalent standard RTOS API should be called (line 14). If hardware-supported, it reads the mutex state and verifies if the mutex is already taken (line 21-28). If mutex is free, then mutex is taken (line 30-32). Otherwise, the thread can specify the time to wait for the mutex to be released (line 36-60). After blocking the thread to wait for the mutex to be released, a context-switch is triggered to find the new highest priority thread ready to run (i.e., this new scheduling point is evaluated).

```
1   int  HW_Take_Mutex(unsigned int Mutex, unsigned BlockingTime )
2   {
3     unsigned int  mutex_id  = Mutex;
4     unsigned int mutex_is_taken;
5     ENTER_CRITICAL();
6     // check if mutex is a hardware-supported one
7     asm volatile ("MCR   p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
8     // if a mutex is found on hardware, HW_xTask will have the mutex handler
9     asm volatile ("MRC   p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
10    if(mutex_id == 0xFFFFFFFF)
11    { // mutex is not a hardware_supported one
12      EXIT_CRITICAL();
13      // for this example we call the FreeRTOS API
14      /* call RTOS API here */
15      return 0;
16    }
17    else
18    {
19      while(1)
20      {
21        // is the mutex taken?
22        // reading mutex state, mutex_is_taken will have 0 if mutex is free,
23        // otherwise mutex_is_taken will have the thread id which owns the mutex
```

```
24          asm volatile ("MRC  p14, 0, %[value], c4, c0, 1": [value]"+r" (mutex_is_taken));
25
26          if(!mutex_is_taken)
27          { // mutex is not taken, so take mutex
28            asm volatile ("MCR  p14, 0, R0, c4, c0, 1");
29            EXIT_CRITICAL();
30            return 1;
31          }
32          else
33          { // mutex is taken
34            if(BlockingTime == 0)
35            {
36              EXIT_CRITICAL();
37              // RTOS specific
38              // FreeRTOS: thread does not want to wait for the mutex to be freed so returns
39              // uCOSII: thread wants to wait forever till mutex is released, do not return
40            }
41            else
42            {
43              // uCOSII specific
44              // if(BlockingTime == 0)
45              // BlockingTime = 0xFFFFFFFF; // thread wants to wait forever for the mutex;
46              // block the thread by setting the time to wait for the mutex to become
                    available
47              asm volatile ("MCR  p14, 0, %[value], c4, c0, 5": [value]"+r" (BlockingTime));
48              EXIT_CRITICAL();
49              // trigger a context switch
50              asm volatile ("SWI 0"); // to find the thread with highest priority
51
52              // thread wakes up again here (due to only two events):
53              // 1- Blocking time is finished and thus thread does not want to wait anymore;
                    so we set BlockingTime to zero
54              // 2- The mutex was given by another thread and thus this thread can now take
                    the released mutex
55              BlockingTime = 0;
56              ENTER_CRITICAL();
57              // select the mutex in hw
58              asm volatile ("MCR  p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
59            }
60          }
61        }
62      }
63  }
```

Give Mutex: Gives a mutex. It checks if the released mutex is a software or a hardware-supported one (line 6-10). If hardware-supported, the state of the mutex is set to free, i.e., not taken (line 18) and the scheduling point is analyzed as this action may unblock a higher priority thread. If software-supported, the equivalent standard RTOS API should be called (line 13).

```
1   int HW_Give_Mutex(unsigned Mutex)
2   {
```

```
3    unsigned int  mutex_id  = Mutex;
4    ENTER_CRITICAL();
5    // check if mutex is a hardware-supported one
6    asm volatile ("MCR  p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
7    // if a mutex is found on hardware, mutex_id will have the mutex handler
8    asm volatile ("MRC  p14, 0, %[value], c4, c0, 0": [value]"+r" (mutex_id));
9    if(mutex_id == 0xFFFFFFFF)
10   { // mutex is not a hardware_supported one
11     EXIT_CRITICAL();
12     /* call RTOS API here */
13     return 0;
14   }
15   else
16   { // mutex is a hardware_supported one
17     // give mutex
18     asm volatile ("MCR  p14, 0, R0, c4, c0, 2");
19     EXIT_CRITICAL();
20     // trigger a context switch
21     asm volatile ("SWI 0"); // a higher thread may be unblocked
22     return 1;
23   }
24 }
```

Create Semaphore: Creates a semaphore. If the calling thread is a hardware-supported one (line 9-11), then if there are available semaphores in hardware (line 15-17) a semaphore is created, its initial count and maximum count are initialized and its handler is returned. If the calling thread is a regular software thread, then the equivalent standard RTOS API should be called instead (line 20).

```
1  void *HW_Create_Semaphore( unsigned int max_count, unsigned int initial_count )
2  {
3    int semaphore_id;
4    ENTER_CRITICAL();
5    // only hw-supported threads can use hw-supported semaphores
6    // NULL for FreeRTOS; OS_PRIO_SELF for uCOSII to get the id of calling thread
7    unsigned int calling_thread = NULL;
8    // check if calling thread is a hardware-supported one
9    asm volatile ("MCR  p14, 0, %[value], c1, c0, 0": [value]"+r" (calling_thread));
10   // if a thread is found on hardware, HW_thread_id will have the thread handler
11   asm volatile ("MRC  p14, 0, %[value], c1, c0, 0": [value]"+r" (HW_thread_id));
12   if(HW_thread_id != (unsigned int) 0xFFFFFFFF)
13   { // thread is a hardware_supported one
14     // check if there are hardware-supported semaphores available
15     asm volatile ("MRC  p14, 0, %[value], c5, c0, 7": [value]"+r" (semaphore_id));
16     // if semaphore_id is equal to 0xFFFF_FFFF all hardware-supported semaphores are
           full
17     if(semaphore_id == 0xFFFFFFFF)
18     {
19       EXIT_CRITICAL();
20       // no semaphores available, fail to create semaphore
21       return (void *)0;
```

```
22      }
23      else
24      { // create semaphore
25        asm volatile ("MCR  p14, 0, R0, c5, c0, 6");
26        // set the initial semaphore count value
27        asm volatile ("MRC  p14, 0, %[value], c5, c0, 3": [value]"+r" (initial_count));
28        // set the maximum semaphore count value
29        asm volatile ("MRC  p14, 0, %[value], c5, c0, 4": [value]"+r" (max_count));
30        EXIT_CRITICAL();
31        return (void *)(semaphore_id);
32      }
33    }
34    else
35    { // thread is not a hardware_supported one
36      EXIT_CRITICAL();
37      // create a software semaphore
38      /* call RTOS API here */
39      return (void *)0;
40    }
41  }
```

Delete Semaphore: Deletes a semaphore. It checks if the semaphore to delete is a software or a hardware-supported one (line 5-9). If hardware-supported, selects the semaphore and change its state to *deleted* and its hardware slot becomes available for a new mutex (line 16). There are two implementations/configurations of this API depending on the OS used, configuration (1) the deletion is canceled if there are pending threads waiting for this semaphore and (2) semaphore is deleted and if there are pending threads waiting on it then their states are changed to *ready*. If software-supported, the equivalent standard RTOS API should be called instead (line 12).

```
1   void HW_Delete_Semaphore( unsigned int Semaphore )
2   {
3     unsigned int  semaphore_id = (unsigned int)Semaphore;
4     ENTER_CRITICAL();
5     // check if semaphore is a hardware-supported one
6     asm volatile ("MCR  p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
7     // if a semaphore is found on hardware, semaphore_id will have the semaphore handler
8     asm volatile ("MRC  p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
9     if(semaphore_id == 0xFFFFFFFF)
10    { // semaphore is not a hardware_supported one
11      EXIT_CRITICAL();
12      /* call RTOS API here */
13    }
14    else
15    { // semaphore is a hardware_supported one
16
17      // configuration 1 - check if it is taken by any task
18      // unsigned int semaphore_is_taken;
19      // asm volatile ("MRC  p14, 0, %[value], c4, c0, 1": [value]"+r" (semaphore_is_taken)
```

```
        );
20      // if(semaphore_is_taken)
21      // {
22      //   EXIT_CRITICAL();
23      //   return; // there are pending threads, do not delete semaphore
24      // }
25      // else
26      // {
27      //   asm volatile ("MCR  p14, 0, R0, c4, c0, 7"); // delete semaphore
28      //   EXIT_CRITICAL();
29      // }
30
31      // configuration 2
32      // by hardware, if a semaphore is deleted then all pending threads
33      // change to ready state
34      asm volatile ("MCR  p14, 0, R0, c5, c0, 7"); // delete semaphore
35      EXIT_CRITICAL();
36    }
37  }
```

Take Semaphore: Takes a semaphore. It checks if the accessed semaphore is a software or a hardware-supported one (line 6-10). If software-supported, the equivalent standard RTOS API should be called (line 14). If hardware-supported, it reads the semaphore state and verifies if the semaphore can be obtained (line 21-27). If semaphore can be obtained, then semaphore is taken (line 29-31). Otherwise, the thread can specify the time to wait until semaphore can be obtained (line 35-53). After blocking the thread in order to wait for the semaphore to be available, the new scheduling point is processed to find and then run the new highest priority thread ready to run.

```
1   int  HW_Take_Semaphore(unsigned int Semaphore, unsigned int BlockingTime )
2   {
3     unsigned int  semaphore_id  = xSemaphore;
4     unsigned int semaphore_is_full;
5     ENTER_CRITICAL();
6     // check if semaphore is a hardware-supported one
7     asm volatile ("MCR   p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
8     // if a semaphore is found on hardware, HW_xTask will have the semaphore handler
9     asm volatile ("MRC   p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
10    if(semaphore_id == 0xFFFFFFFF)
11    { // semaphore is not a hardware_supported one
12      EXIT_CRITICAL();
13      /* call RTOS API here */
14      return 0;
15    }
16    else
17    {
18      while(1)
19      {
20        // is the semaphore full?
```

```
21        // reading semaphore state, semaphore_is_full will have 0 if semaphore is free,
22        // otherwise semaphore_is_full indicates that it cannot be obtained
23        asm volatile ("MRC  p14, 0, %[value], c4, c0, 1": [value]"+r" (semaphore_is_full));
24
25        if(!semaphore_is_full)
26        { // semaphore is not full, so take semaphore
27          asm volatile ("MCR  p14, 0, R0, c4, c0, 1");
28          EXIT_CRITICAL();
29          return 1;
30        }
31        else
32        { // semaphore is full
33          if(BlockingTime == 0) // thread does not want to wait for the semaphore to be
                freed
34          {
35            EXIT_CRITICAL();
36            // RTOS specific
37            // FreeRTOS: thread does not want to wait for the semaphore to be released so
                  returns 0
38            // uCOSII: thread wants to wait forever till semaphore is released, do not
                  return
39          }
40          else
41          { // uCOSII specific
42            // if(BlockingTime == 0)
43            // BlockingTime = 0xFFFFFFFF;// thread wants to wait forever for the semaphore;
44            // block the thread by setting the time to wait for the semaphore to become
                  available
45            asm volatile ("MCR  p14, 0, %[value], c4, c0, 5": [value]"+r" (BlockingTime));
46            EXIT_CRITICAL();
47            // trigger a context switch
48            asm volatile ("SWI 0"); // to find the thread with highest priority
49
50            // thread wakes up again here (due to only two events):
51            // 1- Thread delay is finished and thus thread does not want to wait anymore;
                  so we set BlockingTime to zero
52            // 2- The semaphore was given by another thread and thus this thread can now
                  take the free semaphore
53            BlockingTime = 0;
54            ENTER_CRITICAL();
55            // select the semaphore in hw
56            asm volatile ("MCR  p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
57          }
58        }
59      }
60    }
61  }
```

Give Semaphore: Gives a semaphore. It checks if the released semaphore is a software or a hardware-supported one (line 6-10). If hardware-supported, the semaphore is released (line 19) and a new scheduling point is examined as this action may unblock a higher priority thread. If software-supported, the equivalent standard RTOS API should be called (line 13).

```
1  int HW_Give_Semaphore(unsigned int Semaphore)
2  {
3    unsigned int  semaphore_id  = Semaphore;
4    ENTER_CRITICAL();
5    // check if semaphore is a hardware-supported one
6    asm volatile ("MCR  p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
7    // if a semaphore is found on hardware, semaphore_id will have the semaphore handler
8    asm volatile ("MRC  p14, 0, %[value], c4, c0, 0": [value]"+r" (semaphore_id));
9    if(semaphore_id == 0xFFFFFFFF)
10   { // semaphore is not a hardware_supported one
11     EXIT_CRITICAL();
12     // for this example we call the FreeRTOS API
13     /* call RTOS API here */
14     return 0;
15   }
16   else
17   { // semaphore is a hardware_supported one
18     // give semaphore
19     asm volatile ("MCR  p14, 0, R0, c4, c0, 2");
20     EXIT_CRITICAL();
21     // trigger a context switch
22     asm volatile ("SWI 0");  // a higher thread may be unblocked
23     return 1;
24   }
25 }
```

In order to make our hAPIs transparent to the applications, all standard RTOS APIs are wrapper into our hAPIs. The following "defines" represent the mapping of the standard FreeRTOS APIs to our hAPIs, and they should be automatically applied during the elaboration time as previously mentioned in the Chapter 6.

```
#define  xTaskGenericCreate          HW_Create_Thread
#define  vTaskDelete                 HW_Delete_Thread
#define  vTaskResume                 HW_Resume_Thread
#define  vTaskSuspend                HW_Suspend_Thread
#define  vTaskPrioritySet            HW_Priority_Set
#define  uxTaskPriorityGet           HW_Priority_Get
#define  vTaskStartScheduler         HW_Start_Scheduler
#define  vTaskEndScheduler           HW_Stop_Scheduler
#define  vTaskDelay                  HW_Delay_Thread
#define  xSemaphoreCreateMutex       HW_Create_Mutex
#define  xSemaphoreCreateCounting    HW_Create_Semaphore
#define  vSemaphoreDelete            HW_Delete_SemaphoreMutex
#define  xSemaphoreTake              HW_Take_SemaphoreMutex
#define  xSemaphoreGive              HW_Give_SemaphoreMutex
```

FreeRTOS uses the same function to take, give or delete a semaphore or a mutex. Therefore, these three functions are mapped into intermediate hAPIs (`HW_Delete_SemaphoreMutex`, `HW_Take_SemaphoreMutex`, `HW_Give_SemaphoreMutex`) which only verify if the resource is a mutex or a semaphore and then forward it to the corresponding hAPI (e.g., `HW_Delete_Mutex`, or `HW_Delete_Semaphore`).

The following "defines" represent the mapping of the standard μCOSII APIs to our hAPIs.

```
#define  OSTaskCreateExt      HW_Create_Thread
#define  OSTaskDel            HW_Delete_Thread
#define  OSTaskResume         HW_Resume_Thread
#define  OSTaskSuspend        HW_Suspend_Thread
#define  OSTaskChangePrio     HW_Priority_Set
#define  OSStart              HW_Start_Scheduler
#define  OSTimeDly            HW_Delay_Thread
#define  OSMutexCreate        HW_Create_Mutex
#define  OSMutexDel           HW_Delete_Mutex
#define  OSMutexPend          HW_Take_Mutex
#define  OSMutexPost          HW_Give_Mutex
#define  OSSemCreate          HW_Create_Semaphore
#define  OSSemDel             HW_Delete_Semaphore
#define  OSSemPend            HW_Take_Semaphore
#define  OSSemPost            HW_Give_Semaphore
```

7.2 Memory Footprint

Memory footprint is an important factor to take into account when deploying a RTOS solution in memory constrained-devices. The memory footprint is often characterized by the amount of RAM and Read-Only Memory (ROM) requirements of an RTOS application running on a specific embedded system (Sheikh and Driscoll, 2011). The size of the ROM and RAM are usually affected by the kernel code, kernel data, run-time library code, data structures and global variables used. The memory footprint results presented were obtained by analyzing the memory map files generated by IAR. Since the memory footprint is highly dependent on the kernel components and runtime libraries used by RTOS applications, the results were obtained on the Thread-Metric Benchmark Suite. Using the IAR compiler with no optimizations, we obtained the memory footprint of each Thread-Metric benchmark application for three scenarios: (a) The native system, with no hardware support, where only regular software threads execute, i.e., only the standard software APIs are compiled into the final executable image (b) A system where only hardware-supported threads execute, i.e., only hAPIs compiled into the final executable image; and (c) A hybrid system where hardware-supported threads and regular software threads may co-exist, i.e., both hAPIs and standard APIs are compiled into the final executable image. Figure 7.4 presents the memory footprint results for FreeRTOS, for each benchmark application and for each of the three scenarios. Both, (b) and (c) scenarios present a slight memory footprint overhead. The maximum value is 6K bytes (5.9%), which is perfectly admissible

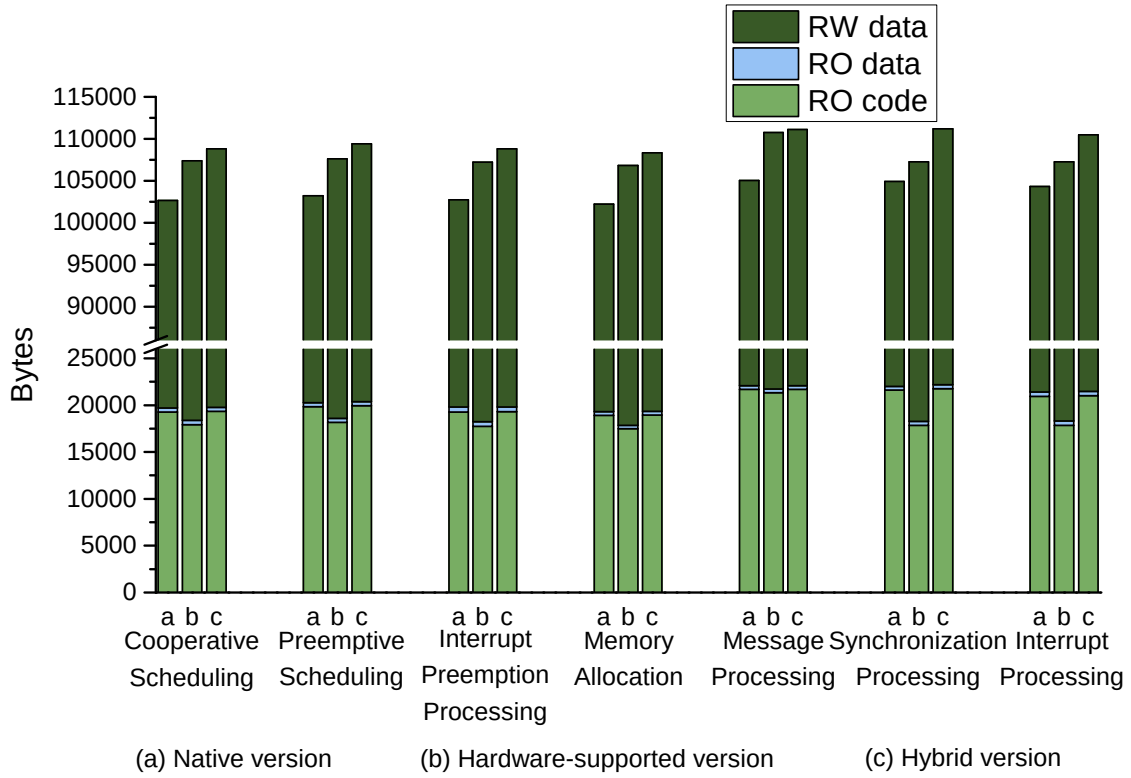


Figure 7.4: FreeRTOS memory footprint for each benchmark on the three scenarios.

in current embedded system platforms. Scenario (b), which only runs hardware-supported threads, requires less amount of read-only code since the hAPIs code is smaller than the software APIs code. However, it occupies more read-write data as it uses static memory to manage the hardware-supported thread's stack. The scenario (c) requires more memory space since it contains the software APIs and hAPIs together.

7.3 Conclusions

This chapter described the integration of the whole system stack. We showed how our approach causes only 6% of memory footprint overhead for a specific configuration. We presented the implementation of our hardware-based APIs and how we abstract those hAPIs from the application level in order to maintain the RTOS API specification intact. Future work will encompass the implementation of an exploration tool capable of performing DSE over all the different hardware-software configurations.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis presented novel micro-architectural enhancements to an ARM softcore processor promoting configurability, reuse, deterministic execution and energy- and performance-efficiency. We have shown how to tackle different issues and requirements of real-time systems. Firstly, we demonstrated how the rate-monotonic problem can be addressed using a novel approach based on a task-aware interrupt controller which unifies the priority space between threads and interrupts at a negligible hardware and memory usage cost. Then, we discussed and presented how real-time systems can take advantage of hardware multithreaded architectures to run current RTOS solutions, increasing deterministic execution and performance while maintaining portability as explained in Section 5. We have showed how a holistic and transparent hardware multithreading approach can be modeled and designed to support current RTOS solutions without the need to rewrite existent code.

Our solution, which is based on hardware multithreaded acceleration managed by a tightly-coupled hardware scheduler, demonstrated promising results in terms of predictability and performance, presenting very low area usage/performance overhead ratio. The designer-defined register-files containing all the data structures offloaded to hardware are easily interfaced by magic instructions allowing any RTOS application to be ported with low engineering effort to our architecture and benefit from the provided hardware support.

Our co-designed hardware-software agnostic solution allows different RTOS architectures to take advantage of the transparent hardware support, easing the adaption process of new RTOS to our solution. Legacy code reuse is ensured by mapping RTOS APIs into our hardware-based APIs. The flexibility of our approach was demonstrated by the significant results obtained running FreeRTOS and μCOSII.

Novel processor architectures can take advantage of the unified scheduling space due to the small impact of hardware multithreading (2% per extra thread as demonstrated in Section 5.5,) to boost the performance and predictability of new and more demanding RTOS applications. This is a great feature to include on new ASIPs, specifically developed for real-time applications where performance and determinism are very important aspects.

Current RTOS solutions can be easily adapted/ported to this new architectures as demonstrated in Section 5.3. RTOS designers may change how RTOSes kernel code is implemented, developing it taking in consideration the hardware multithreading support.

The advent of hardware multithreading support for RTOS can change the paradigm of software-only RTOSes and co-designed hardware-software RTOS implementations can become in a near future a standard approach.

8.2 Limitations

Some limitations can be found in our architecture:

- Supporting other RTOSes must be partially done in a manual way, i.e., the remapping of the RTOS's APIs to our hAPIs is not automatically done. As previously described, our strategy for agnosticism is based on API mapping strategy. APIs of existent RTOSes must be mapped to the APIs supported in hardware. Each time a new RTOS is added to the supported RTOSes list, the APIs of the desired RTOS must be analyzed and a new XMLized mapping metamodel instance developed.
- The hardware-supported threads cannot benefit from all the features provided by the RTOSes as regular software threads. Although, these features can be easily implemented/upgraded in the hardware, hardware-supported

threads are limited to the current implemented features.

- There are some limitations when using a configuration where hardware-supported threads can also be executed as regular software threads (i.e., hybrid threads), as hardware-supported threads can only use hardware-supported synchronization mechanisms and vice-versa.
- Currently, the scheduler only implements a priority-based scheduling algorithm, therefore only RTOSes based on this type of scheduling algorithm can be run in our architecture.

8.3 Future Work

Future work will encompass the offload of other kernel functionalities to hardware. Currently, thread management and thread synchronization APIs are supported but there is space for more services offloading such as the communication related APIs (e.g., queues mechanisms).

The possibility of supporting dynamic switch of software threads to hardware-supported threads and vice-versa will be studied. For instance, if a hardware-supported thread finishes its execution freeing a hardware slot, a software thread could be transferred to the free hardware slot to take inherent advantages of the hardware support.

New scheduling algorithms to tackle real-time requirements such as Earliest Deadline First (EDF) algorithm will be added, allowing the system to contemplate deadline constraints.

Future work will focus on verifying the power efficiency of our approach which was not performed throughout this thesis due to limited time. Appropriate benchmarking will be realized to approve the benefits of our approach in terms of energy.

Research will be carried out towards code transformers to allow a seamless integration/adaptation of new RTOS architectures into our agnostic stack. Moreover, the implementation of an exploration tool capable of performing DSE is expected.

Furthermore, the opportunity to migrate application specific functionalities to hardware will be studied allowing high-level synthesis (HLS) acceleration. This will allow integrating our architecture as a front-end for an EDA design flow tool

such as Vivado design flow.

Ultimately, research will proceed towards the refactoring of our architecture to allow fine-grain customizations. The ultimate goal should be to develop a profiling tool which explores the migration of software threads to hardware accordingly to the application demands and constraints, through a hardware-software co-design methodology.

Bibliography

J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, "Run-time services for hybrid cpu/fpga systems on chip," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, dec. 2006, pp. 3 –12.

S. Ahuja, S. Gurumani, C. Spackman, and S. Shukla, "Hardware coprocessor synthesis from an ansi c specification," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 58 –67, july-aug. 2009.

Altera, *Advanced Synthesis Cookbook*, 1st ed., Altera Corporation, 2011.

Altera, *Nios II Custom Instruction*, Altera Corporation, 2011.

D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving programming model abstractions for reconfigurable computing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 34 –44, jan. 2008.

ARC. (2012, Aug.) Arc international @ONLINE. [Online]. Available: http://www.arc.com

G. Ascia, V. Catania, and M. Palesi, "A ga-based design space exploration framework for parameterized system-on-a-chip platforms," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 4, pp. 329 – 346, aug. 2004.

K. Atasu, W. Luk, O. Mencer, C. Ozturan, and G. Dundar, "Fish: Fast instruction synthesis for custom processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, no. 1, pp. 52 –65, jan. 2012.

Atmel, *AT91SAM9XE Programmer Guide*, Atmel, 2010.

Atmel, *DATASHEET: SMART ARM-based Embedded MPU, AT91SAM9XE Series, Revision D*, Atmel, Octorber 2014.

I. Bahri, M. Benkhelifa, and E. Monmasson, “Hw-sw real-time operating system for ac drive applications,” in *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2012 International Symposium on*, June 2012, pp. 194–199.

J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park, “An energy-efficient processor architecture for embedded systems,” *Computer Architecture Letters*, vol. 7, no. 1, pp. 29–32, Jan 2008.

R. Barry, *Using the FreeRTOS Real Time Kernel*, 1st ed., FreeRTOS, 2010.

G. Beltrame, L. Fossati, and D. Sciuto, “Resp: A nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1857 –1869, dec. 2009.

G. Bloom, G. Parmer, B. Narahari, and R. Simha, “Shared hardware data structures for hard real-time systems,” in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT ’12. New York, NY, USA: ACM, 2012, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/2380356.2380382

U. Bordoloi, H. P. Huynh, S. Chakraborty, and T. Mitra, “Evaluating design trade-offs in customizable processors,” in *Design Automation Conference, 2009. DAC ’09. 46th ACM/IEEE*, july 2009, pp. 244 –249.

C.-C. Chen, C.-T. Yeh, and I.-J. Huang, “Software-based microprocessor verification methodology for linux booting,” in *Next-Generation Electronics (ISNE), 2013 IEEE International Symposium on*, Feb 2013, pp. 325–328.

T. S. Consortium. (2012, "Set") The spirit consortium @ONLINE. [Online]. Available: http://www.accellera.org

J. Cooling and P. Tweedale, “Task scheduler co-processor for hard real-time systems,” *Microprocessors and Microsystems*, vol. 20, no. 9, pp. 553 – 566, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933197000021

K. Craig, “White paper: MapuSoft OS Abstractor,” MapuSoft Technologies, Tech. Rep., 2015, accessed: 2015-08-03. [Online]. Available: http://www.mapusoft.com/wp-content/uploads/documents/osabstractor_whitepaper.pdf

N. Dave and M. Pellauer, "UNUM: A General Microprocessor Framework Using Guarded Atomic Actions," in *in Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture. IEEE Computer Society*, 2005.

R. Dimond, O. Mencer, and W. Luk, "CUSTARD - a customisable threaded FPGA soft processor and tools," in *Field Programmable Logic and Applications, 2005. International Conference on*, Aug 2005, pp. 1–6.

G. Dittmann, "On instruction-set generation for specialized processors," Ph.D. dissertation, Swiss Federal Institute of Technology, Zurich, 2006.

I. Express Logic. (2015) Thread-Metric Benchmark Suite. [Online]. Available: http://rtos.com/downloads/articles_and_white_papers-1/

A. Gaisler. (2012, Aug.) Leon3 @ONLINE. [Online]. Available: http://www.gaisler.com

R. Ginosar, "Metastability and synchronizers: A tutorial," *Design Test of Computers, IEEE*, vol. 28, no. 5, pp. 23–35, Sept 2011.

T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares, "Bringing Hardware Multithreading to the Real-Time Domain," *Embedded Systems Letters, IEEE*, (Accepted) 2015.

T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-Aware Interrupt Controller: Priority Space Unification in Real-Time Systems," *Embedded Systems Letters, IEEE*, vol. 7, no. 1, pp. 27–30, March 2015.

R. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE*, vol. 20, no. 2, pp. 60 –70, mar/apr 2000.

L. Goudge and S. Segars, "Thumb: Reducing the Cost of 32-bit RISC Performance in Portable and Consumer Applications," in *Proceedings of the 41st IEEE International Computer Conference*, ser. COMPCON '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 176–. [Online]. Available: http://dl.acm.org/citation.cfm?id=792769.793631

N. Goulding-Hotta, J. Sampson, Q. Zheng, V. Bhatt, J. Auricchio, S. Swanson, and M. Taylor, "Greendroid: An architecture for the dark silicon age," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, Jan 2012, pp. 100–105.

D. Gour and M. Jain, "Asip design space exploration: Survey and issues," in *International Journal of Computer Science and Information Security*, vol. 9, no. 4, 2011.

M. Grad and C. Plessl, "Just-in-time instruction set extension - feasibility and limitations for an fpga-based reconfigurable asip architecture," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 278–285.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001.

P. Hallschmid and R. Saleh, "Fast design space exploration using local regression modeling with application to asips," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 3, pp. 508 –515, march 2008.

R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 37–47, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815968

W. Hofer, D. Lohmann, F. Scheler, and W. Schroder-Preikschat, "Sloth: Threads as Interrupts," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, Dec 2009, pp. 204–213.

W. Hofer, D. Lohmann, and W. Schroder-Preikschat, "Sleepy Sloth: Threads as Interrupts as Threads," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 67–77.

M. Hohenauer and R. Leupers, *C Compilers for ASIPs: Automatic Compiler Generation with LISA*, 1st ed. Springer Publishing Company, Incorporated, 2009.

H. P. Huynh, Y. Liang, and T. Mitra, "Efficient custom instructions generation for system-level design," in *Field-Programmable Technology (FPT), 2010 International Conference on*, dec. 2010, pp. 445 –448.

X. Iturbe, K. Benkrid, A. T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, and A. Perez, "R3tos: A reliable reconfigurable real-time operating system," in *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, june 2010, pp. 99 –104.

P. Jääskeläinen, H. Kuitala, T. Pitkänen, and J. Takala, "Reducing the overheads of hardware acceleration through datapath integration," in *Proc. SPIE 6821, Multimedia on Mobile Devices 2008*. Bellingham, Wash.: Society of Photo-optical Instrumentation Engineers, 2008.

Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization," in *Design, Automation and Test in Europe, 2005. Proceedings*, march 2005, pp. 12 – 17 Vol. 1.

K. D. Kissell. (2007) Demystifying multithreading and multi-core. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1271568

A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai, "Effectiveness of the ASIP design system PEAS-III in design of pipelined processors," in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, 2001, pp. 649 –654.

S. Kleiman and J. Eykholt, "Interrupts As Threads," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 2, pp. 21–26, Apr. 1995.

P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 45–51. [Online]. Available: http://doi.acm.org/10.1145/944645.944656

D. Koufaty and D. Marr, "Hyperthreading technology in the netburst microarchitecture," *Micro, IEEE*, vol. 23, no. 2, pp. 56–65, March 2003.

C. Kumar, S. Vyas, J. Shidal, R. Cytron, C. Gill, J. Zambreno, and P. Jones, "Improving system predictability and performance via hardware accelerated data structures," in *Proceedings of Dynamic Data Driven Application Systems (DDDAS)*, June 2012.

M. Labrecque and J. Steffan, "Improving pipelined soft processors with multi-threading," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, aug. 2007, pp. 210 –215.

J. J. Labrosse, "White paper: Hardware-accelerated rtos: uc/os-iii hw-rtos and the r-in32m3," Micrium Embedded Software, Tech. Rep., 2014, accessed: 2015-08-03. [Online]. Available: http://micrium.com/hardware-accelerated-rtos-%C2%B5cos-iii-hw-rtos-and-the-r-in32m3/

J. J. Labrosse, *uC/OS-II: The Real-Time Kernel*, 1st ed., Micrium Press, 2015.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks," in *in Ambient Intelligence*. Springer Verlag, 2004.

L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, April 2006.

L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Integrated Task and Interrupt Management for Real-Time Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 2, Jul. 2012.

L. Lindh, "Fastchart-a fast time deterministic CPU and hardware based real-time-kernel," in *Real Time Systems, 1991. Proceedings., Euromicro '91 Workshop on*, Jun 1991, pp. 36–40.

M. Liu, D. Liu, Y. Wang, M. Wang, and Z. Shao, "On improving real-time interrupt latencies of hybrid operating systems with two-level hardware interrupts," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 978 –991, july 2011.

H. Marzouqi, M. Al-Qutayri, K. Salah, D. Schinianakis, and T. Stouraitis, "A high-speed fpga implementation of an rsd-based ecc processor," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

J. Metrolho, "Miadl: linguagem para geraÃ§Ã£o automÃ¡tica de simuladores redireccionÃ¡veis," Ph.D. dissertation, Universidade do Minho, Portugal, 2008.

I. Mooney, V.J. and D. Blough, "A hardware-software real-time operating system framework for socs," *Design Test of Computers, IEEE*, vol. 19, no. 6, pp. 44 – 51, nov/dec 2002.

Z. Murtaza, S. Khan, A. Rafique, K. Bajwa, and U. Zaman, "Silicon real time operating system for embedded dsps," in *Emerging Technologies, 2006. ICET '06. International Conference on*, nov. 2006, pp. 188 –191.

T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai, "VLSI implementation of a real-time operating system," in *Design Automation Conference, 1997. Proceedings of the ASP-DAC '97 Asia and South Pacific*, jan 1997, pp. 679 –680.

M. Naotaka, I. Takuya, H. Shinya, T. Hiroaki, and S. Katsunobu, "ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems," in *Proceedings of the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, ser. OSPERT '14, 2014, pp. 9–16.

A. Nery, L. Jozwiak, M. Lindwer, M. Cocco, N. Nedjah, and F. Franca, "Hardware reuse in modern application-specific processors and accelerators," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, 31 2011-sept. 2 2011, pp. 140 –147.

P. Nsame, G. Bois, and Y. Savaria, "Adaptive real-time dsp acceleration for soc applications," in *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*, Aug 2014, pp. 298–301.

I. of Computing University of Campinas. (2012, "Set") The ArchC Architecture Description Language @ONLINE. [Online]. Available: http://archc.sourceforge.net/

A. Oliveira, L. Almeida, and A. de Brito Ferrari, "The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator," *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 890–904, March 2011.

S. E. Ong, S. C. Lee, N. Ali, and F. Hussin, "SEOS: Hardware Implementation of Real-Time Operating System for Adaptability," in *Computing and Networking (CANDAR), 2013 First International Symposium on*, Dec 2013, pp. 612–616.

G. Palermo, C. Silvano, and V. Zaccaria, "Respir: A response surface-based pareto iterative refinement for application-specific design space exploration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1816 –1829, dec. 2009.

S. Palnitkar, *Verilog Hdl: A Guide to Digital Design and Synthesis, Second Edition*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2003.

S. Panneerselvam and M. M. Swift, "Operating systems should manage accelerators," in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342788.2342792

S. Pinto, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral, and A. Tavares, "Porting SLOTH system to FreeRTOS running on ARM Cortex-M3," in *Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on*, June 2014.

N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul, "A high-level synthesis flow for custom instruction set extensions for application-specific processors," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, jan. 2010, pp. 707 –712.

E. PROGRAM. (2012, Aug.) Express program @ONLINE. [Online]. Available: http://www.ics.uci.edu/~express/

S. Radhakrishnan, H. Guo, and S. Parameswaran, "Customization of application specific heterogeneous multi-pipeline processors," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, march 2006, p. 6 pp.

F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, "Parallel, Hardware-supported Interrupt Handling in an Event-triggered Real-time Operating System," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 167–174.

O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Architecture implementation using the machine description language lisa," in *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, 2002, pp. 239 –244.

D. Seal, *ARM Architecture Reference Manual*, 2nd ed. Addison-Wesley, 2000.

Segger, *embOS for ARM and RealView Developer Suite*, 1st ed., Segger, 2008.

O. Shacham, "Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms," Ph.D. dissertation, Stanford University, 2011.

O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9–24, Nov 2010.

O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 623–629.

F. Sheikh and D. Driscoll, "White paper: Mentor Graphics - Measuring RTOS Performance: What? Why? How? ," Mentor Graphics, Tech. Rep., 2011.

Y. shuai Lu, L. Shen, L. bo Huang, Z. ying Wang, and N. Xiao, "Customizing computation accelerators for extensible multi-issue processors with effective optimization techniques," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, june 2008, pp. 197 –200.

M. Sindhwani, T. Oliver, L. Douglas, and T. Srikanthan, "RTOS Acceleration Techniques - Review and Challenges," in *Sixth Real-Time Linux Workshop*, Singapura, Nov 2004, pp. 123–128.

A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via Multithreaded and Multicore CPUs," *Computer*, vol. 43, no. 3, pp. 24–32, March 2010.

A. Solomatnikov, A. Firoozshahian, O. Shacham, Z. Asgar, M. Wachs, W. Qadeer, S. Richardson, and M. Horowitz, "Using a configurable processor generator for computer architecture prototyping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 358–369.

Y. Tang and N. Bergmann, "A hardware scheduler based on task queues for fpga-based embedded real-time systems," *Computers, IEEE Transactions on*, vol. 64, no. 5, pp. 1254–1267, May 2015.

R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, Mar 2000.

M. Varela, R. Cayssials, E. Ferro, and E. Boemo, "Real-time scheduling coprocessor for NIOS II processor," in *Programmable Logic (SPL), 2012 VIII Southern Conference on*, March 2012, pp. 1–6.

N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, "The ARISE Approach for Extending Embedded Processors With Arbitrary Hardware Accelerators," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 2, pp. 221 –233, feb. 2009.

T. P. Wijesinghe, "Design and implementation of a multithreaded softcore processor with tightly coupled hardware real-time operating system," Master's Thesis, College of Engineering and Mineral Resources, Nov 2008.

Xilinx, *MicroBlaze Processor Reference Guide*, Xilinx Corp, 2008.

Xilinx, *User Guide UG5850: Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v2.0*, Xilinx, April 2 2014.

Xilinx, *User Guide UG029: ChipScope Pro Software and Cores*, 14th ed., Xilinx, October 16 2012.

P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '05. New York, NY, USA: ACM, 2005, pp. 202–212. [Online]. Available: http://doi.acm.org/10.1145/1086297.1086325

P. Zaykov, G. Kuzmanov, A. Molnos, and K. Goossens, "Hardware task-status manager for an rtos with fifo communication," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–8.

L. Zhang, S. Li, Z. Yin, and W. Zhao, "A research on an asip processing element architecture suitable for fpga implementation," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 3, Dec 2008, pp. 441–445.

Y. Zhang and R. West, "Process-Aware Interrupt Scheduling and Accounting," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, Dec 2006, pp. 191–201.