A Multi-core Processor for Hard Real-Time Systems

Marco Paolieri

2011

A THESIS SUBMITTED IN FULFILLEMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy / Doctor per la UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

A Multi-core Processor for Hard Real-Time Systems

Marco Paolieri

2011

Advisor:
Francisco J. Cazorla Almeida
Barcelona Supercomputing Center

Co-Advisors:
Eduardo Quiñones
Barcelona Supercomputing Center
Mateo Valero Cortés
Universitat Politècnica de Catalunya

A THESIS SUBMITTED IN FULFILLEMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy / Doctor per la UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Acknowledgments

Before concluding my PhD I would like to spend few words to thank all the people that helped me and supported me during this *adventure*. It seems like yesterday that I started and instead another chapter of my life is already at the end. I am really glad to have decided to do a PhD because I think I learnt a lot from the people who surrounded me both at personal and professional level.

First and foremost I want to thank my advisor Francisco J. Cazorla and my co-advisors Eduardo Quiñones and Mateo Valero. It has been an honor to be their PhD student. Thanks to them I have learnt what *research* truly means!

The work done in this thesis was part of an FP7 European project called Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (MERASA - grant agreement number FP7-216415). I am especially grateful to Prof. Theo Ungerer to be the coordinator of such a great project and to all the other partners from Univesity of Augsburg, Université Paul Sabatier, Rapita Systems Ltd., Honeywell International s.r.o.: Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Hugues Casse, Christine Rochange, Sascha Uhrig, Mike Gerdes, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, Jörg Mische and Julian Wolf.

Many thanks to all my colleagues and friends from BSC, and in particular from the CAOS group with whom I shared the office. Four years have passed so fast that

it seems like yesterday I was jumping on the metro towards the university to start my first day as a *PhD student* in the beautiful city of Barcelona...and now I can finally remove the word *student*. I will miss a lot Barcelona and BSC!

A special thank to my friend Edu for his great help, advices and presence. He was always there when I had questions, doubts or problems. I really enjoyed all the business trips we had together.

Thanks to my Italian friends in the group: Roberto and Alessandro. Roberto thanks for all the advices and all the support! Ale, I will never forget all the time we spent together, having lunch and discussing about anything, trying to figure out how we could open our business.

I express deep gratitude and heartfelt thanks to my parents Paolo and Ida for standing by me. They always have given a good ear to me and helped me. I would also like to thank them for the advices and the guidelines they gave me whenever I was low!

A really special thank to Mara, who has always been by my side for these four years, and during the last three she made Barcelona even more beautiful! She is always able to make me happy and make every day special.

Lastly, thanks to those I have forgotten..sorry about that!

“ *You’ve got to find what you love. And that is as true for your work as it is for your lovers. Your work is going to fill a large part of your life, and the only way to be truly satisfied is to do what you believe is great work. And the only way to do great work is to love what you do. If you haven’t found it yet, keep looking. Don’t settle. As with all matters of the heart, you’ll know when you find it. And, like any great relationship, it just gets better and better as the years roll on. So keep looking until you find it.*

Don’t settle.“

Steve Jobs

Abstract

The increasing demand for new functionalities in current and future hard real-time embedded systems, like the ones deployed in automotive and avionics industries, is driving an increment in the performance required in current embedded processors. Multi-core processors represent a good design solution to cope with such higher performance requirements due to their better performance-per-watt ratio while maintaining the core design simple. Moreover, multi-cores also allow executing mixed-criticality level workloads composed of tasks with and without hard real-time requirements, maximizing the utilization of the hardware resources while guaranteeing low cost and low power consumption.

Despite those benefits, current multi-core processors are less analyzable than single-core ones due to the interferences between different tasks when accessing hardware shared resources. As a result, estimating a meaningful *Worst-Case Execution Time* (WCET) estimation – i.e. to compute an upper bound of the application's execution time – becomes extremely difficult, if not even impossible, because the execution time of a task may change depending on the other threads running at the same time. This makes the WCET of a task dependent on the set of inter-task interferences introduced by the co-running tasks. Providing a WCET estimation independent from the other tasks (*time composability* property) is a key requirement in hard real-time systems.

This thesis proposes a new multi-core processor design in which time composability is achieved, hence enabling the use of multi-cores in hard real-time systems. With our proposals the WCET estimation of a HRT is independent from the other co-running tasks. To that end, we design a multi-core processor in which the maximum delay a request from a *Hard Real-time Task* (HRT), accessing a hardware shared resource can suffer due to other tasks is bounded: our processor *guarantees* that a request to a shared resource cannot be delayed longer than a given *Upper Bound Delay* (UBD).

In addition, the UBD allows identifying the impact that different processor configurations may have on the WCET by determining the *sensitivity* of a HRT to different resource allocations. This thesis proposes an off-line task allocation algorithm (called IA^3: Interference-Aware Allocation Algorithm), that allocates tasks in a task set based on the HRT's sensitivity to different resource allocations. As a result the hardware shared resources used by HRTs are minimized, by allowing Non Hard Real-time Tasks (NHRTs) to use the rest of resources. Overall, our proposals provide analyzability for the HRTs allowing NHRTs to be executed into the same chip without any effect on the HRTs.

The previous first two proposals of this thesis focused on supporting the execution of multi-programmed workloads with mixed-criticality levels (composed of HRTs and NHRTs). Higher performance could be achieved by implementing multi-threaded applications. As a first step towards supporting hard real-time parallel applications, this thesis proposes a new hardware/software approach to guarantee a predictable execution of software pipelined parallel programs.

This thesis also investigates a solution to verify the timing correctness of HRTs without requiring any modification in the core design: we design a hardware unit which is interfaced with the processor and integrated into a functional-safety aware methodology. This unit monitors the execution time of a block of instructions and it detects if it exceeds the WCET. Concretely, we show how to handle timing faults on a real industrial automotive platform.

Contents

CHAPTER 1

Introduction

Real-time embedded systems surround us in every-day life: they are present when we listen to music, switch on the TV, drive a car or flight on an airplane. The correctness of these systems does not only rely on the *functional correctness* of their operations but also on their *time correctness*: it is required to guarantee that the execution time of an operation never exceeds its corresponding *deadline*. To do so, a *Worst Case Execution Time* (WCET) analysis has to be derived. The WCET estimation of a task is an upper bound estimation of the maximum execution time an operation can take to execute on a specific hardware platform.

Real-time embedded systems can be classified into several categories depending on the *severity* of their time constraints; among them we identified three main classes: *hard real-time*, *soft real-time* and *firm real-time*.

- Hard Real-time systems composed of Hard Real-time Tasks (HRTs) are in charge of controlling the most critical functions of a system, e.g. the brake control system in a car and the flight control system in an airplane. HRTs must never miss any deadline, since the consequence of missing a deadline could

involve critical failures or loss of human lives. Thus, computing safe WCET estimations becomes mandatory.

- Soft real-time systems are composed by soft real-time tasks. The usefulness of a result degrades after its deadline, thereby degrading the system's Quality of Service (QoS). Hence such systems can allow missing some deadlines because the consequence is a reduction of the Quality of Service (QoS) [25] [24] [26]. Examples of soft real-time systems are video algorithms and image processing, where deadline misses involve a decreased service quality (e.g., dropping frames while displaying a video).

- Firm real-time systems are in the border between hard and soft real-time systems. An example of such systems are Software-Defined Radio applications [64]. Missing a deadline neither has catastrophic consequences like for hard real-time systems nor involves a quality of service degradation like for soft real-time systems, but it is highly undesirable and it may fail to comply with a given standard, hence being not functionally correct.

In this thesis we focus on hard real-time systems. The market for this kind of systems is really important: it has experienced unprecedented growth over the last five years and is expected to continue to grow steadily for the foreseeable future. A study performed by the international ARC Advisory Group claims that this market is expected to increase with an yearly rate of over 12% [1], reaching an overall market of $2.5 billion in 2012. Gartner Inc. has recently reported that the semiconductor content in automotive safety systems will increase from $2.2 billion in 2009 to $4.3 billion in 2014 [29].

Like in any other computing system in hard real-time systems it is necessary to ensure functional correctness. In addition to that, like any other embedded system, a hard real-time system must satisfy the stringent requirements in terms of low power consumption, low cost and low weight. One of the main requirements that distinguish hard real-time systems from any other type of systems is the level of *assurance* on the functional and time correctness of the system. This additional requirement is neces-

sary because these systems are in charge of controlling the most critical functions, and so it is necessary to prevent any catastrophic consequence that could occur due to a deadline miss or any misbehavior of the system. To ensure correctness hard real-time systems must be *verified*. Verification is the process used to check that the requirements of a system are satisfied. The verification can be classified into functional verification and timing verification; the former checks that the system is functionally correct while the latter verifies that timing constraints are met. Verification is generally covered by several standards that are used in the different hard real-time domains: examples of those are ISO26262 [5] in automotive, and DO-178B [6] in avionics. For industries is of primary importance to keep the costs of such verification low [27].

One of the main goals of this thesis is to provide timing verification at low cost. During the timing verification phase it is required the system to be *timing analyzable*. In other words, it is required to derive a safe and tight WCET for the tasks and perform a schedulability analysis. The schedulability analysis uses the WCET estimations derived with the WCET analysis performed for every task of the system on the target hardware or on a model of the target platform.

1.1 Challenges of Future Hard Real-Time Systems

Until recently, hard real-time systems have been designed following the *Federated Architectures* [67] design principle, where each function is implemented in a different hardware unit (processor, sensors, actuators, etc.). The separation of functions facilitates timing and functional verifications, maintaining the cost of the system low. Federated Architectures simplify the verification, providing a separation of responsibilities, since every provider can implement the hardware and the software for a function, independently from the other suppliers. Moreover as each function is implemented on a different hardware unit, the processors commonly used in the hard real-time domain are simple uniprocessor architectures, with short pipeline and in-order execution (e.g. most of the processors used in cars are still 8-bit microcontrollers lacking caches or pipelines). Such simple processors provide enough performance and time analyz-

ability. However, as the number of functions implemented increase, so the number of units does. Implementing more functions in a system following a Federated Architecture approach implies a high number of hardware units. This makes federated implementations inefficient in terms of size, weight and energy consumption.

Nowadays, the complexity of the systems is rapidly increasing as an answer to the requirements for more and much better services, and consequently the number of the functions implemented into a hard real-time system. These new requirements aim to increase safety, comfort, number and quality of services, and lower emissions as well as fuel demands for automotive, avionic and automation applications. A typical high end vehicle, for example, includes more than 70 MCUs (Microcontroller Units) [27]. Moreover, the functions to be implemented are getting more complex. Hence, one of the main requirements of current and future hard real-time embedded systems is to provide high performance in order to satisfy the computational requirements of these functions. Such a demand for increased computational performance is widespread among the industries [7]:

1. In the avionics industry, the ever increasing demands for additional aircraft functionality, safety and security drive both commercial and military markets towards the need for greater performance. For instance, next generation UAVs (unmanned aerial vehicles) are expected to be much more complex. In addition to that it would be desirable to host the applications workload of the airplane functionalities on as few hardware as possible in order to reduce size, weight and power. As demonstration of such trend, in the last F-35 tactical fighter 90% of the fuctions are managed by software [18].

2. In the automotive industry weight constraints are less stringent than in avionics but cost demands are significantly more severe. In today's automotive designs MCUs (Microcontroller Units) are involved in controlling function and safety in airbags, brakes and chassis control, engine control and in the future they will play a crucial role in x-by-wire cars. If higher performance than what currently available is provided future automotive systems can evaluate and process more

sensor data being able to master more complex situations, for instance steer-by-wire, automatic emergency-braking triggered by collision avoidance systems.

To cope with the inefficiency of Federated Architectures as embedded systems become more complex, automotive and avionics industries are adopting *Integrated Architectures*, such as Integrated Modular Avionics (IMA) [15] or Automotive Open System Architecture (AUTOSAR) [3]. These architectures are becoming de facto standard in each industry respectively. Integrated Architectures execute more system functions into each hardware unit (i.e. a processor). The design principle is conceived from the fact that most of the federated computers perform essentially the same functions (input acquisitions, processing and computation, and output generation), hence, a natural optimization of resources is to share common subsystems that perform common subfunctions, in addition to standardizing interfaces and encapsulating services.

In Integrated Architectures a key design principle in order to contain the cost of timing verification is to guarantee that there is no interaction between the different functions sharing the resources. To that end, at functional level, it is necessary to provide functional isolation, such that a bug/misbehavior in a function does not affect the others. At timing level, it is necessary to provide timing isolation, such that the timing behavior of a task is not affected by the others. These features allow the different suppliers to perform the timing analysis independently. In order to provide functional and timing isolation, to avoid dependencies among different components, Integrated Architectures require hardware and software to ensure Incremental Qualification: it is the property according with, it is not required to re-certify unchanged components that interact with new components that are integrated into the system. By guaranteeing such property, components can be changed or upgraded without affecting the timing behavior of the others, hence without the need to re-analyze, re-integrate and in particular re-certify the system.

In order to maintain the timing verification costs at the level of the ones in Federated Architectures, incremental qualification relies on each software and hardware component exhibiting the property of *time composability*. Such property dictates that the timing behavior of an individual component does not change by the composition,

i.e. composing the system. Time composability also alleviates *System Integration*: it is well recognized that the most difficult and expensive problems, that appear during the development of a system, occur during its integration; multiple components, indeed, interact at software and hardware level. It is then essential to remove the dependencies between the different sub-systems, and, in particular, individual sub-suppliers can perform meaningful timing analysis of their components independently from other suppliers.

It is important to remark, that to be time composable it is also required to be time analyzable so the requirements of hard real-time systems of being both timing correct and to contain the verification costs can be ensured by providing time composability. The other main requirement in addition to time composability is, hence, higher performance that is required to cope with the new system requirements, and to allow integrating more functions into the same system.

1.2 Possible Solutions to Achieve High Performance Guaranteeing Time Composability

Embedded system designers could achieve such required high performance by designing more complex processors with longer pipelines, out of order execution or higher clock frequency. Unfortunately, these solutions are not suitable, because on the one hand complex processors with out of order execution suffer timing anomalies [56] due to their non deterministic run-time behaviors. A timing anomaly is a situation where the local worst-case does not entail the global worst-case. For instance, a cache miss – the local worst-case – may result in a shorter execution time, than a cache hit, because of scheduling effects. On the other hand, the high energy requirements of complex processors with longer pipeline or higher frequency do not satisfy the low-power constraints and the severe cost limitations of common embedded systems.

Embedded processor design is, indeed, not only performance-driven but it also requires meeting other goals as important as the performance: low power consumption, low cost. Moreover, clocking a processor at a higher frequency would potentially in-

crease the number of errors generated by radiations, e.g. on airplane. These multiple-goals make the design of embedded systems a very hard task; it is then necessary to trade-off all the requirements.

Multi-core processors are increasingly being considered as an effective solution to cope with the higher performance requirements while maintaining a relatively simple core design that avoids suffering from timing anomalies. This kind of processors scale performance putting multiple cores on a single chip, effectively integrating a complete multiprocessor on a chip providing better performance per watt ratio, maintaining low chip costs, low power consumption, etc. Moreover, multi-core processors ideally enable co-hosting applications with mixed criticality-levels (i.e. hard, soft and non real-time tasks). Co-hosting non-safety and safety critical applications on a common powerful multi-core processor is of paramount importance in the embedded system market. Overall multi-cores allow to schedule a higher number of tasks on a single processor so that the hardware utilization is maximized, while cost, size, weight and power requirements are reduced.

Unfortunately, despite the benefits offered by multi-core processors, they are much harder to analyze than single-core processors, due to inter-task interferences accessing shared resources such as on-chip buses and caches. Inter-task interferences appear when two or more tasks that share a hardware resource try to access it at the same time, so arbitration is required to select which task is granted access to such a shared resource, potentially delaying the requests of the other tasks that do not get the access to the shared resource granted. As a result, the execution time and so the WCET increase, making the WCET analysis extremely difficult, if not even impossible, because the execution time of a task may change depending on the other tasks running at the same time, and so this makes the WCET of a task dependent on the set of inter-task interferences introduced by the co-running tasks. Inter-task interferences are an important drawback when moving towards integrated architectures, that if not considered in the design, may prevent timing composability.

Hence, even though multi-core processors are good candidates for hard real-time systems due to the several advantages they provide, they do not satisfy the easy timing

verification requirement. Until now, there is no de facto solution to provide a time composable multi-core processor.

1.3 Thesis Contributions

This thesis proposes a multi-core processor design that provides time composability, and hence time analyzability, that is suitable for future hard real-time systems featuring Integrated Architectures. The aim of this thesis is to investigate the design of a multi-core processor for hard real-time systems that allows executing mixed-criticality workloads composed of HRTs and Non Hard Real-time Tasks (NHRTs) running at the same time achieving higher performance, satisfying the timing constraints and guaranteeing timing composability to maintain the cost of verification low. Our proposals provide analyzability for the HRTs and allow NHRTs to be executed into the same chip without affecting time analyzability. The HRTs are, in fact, not affected by NHRTs and the NHRTs can use all the resources that are not used by the HRTs. Our solution ensures time composability for the HRTs facilitating the timing verification, hence allowing to change, after the integration phase of the system, the tasks of the workload. It does not, indeed, require a re-estimation of the WCET of all the tasks in the task set. This reduces the cost of analyzability, and consequently the cost of the verification, as only the tasks involved in the change need to be re-analyzed, as opposed of having to re-analyze the whole system. This work is part of an FP7 European project called Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (MERASA - grant agreement number FP7-216415).

The approach followed in this thesis is that for every shared resource of our processor it is guaranteed that the maximum delay a request to a shared resource from a HRT can suffer due to other tasks is bounded. Our multi-core processor guarantees that a request from a HRT cannot be delayed longer than a given *Upper Bound Delay* (UBD). We analyze the UBD for common shared resources in a multi-core processor. Concretely, this thesis focuses on a multi-core architecture composed of a small number of cores (2 to 8), in which each core, that has its private data and

instruction first level caches, is connected to a second level shared cache through a shared bus. The shared cache is interfaced through a memory controller to an off-chip Double Data Rate Synchronous Dynamic Random Access Memory (DDRx SDRAM) memory device. Under this processor configuration the main sources of inter-task interferences are the on-chip shared bus, the shared cache and the memory controller. By doing so, we can ensure that the WCET is independent from the workload in which the HRT is running. For this reason we can ensure time composability, reducing the efforts and the costs necessary to perform the certification and verification phases of an integrated architecture. In particular:

- We analyze different arbitration policies used in the on-chip shared bus in order to determine the *Upper Bound Delay* (UBD) that a request from a HRT can suffer due to inter-task interferences.

- We also implement a dynamically partitioned cache to allow different HRTs to benefit from a second level cache. We analyze the effect of different cache partition sizes on the WCET estimations of different HRTs.

- We propose an analytical model of a JEDEC DDRx DRAM memory system to compute the UBD that a request from a HRT can suffer due to inter-task interferences accessing the shared memory. We also design a Real-Time Capable Memory Controller (RTCMC) for our multi-core processor.

- We introduce a hardware feature called *WCET Computation Mode*. In this execution mode, each HRT is run in isolation: The processor – on each access to both shared resources: the on-chip bus and the memory controller – artificially introduces the UBD that a request from HRT can suffer because of inter-task interferences. The result of the WCET analysis, it is an estimation that is guaranteed to provide a safe upper bound of the execution of the HRT when it runs in *Standard Execution Mode* together with other tasks sharing processor resources. The main advantage of our proposal is that it allows computing for each HRT a safe WCET estimation that does not depend on the other co-running tasks. Our

proposed multi-core processor can be easily analyzed by current measurement-based WCET tools without any *modification*. Hence, it is possible to perform WCET analysis of a multi-core processor using the same tool chain used for single-cores.

In addition, the UBD allows identifying the impact that different processor configurations may have on the WCET by determining the *sensitivity* of a HRT to different resource allocations. This thesis proposes an off-line task allocation algorithm (called IA^3: Interference-Aware Allocation Algorithm), that allocates the task set based on the HRT's sensitivity to different resource allocations. As a result the hardware shared resources used by HRTs are minimized, by allowing NHRTs to use the remaining resources. Hence, our approach also allows NHRTs to satisfy their high data processing demands using the resources not used by HRTs.

All the previous proposals of this thesis focused on supporting the execution of multi-programmed workloads composed of single-threaded mixed-criticality tasks (HRTs and NHRTs). The advantages of this are numerous, in particular the weight of the system is reduced, a smaller number of hardware units is necessary into the system with the consequence that the power consumption is reduced, easier installation (with less cables necessary to interconnect the systems), smaller costs, etc. Intuitively, higher average performance could be achieved by implementing multi-threaded applications.

As a first step towards the support of hard real-time parallel applications, this thesis proposes a new hardware/software approach to guarantee a predictable execution of software pipelined parallel programs. Our initial results show that a tight interaction between the time analysis tool and the programming language is required to provide reduced WCET estimations. Otherwise, although the average execution time decreases the WCET estimation may increase. To that end, we describe a software/hardware cache partitioning technique that reduces the inter-thread memory interferences generated by hard real-time software-pipelined parallel applications while still guaranteeing a predictable timing behavior.

This thesis also investigates a solution to verify the timing correctness of HRTs without requiring any modification in the core design: it is commonly the case that different subsuppliers provide the different IPs of a System-On-Chip, in that case it is not possible to modify an IP designed by another supplier. Hence, we design a hardware unit which is interfaced with the IP core and integrated into a functional-safety aware methodology. This unit monitors the execution time of a block of instructions and it detects if it exceeds the WCET. Concretely, we show how to handle timing faults on a real industrial automotive platform.

1.4 Thesis Structure

This thesis is composed of these main parts:

- Chapter 2 is devoted to explain our experimental environment. This includes the processor simulation tools, the WCET analysis tool and the benchmarks used in this thesis.
- Chapters 3 and 4 describe our proposals for the design of a hard real-time capable multi-core processor. In particular the design of time-predictable on-chip bus, shared cache and off-chip DRAM memory controller.
- In Chapter 5 we propose an interference-aware allocation algorithm to create, given a task set, the task partitions that minimize the resources allocated to each core. All these chapters describe our research on multi-programmed workloads, where we propose a multi-core processor capable of executing mixed-criticality level application workloads and an algorithm to map HRTs to cores.
- Research on parallel applications, where we propose hardware and software mechanisms to support the execution of hard real-time parallel applications is described in Chapter 6.
- Chapter 7 presents the work done to introduce a WCET on-line monitoring unit in an automotive environment.

- Chapter 8 shows the conclusions of this thesis.

CHAPTER 2

Experimental Setup

In this chapter we describe the set of tools we used to evaluate the multi-core processor proposed in this thesis: a cycle accurate execution-driven simulator, a commercial WCET analysis tool and the set of benchmarks used as representative of the HRTs and NHRTs. In addition to that we also describe the experimental setup for the proposal of Chapter 7 which differs from the rest of the thesis. In the last part of this chapter the criteria adopted to evaluate our experiments are addressed.

2.1 Introduction

Simulation is a well known established technique used in both academic and industry research to evaluate new processor architectures. During this thesis we developed a cycle accurate simulator that models a common homogeneous multi-core processor composed by simple cores with private level one instructions and data caches, a shared bus that connects the cores with the shared cache and a shared memory controller that interfaces the processor to the off-chip main memory. The proposals presented in Chapters 3, 4, 5 and 6 have been implemented in this simulator. Moreover we inte-

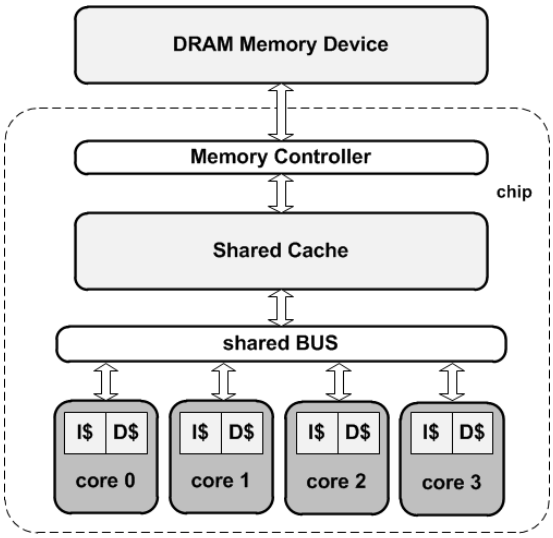


Figure 2.1: Baseline multi-core processor

grated a commercial WCET analysis tool into our simulation environment, required to derive a WCET estimation that is a safe upper bound of the Maximum Observed Execution Times (MOETs).

For the proposals presented in Chapter 7 we used a different simulation infrastructure: our solution was integrated into a gate-level simulation environment of an existing automotive hardware platform.

2.2 Evaluation Tools

2.2.1 Our Multi-core Cycle-Accurate Simulator

In order to evaluate the work proposed in this thesis we developed a cycle-accurate execution-driven simulator compatible with TriCore binaries [45]. The simulator is composed of two main components: a functional emulator, and a timing simulator. The emulator developed based on CarCore[1] [87] is responsible of executing the ap-

[1]CarCore is a System-C simulator developed by the University of Augsburg (Germany) that models in detail a single-core multi-threaded processor.

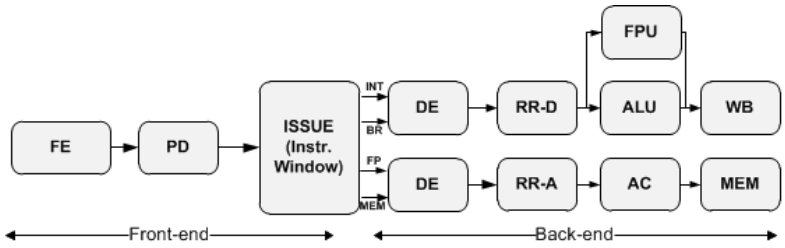


Figure 2.2: Core pipeline details

plications, by maintaining the architectural state of each core in the processor. We enhanced the emulator with the support for floating-point instructions (based on IEEE-754 single precision standard) and for the most common Operating System (OS) calls necessary to perform operations such as open, read, write, close and lseek. This allows benchmarks to work with I/O (file handling), as well as to use printf and similar functions for debugging purposes.

The timing simulator models the micro-architecture of our multi-core processor. In the development of the timing simulator we considered as a guideline CMP-SMTSim [9]: a highly configurable trace-driven simulator, compatible with Alpha binaries, developed by the University of California (San Diego, USA), and Universitat Politecnica de Catalunya (Spain).

To model the DRAM memory system we used the DRAMsim (Version 1) [89], which we integrated inside our simulation framework. DRAMsim is a well known C-based memory system simulator developed by the University of Maryland (USA). It is highly-configurable, parameterizable and implements detailed timing models for different types of existing DRAM memory systems.

We have validated the simulator through a high number of tests. Moreover, the fact that DRAMsim and CarCore emulator have been used by other research groups provides to our simulator higher confidence in the results we have obtained.

The processor model considered along this thesis is a four core processor connected to an off-chip JEDEC-compliant DDR2-SDRAM memory system. The cores are connected through an on-chip shared bus to a shared cache, that is interfaced through a memory controller to the off-chip memory system. A figure of the overall

architecture is shown in Figure 2.1, while the details of the pipelines of each core are shown in Figure 2.2. The front-end of the pipeline includes a Fetch (FE), Decode (DE) and Issue Stages, while the back-end is composed by two pipelines: one for memory operations, and the other one for integer and floating point computations. Both back-end pipelines include a Decode (DE), a Register Read (RR) stages and a Write Back (WB) stages. In addition to that they include a stage for Address Computation, an Arithmetic Logic Unit and a Floating Point Unit.

Regarding the core design, each core implements an in-order dual-issue pipeline inspired from Infineon Tricore [45] and CarCore [87] adding the support for floating point operations. Each core is single threaded, and contains two pipelines: one for memory operations and one for the other types of instructions. No branch prediction is used. The instruction bandwidth between different stages is one instruction. Bypass mechanisms are implemented. As presented in CarCore [87], microcode sequences are used: Call (CALL) and return (RET) instructions use the microcode instructions (i.e. they are split into microcode operations); the additional instructions are inserted by the issue stage. Each core contains private level one instruction and data caches. Regarding the data cache, it implements a write-through write-not-allocate policy: Store (ST) instructions do not stall the pipeline and they access directly the second level cache through the shared bus. A write buffer is used to maintain multiple pending ST. Instead Load (LD) instructions are always blocking the pipeline, being possible to have only one outstanding LD at the time per core.

In particular we chose a shared cache of 128KB in order to have a significant pressure also on the off-chip memory and being able to test also our proposals described in Chapter 4. As local storages we implemented instruction and data caches to avoid the drawbacks of the scratchpads where the programmer or the compiler needs to be aware of the memory regions that are mapped on such memories. However our proposals are independent of the use of level one caches and can work even in presence of scratchpad memories, implementing solutions like the ones proposed in [60] and [61]. Regarding the memory system, in this thesis we model a memory controller interfaced with three different JEDEC-compliant 256Mb x16 DDR2 SDRAM devices: DDR2-

400B, DDR2-800C and DDR2-800E, each composed by a single DIMM, single rank and a single 4-banks memory device. We assume a CPU frequency of 800MHz, being a CPU-SDRAM clock ratio of 2 in case of DDR2-800C and DDR2-800E and 4 in case of DDR2-400B.

Table 2.1 summarizes the architectural details of our processor providing the values of the different parameters that we used for our experiments. The configuration parameters that we used are suggested by the members of the Industrial Advisory Board of the MERASA project [7], and they characterize the requirements of future hard real-time systems.

All the proposals of this thesis are independent of the underlying architecture we have used to carry out the experiments. That is, all new techniques have been designed at the microarchitectural level without considering any special feature of the TriCore ISA. We selected such ISA because the TriCore is one of the most common processor used in real-time systems by industries. It joins the elements of a RISC processor core, a microcontroller and a DSP in one single chip. Even though this thesis considers a single-threaded core, different core designs could be used. For example, in the MERASA project, University of Augsburg proposed a hard real-time capable SMT core [63] [62]. Their proposals are orthogonal to ours and hence can be implemented into our proposed multi-core processor.

In this thesis we do not consider the effect of the Operating System (OS) on the timing behavior of HRTs. In other words, from an execution point of view, we consider HRTs and NHRTs as sequences of instructions executed on the processor, without distinguishing if the instructions belong to the user or the system space. It is interesting to point out that in the MERASA project, University of Augsburg proposed a predictable system-level software [91] that contains functionalities of a Real-Time Operating System (RTOS). The system software guarantees an isolation of multiple HRTs on memory and I/O resources. When isolation cannot be achieved, the system software ensures a time-bounded access to avoid mutual and possibly unpredictable interferences. The intention of this isolation and bounding is also to enable an effective WCET analysis of the application's code.

Table 2.1: Baseline configuration

Parameter	Configuration used
Pipeline depth	7 stages
Number of contexts	1 NHRT or 1 HRT
Pipeline stage latency	1 cycle
Functional unit latency	1 cycle
Fetch Width	2
Instruction Buffer Size	16
Instruction Window Size	16
Physical Registers	32 address, 32 data
Branch Predictor	none
Write Buffer Entries	8
Primary L2 MSHR Entries	10
Secondary L2 MSHR Entries	10
Icache, Dcache	8 Kbytes, 4-way, 1-bank, 8-byte lines, 1 cycle access write-through write-not-allocate policy
L2 cache	128 Kbytes, 16-way, 16-bank, 32-byte lines, 4 cycle access write-back write-allocate policy
Shared Cache Partitioning Techniques	Columnization and Bankization
Instruction/Data L1 miss + L2 hit Latency	9 cycles
Bus Latency	2 cycles
Inserting Bus Arbiter	1 cycle
Scheduling Bus Request	1 cycle
Write back into the Register	1 cycle
Bus Arbiter Queue Size	8 entries
DDRx SDRAM Devices	256Mb x16 DDR2 devices: 400B, 800C and 800E [49]

2.2.2 Experimental Setup of an Automotive Platform

The experiments described in Chapter 7 use a simulation framework different from the one explained in this Chapter. That is, the proposals presented in Chapter 7 were integrated into an internal framework developed by YOGITECH SpA [58] [57].

Such platform is composed by different IPs described in Verilog at RTL and simulated at gate level. The real-platform used for automotive applications includes a fRCPU_armcm3 [40], an IP for on-line fault detection and fault diagnosis, and an ARM Cortex-M3 processor [2].

The platform is simulated injecting permanent faults using a fault-injector tool that was also developed internally by YOGITECH SpA.

2.3 WCET Analysis

In order to verify the timing correctness of a hard real-time application it is necessary to perform WCET analysis, i.e. to compute a WCET estimation. To do so, today industries and academias follow two main approaches for WCET analysis [90]: static analysis and measurements based analysis.

- Static timing analysis, e.g. aiT [42], OTAWA [23], relies on the construction of a specific cycle accurate model of the processor and the construction of a mathematical representation of the timing behavior of the application under analysis running on that processor. The mathematical representation is then processed with linear programming techniques to determine a safe upper-bound on the execution time.

- Measurement-based analysis, e.g. [19], relies instead on thorough testing of the application under analysis on the real processor or a cycle accurate timing simulator of that processor, with high-coverage stressful input data, and recording the longest observed execution time.

In this thesis we use the measurement-based WCET analysis tool RapiTime [8]. RapiTime computes the WCET estimation of a program as a whole probability distribution of the execution time of the longest control-flow path, from which the absolute upper bound (i.e. the WCET estimation) is obtained. To do so, RapiTime first derives an upper bound of the Maximum Observed Execution Time (MOET) for a particular section of code (generally a basic block) from measurements; such execution times

are then combined with the control flow graph to determine an overall estimation for the longest control-flow path through the program. In order to generate the control flow graph, RapiTime automatically instruments the program to be analyzed through annotations. The granularity level is defined by the user and can be set at the level of basic block, function, etc. This information is then processed by RapiTime to rebuild the actual control flow graph and to determine the execution time of the longest path based on measurements.

The complete overview of our simulation environment comprehensive of the WCET analysis tool is shown in Figure 2.3. The source code (.c) of the program is instrumented by RapiTime (it inserts the so called RapiTime_IDPoint). A RapiTime_IDPoint is an identifier introduced by RapiTime and used to identify the basic block currently in execution. The program is then compiled generating the executable file (.elf). When a RapiTime_IDPoint is processed, the timing simulator appends a new trace line into the Rapita Trace file (.rpz) with the RapiTime_IDPoint and the time stamp of the current cycle time. The complete RapiTime trace file (.rpz) is then processed by RapiTime to estimate the WCET.

Even though we used a measurement-based WCET analysis tool into our simulation environment, the proposals of this thesis are independent from the WCET analysis tool adopted and they could also be verified with a static based WCET analysis tool.

2.4 Benchmarks

All experiments conducted in this thesis use several benchmarks suites that are representative of both hard real-time and non real-time domains. All benchmarks have been compiled with Tasking [85], a commercial embedded compiler produced by Altium[2] corporation, using maximum optimization levels. This compiler is the defacto industry standard for TriCore architecture software development.

[2] www.altium.com

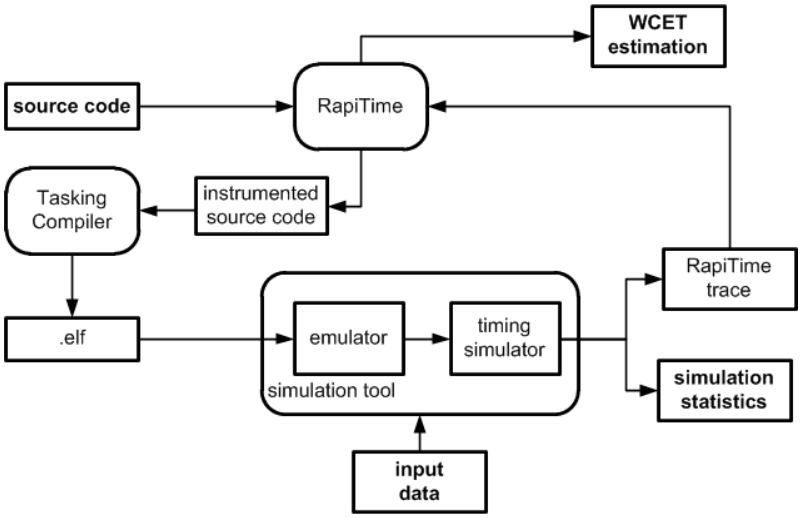


Figure 2.3: Simulation environment including RapiTime

2.4.1 Hard Real-Time Applications

EEMBC Autobench[3] benchmark suite and a real hard real-time application: collision avoidance algorithm, provided by Honeywell Corporation are the HRTs considered within this thesis.

The Collision Avoidance (CA) algorithm provided by Honeywell is based on an algorithm for 3D path planning used in autonomous-driven airplanes to process the frames captured by on-board cameras and to build the path to reach the target points avoiding the obstacles. It requires high-performance with high-data rate throughput and it is strictly hard real-time.

EEMBC AutoBench [77] is a well-known benchmark suite composed by sixteen applications used in both industry and academia that reflect the current *real-world* demands that embedded systems encounter in automotive, industrial, and general-purpose applications. It includes generic workload tests, basic automotive algorithms and signal processing algorithms. Unfortunately, the memory requirements of the EEMBC benchmarks are only around 33 Kilo Bytes each. In order to be represen-

[3] www.eembc.org

tative of future hard-real time applications we increased their memory requirements according to the memory requirements of CA [7], i.e. we increased the number of iterations and the size of the data array without modifying any instruction inside the source code. The new data memory footprint of some of the EEMBC benchmarks is: aifftr01/aiifft01 (64 KB), aifirf01 (72 KB), pntrch01 (62 KB), ttsprk01 (104 KB), tblook01 (70 KB), matrix01 (91 KB).

Benchmark Classification

In order to evaluate how benchmarks impact the different shared resources, we evaluate the pressure that EEMBC and collision avoidance benchmarks have on the on-chip shared resources. To that end, we plot the cache Misses per Kilo Instruction (MpKI) of first level data and instruction caches (shown in Figure 2.4, when varying its size from 4KB to 32KB). Increasing the size of first level cache reduces the amount of requests accessing the shared bus and so, the execution time is reduced due to a smaller delay originated by the contention accessing on-chip shared resources. However, such reduction does not affect uniformly all benchmarks. While there are benchmarks, like aiifft01 and aifftr01, that reduce their accesses to shared resources by more than 22%, others are not affected at all.

On average, EEMBC benchmarks reduce the accesses to the bus and second level cache by 12% when increasing the size of first level cache from 4KB to 32KB. In case of the collision avoidance algorithm such cache-size increment does not affect the MpKI with a consequent reduction by less than 1%. The results of such experiments are shown in Figure 2.4.

As a result of such experiment, we classified the EEMBC benchmarks into two groups, according to the MpKI obtained with a first level cache of 4 KB:

- High bus utilization, formed with benchmarks whose MpKI lay between 180 and 40: aiifft01, aifirf01, pntrch01, cacheb01, puwmod01, idctrn01, bitmnp01 and aifftr01.

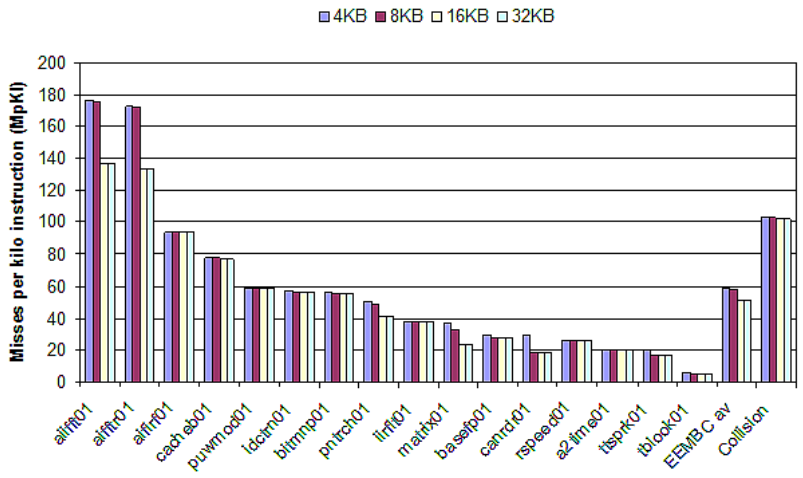


Figure 2.4: Misses per Kilo Instruction (MpKI) of first level cache when varying the cache size from 4KB to 32KB

- Low bus utilization, formed with benchmarks whose MpKI lay between 40 and 0: iirflt01, ttsprk01, tblook01, matrix01, basefp01, canrdr01, rspeed01 and a2time01.

When accessing the second level cache, benchmarks with higher cache utilization could potentially produce higher cache interferences. Figure 2.5 shows the performance speedup when increasing the size of second level cache from 16 KB to 128 KB, taking as a baseline a first level cache size of 8 KB and having a first level cache of 4 KB. Increasing the cache size, the amount of accesses to the off-chip memory reduce and so the performance increase.

According to the results presented in Figure 2.5, the benchmarks are classified into three groups:

- High memory demanding, composed by benchmarks that increase their performance as the cache size increases: aiifft01 and aifftr01.

- Medium memory demanding, composed by benchmarks that increase their performance until a certain cache size is assigned: iirflt01, aifirf01, pntrch01, ttsprk01, tblook01, matrix01 and cacheb01.

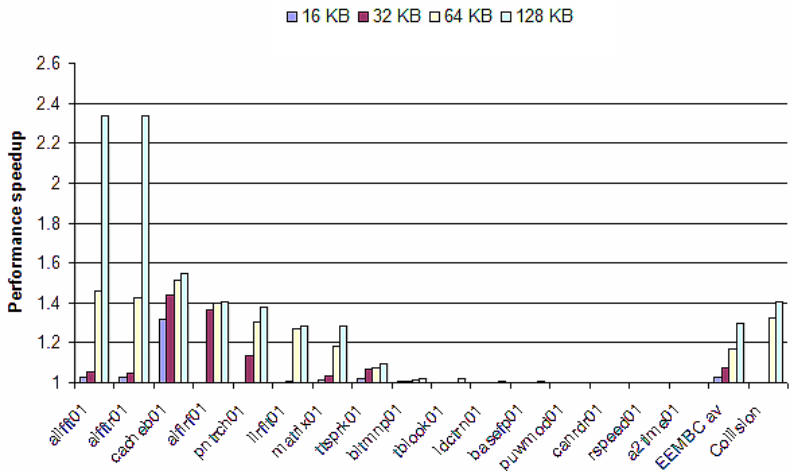


Figure 2.5: Performance speedup when varying the second level cache size from 16 KB to 128 KB, taking as a baseline a second level cache size of 4KB

- Low memory demanding, composed by benchmarks that do not increase their performance as the cache size increases: puwmod01, idctrn01, bitmnp01, basefp01, canrdr01, rspeed01 and a2time01.

According to the results presented in Figure 2.4 and Figure 2.5, the size of the first level cache is fixed to 8 KB per core (4 KB of data and 4KB of instruction caches) and the size of the second level cache is fixed to 128 KB.

To sum up, the on-chip inter-thread[4] interferences effect over the execution time can be classified according to the bus access classification (Figure 2.4), and off-chip cache classification. Hence, taking into account both effects, we classified the EEMBC benchmarks in three groups according to the bus utilization and the memory requirements:

- High shared resource (SR) demanding group, formed by benchmarks with high bus utilization and high memory demanding: aiifft01, aifftr01 and cacheb01.

[4]Please consider that in this thesis we use the words: *application*, *thread* and *task* with the same meaning.

- Medium SR demanding group, formed by benchmarks with small working set, independently of the bus utilization: aifirf01, iirflt01, matrix01 and pntrch01

- Low SR demanding group, by benchmarks with low bus utilization and low memory demanding: a2time01, basefp01, bitmnp01, canrdr01, idctrn01, puwmod01, rspeed01, tblook01 and ttsprk01.

The Worst-Opponent

Another contribution of this thesis is the concept of *worst-opponent*. For a given task, we define *worst-opponent* a task that accesses the shared resources just a cycle before every request produced by the given HRT. This indeed represents the worst-case scenario. It can be used to verify our proposals; in fact the HRTs must meet their deadlines even if running in such worst-case workload.

To design the *worst-opponent* it is necessary to perfectly know the micro-architecture and even though, in some processors could even be the case that is not possible to define such task due to the hardware characteristics.

Hence, in this thesis, we developed a synthetic benchmark that tries to resemble the *worst-opponent* for our multi-core architecture. To that end, since Store operations do not block the pipeline and a write-through write-not-allocate policy is used in the first level cache, all Stores instructions correspond to accesses to the bus and then to the shared cache. For these reasons the *opponent* we built is a *for-loop* of 1000 Stores operations (i.e. to remove the overhead of loop instructions) where the addresses of two consecutive stores are generated in such a way that they target different bank of the shared cache. When performing the experiments for Chapter 4, we modified this synthetic benchmark such that the stores operations always miss in the shared cache and hence they always access the off-chip shared memory.

Parallel Applications

To evaluate the proposals described in the Chapter 6 we used the parallel version of the following applications: *LU decomposition* and *stereo navigation* provided by Honeywell Corporation.

LU Decomposition: In linear algebra, the LU decomposition (also called LU factorization) of the square matrix A is defined as $A = L \times U$, where L and U are lower and upper triangular matrices of the same size respectively, in which L has only zeros above the diagonal and U has only zeros below the diagonal. This decomposition is used in numerical analysis to solve systems of linear equations or to calculate the determinant. We have split the applications into two stages: Given a $n \times n$ square matrix A, the *stage 0* replaces A by the LU decomposition of a row wise permutation of itself. The resultant matrix A' is arranged using the Crout's method. Based on the stage 0's output, A', *stage 1* solves the set of n linear equations $A'X = B$ computing the solution vector X, being B the right-hand side vector.

Stereo Navigation: The stereo navigation application is intended for an aircraft localization in case that the GNSS (Global Navigation Satellite System) used in aircraft is temporarily unavailable and the plane has to localize itself for some period of time. In the latter situation the lack of stereo navigation application responsiveness may have a catastrophic outcome and therefore it is typically considered as a hard real-time application. The application is built on the idea that from two independent images derived from cameras looking in approximately the same direction features can be extracted (dominant entities in the image invariant to rotation and translation). Using two cameras recording images at the same time allows for localization of the features in a 3D-space. Furthermore, from two adjacent image snaps taken in two subsequent time moments t1 and t2 and the change of position of the features in the images, absolute translation $(dT/(t2 - t1))$ and absolute rotation $(dR/(t2 - t1))$ can be inferred. The stereo navigation computation is composed by the 10 phases (rectify, tile, sort, match_lr, match_t1t2, circular_check, reproj, ransac_loop, refin, finit_state). We have split the application in two stages: the two first phases compose the first stage; the rest composes the second stage.

2.4.2 Non Hard Real-Time Applications

MediaBench II, SPEC2006 CPU and MiBench Automotive benchmark suites are the non hard real-time applications considered in this thesis.

MediaBench II[5] is a suite of benchmarks representative of the multimedia and communication industry. This suite reflects the characteristics of most advanced and most widely-used applications. From this benchmark suite we use *h264* coder and decoder, and *mpeg2* coder and decoder.

We also selected some benchmarks from SPEC CPU2006[6] that is an industry standard, CPU-intensive benchmark suite, stressing the processor and the memory subsystem. They are commonly used by high-performance computing community but we consider them also representative of NHRTs. From this suite we used *bzip2*.

MiBench Automotive[7] includes benchmarks representative of embedded control systems. They require performance in math computations, bit manipulation, data input/output and data organization. The benchmarks included in this suite are *basicmath*, *susan_smooth*, *susan_corners*, *bitcount* and *qsort*.

We selected such benchmarks because we consider that they are good examples of common and future NHRTs that are typically executed on hard real-time systems, e.g. the payload functions of a satellite that compress acquitted data, or process signals/images.

2.5 Evaluation Criteria

The criteria that we use to evaluate our proposals are the following:

- The first criterion is to ensure *time analyzability*: to that end, we need to check that the WCET estimation we obtain during the analysis is a safe upper bound of the MOET when running in any workload with other tasks. In other words, we estimate the WCET for a HRT, and then we run it into a workload composed

[5]http://euler.slu.edu/ fritts/mediabench/

[6]www.spec.org/cpu2006/

[7]www.eecs.umich.edu/mibench

by several HRTs verifying that it is safe. In addition to that, we evaluate the *tightness* of the WCET estimations, showing the distance between the WCET estimations obtained and the MOETs. The tighter is the WCET with respect to the MOET, the less the resources allocated to accommodate the worst-case, allowing to execute more tasks (both NHRTs and HRTs) on the same processor. It is also necessary to verify the capability to provide a safe WCET estimation for HRTs while running together with NHRTs. We also evaluate the impact that different resource allocations (like the size of the private cache partition of each HRT) have on the WCET estimations. We show, in other words, how the WCET of a task varies by changing the configuration of the processor.

- The second criterion is to ensure that time composability is satisfied: the WCET estimation must be independent of the workload, i.e.. the WCET estimation does not vary by changing the tasks in the workload. We design our multi-core architecture such that this property is ensured by construction. Further details will be shown in Chapters 3 and 4.

- In our multi-core architecture the NHRTs can use the resources that are not used by HRTs, and our aim is to maximize the utilization of the resources providing the highest possible throughput to the NHRTs. To that end, an additional metric of success is to optimize the performance of the NHRTs. In particular, we choose to optimize the total throughput as it provides a measurement of the performance per resource we can get from the architecture. Anyway, other metrics like weighted speedup $1/N \cdot (\sum_{i=1}^{N} \frac{IPC_i^{CMP}}{IPC_i^{isol}})$ or harmonic mean of relative IPCs defined as $N/(\sum_{i=1}^{N} \frac{SingleIPC_i}{IPC_i})$ can be optimized instead of throughput.

Finally another major goal in the design of our multi-core architecture is the possibility of using the techniques adopted in single-core processors to perform WCET analysis without requiring any change. In this thesis we use RapiTime without any change.

CHAPTER 3

Predictable On-Chip Shared Resources: the Bus and the Cache

In this chapter we present our designs for an on-chip shared bus and a shared cache so that they are time analyzable: regarding the bus we provide the UBD for different arbitration policies and an interference-aware bus arbiter. Regarding the shared cache we show the effect of two types of interferences: storage and bank interferences. To avoid the latter kind of interferences we propose two cache partitioning techniques.

3.1 Introduction

In a common multi-core processor, the different cores are usually connected to shared memories (e.g. cache, SDRAM and DRAM) through an interconnection network. In this thesis, since we focus on a small number of cores, the interconnection network is a *bus*. In the multi-core processor we consider in this chapter each core, with its private data and instruction first level cache, is connected to a second level shared cache through a shared bus. An overall diagram of the multi-core architecture is shown in Figure 3.1.

A bus, used to interconnect several components, is characterized by a *latency*, that is a fixed amount of time necessary for a request to cross it. When one request is granted access to the bus, no other request can use it, so an arbitration policy is required if two requests try to access the bus at the same time. In such scenario, one thread will delay the execution of the other one until it frees the bus, producing a *bus interference*. We first focus on an architecture where each core has a banked private memory and the only shared resource is the bus, and then we describe how to use a shared level of cache. Caches are normally used in high performance processors to reduce the latency of memory accesses; they are generally implemented using multiple banks to allow parallel accesses. However, a bank can only handle one memory request at a time, and so if two memory requests try to access the same bank at the same time a *bank interference* or *bank conflict* is originated. This interference may introduce variability in the execution time of a thread. In addition to that, shared caches have an additional drawback that occurs when one thread evicts valid data of another thread: the so called *storage interferences* .

With those inter-task interferences, even if we were able to redefine the WCET as the longest execution time within all possible workloads, it would be required to analyze a very huge amount of workloads. For example, for a set of n tasks and a target processor with k cores, we should profile all $n/(k!(n-k)!)$ possible combinations. Furthermore, any change in the workload, like a shift in the time at which each application in the workload starts, would invalidate the previous analysis resulting in an unsafe WCET estimation for the remaining threads. This may be often the case when running mixed workload applications with non real-time applications.

In a multi-core environment to estimate the WCET taking into account inter-task interferences it would be required to know beforehand for each task the exact sequence of accesses to shared resources in the worst-case. This is because different access sequences to a shared resource may result in different state of such resource, which leads to different WCET estimations. Unfortunately, it is commonly the case that the sequence of accesses to shared resources is known only at run-time. Moreover, for

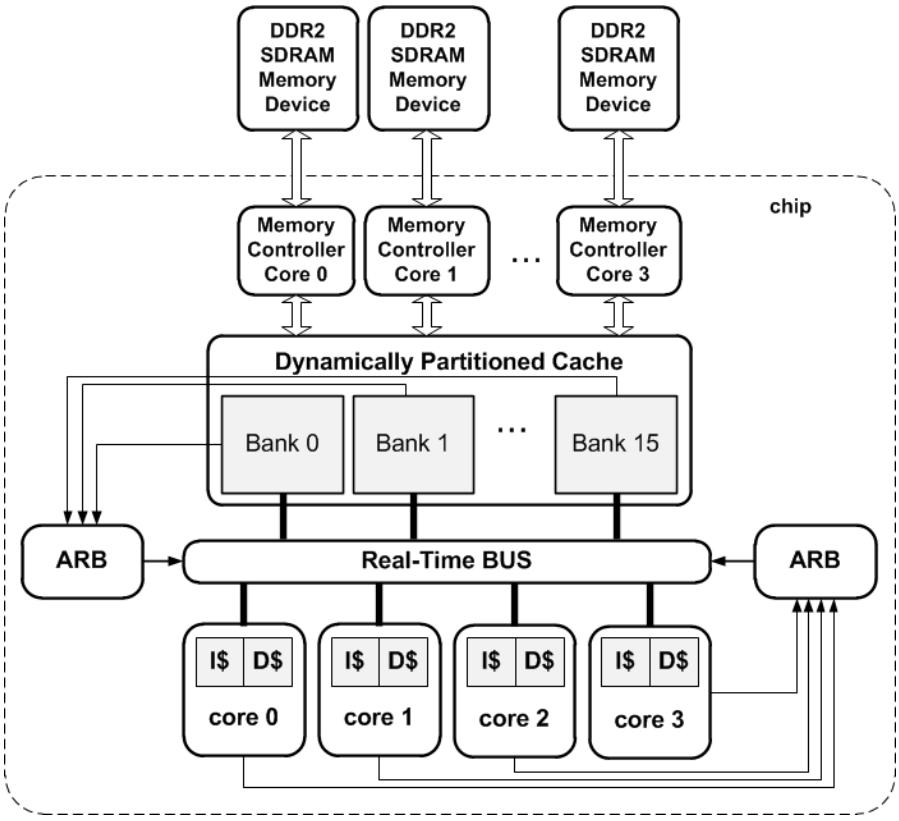


Figure 3.1: Multi-core processor with real-time bus and dynamically partitioned cache

non-real time applications it may be impossible to compute such sequence, mainly if they are programmed using dynamic memory allocation (e.g. pointers).

Let us consider that a HRT T runs in a multicore architecture together with other tasks and it is then necessary to ensure a safe WCET estimation for T. The solution we propose in this thesis avoids profiling all worst-case sequences of accesses of all threads when estimating the WCET. In order to do so, we determine, for every access to a shared resource r, an *Upper Bound Delay* (UBD), that T can suffer due to inter-task interferences. When computing the WCET of T, we assume that the time to access a shared resource r, equals the access time to that resource with no inter-task conflicts plus UBD. As a consequence, the resulting WCET estimation $WCET_{max}$ is a safe upper bound of the execution time of T, running in the multi-core architecture

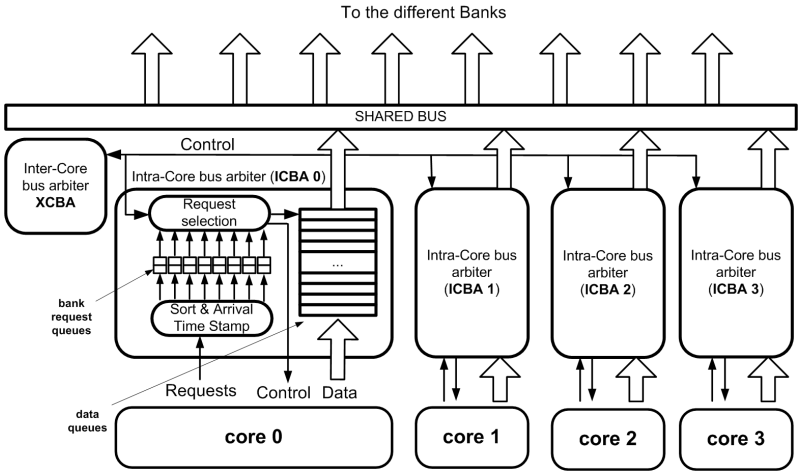


Figure 3.2: Our interference-aware bus arbiter

and sharing the resource r with the other threads. Moreover, given that UBD does not depend on the other threads, $WCET_{max}$ is also independent on the other threads.

In this chapter we focus on predictability issues of both shared resources: the on-chip bus and the shared cache. In particular, regarding the bus, we analyze different arbitration policy, computing the UBD that a request may suffer due to inter-task interferences with the other co-running tasks and propose an interference-aware bus arbiter. Regarding the cache we study the effect of bank interferences on the UDB and we evaluate the impact on the WCET of two different cache partitioning technique necessary to avoid storage interferences.

We recall, from Chapter 2 that in our multi-core processor and along this thesis, we assume that the arbiter needs 1 cycle to select which request accesses the bus (labeled as A in following Figures), 2 additional cycles to send the data through the bus (labeled as B in Figures and L_{bus} in Formulas) and 4 cycles to access a bank (an access to bank n is labeled M_n in Figures and L_{bank} in Formulas).

3.2 Interference-Aware Bus Arbiter

An overall picture of our interference-aware bus arbiter that controls bus interferences when computing the WCET is shown in Figure 3.2 . Our proposal splits the bus arbiter into two hierarchical components: The *Inter-Core Bus Arbiter* (XCBA) that schedules among requests from different cores, and several *Intra-Core Bus Arbiters* (ICBAs), one per core, which schedules among requests from the same core. The idea behind our design is to ensure that the delay that a thread can suffer due to requests of any other thread is bounded by a fixed amount of time.

In our architecture, a thread sends a request to the bus on (1) every data cache load miss, (2) instruction cache miss and (3) store operation. These requests are handled by its corresponding ICBA, which selects the next memory request to be sent to the XCBA. Hence, by maintaining the requests of the different cores apart, the execution time, and so the WCET of a task does not depend on the number of request from the other tasks that are ready and waiting to be granted access to the bus. The XCBA is in charge of selecting which of those requests from different cores access the bus.

In order to accomplish with our second objective of providing high performance, in each ICBA the requests to the cache are placed in different *bank request queues*. There is a bank request queue per second level cache (L2) bank, which holds requests based on their target destination bank. Bank request queues contain the information of the memory request and the index to the data buffer entry that stores all the data to transfer with that request. Thus, once a core sends a request, the ICBA time-stamps it and inserts it into its corresponding bank request queue. The ICBA implements a given policy, which selects the next memory request that is forwarded to the XCBA (that sends it to the bus). In particular, the ICBA applies the following policies among requests from different bank queues. In the case of NHRTs, we allow parallel out of order execution of different cache requests that do not address the same bank in order to increase the overall performance, that is, we apply a *First Ready First Serviced policy*. In the case of HRTs, in order to prevent timing anomalies [56] we apply a FIFO policy, so the oldest request in all bank request queues is selected.

Our ICBA splits wide bus transfers into independent request so they can be sent in non-consecutive bus slots. We allow this way bus transfers wider than the bus bandwidth. A wide bus transfer is completed when the last request has been sent.

3.2.1 Analyzing the Effect of Different Bus Arbitration Policies

The delay a thread can suffer due to bus interferences depends on the bus arbitration policy. In this section we analyze the variation that a shared bus can introduce on the execution time of different tasks. We also show how XCBA enforces that a request from a given task cannot be delayed longer than UBD, and we determine formally UBD's value. Through this section we assume that each task has its own private memory, so tasks are only affected by bus interferences. In Section 3.3 we study an architecture in which tasks share both the bus and the cache.

Scheduling One Hard Real-Time and Several Non Hard Real-Time Threads

In a mixed workload composed of only one hard real-time thread and $N-1$ non hard real-time threads, bus interferences could be avoided by flushing the requests from the NHRT. That is, if the HRT requires the bus and it is being used by a NHRT, the requests from the NHRT can be flushed, so that the bus arbiter immediately grants access to the HRT without introducing any extra delay to its execution time. This technique is too costly in terms of power consumption since it requires re-sending the flushed request and cannot be applied if we run several HRTs simultaneously.

Analogously, if all shared resources are fully pipelined or they have a single-cycle access no interferences occurs between the HRTs and NHRTs. In fact, if a request from a NHRT arrives at the same time as a request from the HRT, the latter is prioritized. While in the case where the request from the NHRT thread arrives one cycle before than the request from the HRT, the latter can proceed as the resource is available in that cycle.

In a more realistic scenario where no flushing technique is used and the access to shared resources takes multiple cycles, the XCBA arbiter prioritizes requests from HRTs on NHRTs in order to minimize the interference of NHRTs on HRTs. Hence, if

a request from the HRT and a request from a NHRT are ready at the same cycle, the arbiter prioritizes the request from the HRT. However, it may happen that the request coming from the HRT arrives just one cycle after the request from the NHRT has been already granted the access to the bus. In such a situation, the request from HRT will be delayed by the request from the NHRT (see Figure 3.3). In this case, the maximum delay that a request from HRT can suffer is upper bounded and can be computed by the following expression: $UBD = L_{bus} - 1$

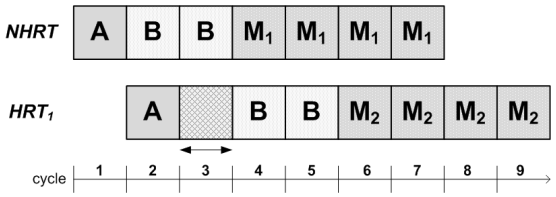


Figure 3.3: Example of interference between a NHRT and a HRT accessing the bus

Scheduling Several Hard Real-Time Threads

In a more realistic scenario, in which we have more than one HRT running at the same time inside the processor, it may happen that two or more requests from different HRTs try to access the bus at the same time. In this case, there is not always an upper bound on the time one HRT can delay the other to access the bus. The existence of such an upper bound depends on the arbitration policy. We are going to use three different XCBA arbitration policies for illustrative purposes. *Thread Prioritization* always gives priority to requests that are generated from the highest priority HRT (or a set of HRTs). *Round Robin* assigns the same priority to all the requests from HRTs. Finally, *FIFO* prioritizes the requests in arrival order to the arbiter.

When a *thread prioritization* is used the maximum delay a thread can suffer is not bounded. As shown in Figure 3.4, in cycle 0, a request from each HRT is ready. In cycle 1, the arbiter prioritizes requests from HRT_1, so HRT_2 is stalled until HRT_1 finishes. However, before leaving the bus another request from HRT_1 becomes ready in cycle 2. In this situation, the amount of time HRT_2 needs to wait to get access to the bus depends on the total time the HRT_1 has requests ready. Thus, the UBD that

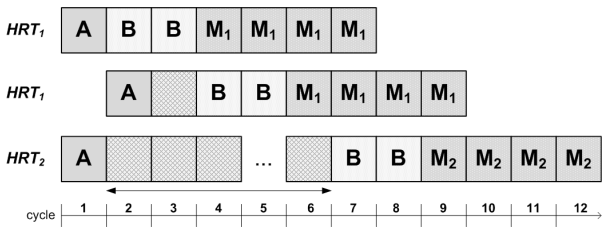


Figure 3.4: *Thread prioritization*. The delay that HRT_1 produces over HRT_2 is unbounded

HRT_2 suffers due to interferences with HRT_1, depends on HRT_1. Even though this UBD can be computed knowing HRT_1 sequence of accesses, it can be too long/pessimistic to be useful.

With *round robin*, the maximum delay a request from a HRT can suffer is bounded by the total number of HRTs that can send a request at the same time. Figure 3.5 shows an example of such worst-case scenario that occurs when two requests from two different HRTs become ready at the same time. In this case, a given HRT, let's say HRT_2 must wait until the previous request from HRT_1 finishes. The maximum delay HRT_2 suffers due to other HRTs, is: $UBD = (N_{HRT} - 1) \cdot L_{bus}$, N_{HRT} is the number of HRTs running at the same time in the processor, which is upper bounded by the number of cores.

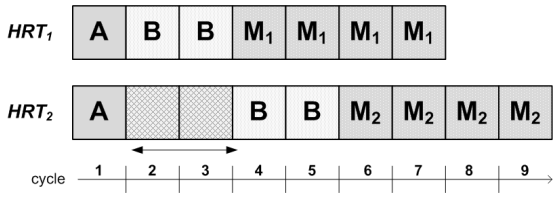


Figure 3.5: Worst-case scenario between two HRTs

Finally, if a *FIFO* policy is applied, the maximum delay a request from a HRT can suffer is bounded by the total number of HRTs, that can send a request at the same time, times the number of entries in the request queues each thread is allowed to have.

Overall Effect of Bus Arbitration

Our multi-core uses a round robin bus arbitration policy between HRTs and prioritize them over NHRTs. The maximum delay is determined by the combination of the effects of the requests coming from HRTs and NHRTs:
$UBD = L_{bus} - 1 + (N_{HRT} - 1) \cdot L_{bus}$. It can be simplified as follows:

$$UBD = N_{HRT} \cdot L_{bus} - 1 \tag{3.1}$$

We want to highlight that the N_{HRT} is the number of HRTs running at the same time inside the processor (and not the total number of HRTs that form the system), which is upper bounded by the number of cores.

Therefore, by using *round robin* policy the maximum delay that a request will suffer due to bus inferences does not depend on previous knowledge of the workload (task set), but only on the total number of HRTs that are going to be executed simultaneously inside the multi-core processor. Moreover, the WCET analysis of each thread can be performed in isolation, since by design our architecture ensures that the UBD of Formula (3.1) is never going to be violated. The requests to the bus from a HRT will never be delayed longer than UBD due to the interactions with the other threads, regardless of the workload.

In conclusion, to guarantee a bounded inter-thread interference delay when running in a mixed application workload, the XCBA applies the following policy to select the next memory request that will access the bus. First, the requests from HRTs have priority over requests from NHRTs. Second, between different requests from HRTs, a round robin policy is applied as well as between different requests from NHRTs. Having more than one pending request from the same thread it does not affect inter-thread interferences but it is a concern of single-core WCET analysis [19].

Although not shown in Figure 3.2, our bus is full-duplex and the same principle is applied for the bus arbiter that controls the requests that go from second level cache to the corresponding core, i.e., load misses and instruction misses. Hence, an ICBA for each core, as well as a global XCBA is required to control the request from L2 to cores. In this case the L2 banks insert the request into the ICBAs while the cores are

the destinations of those requests. Notice that requests from cores to L2 banks do not interact with requests from L2 banks to cores and vice versa.

3.3 Analyzing the Shared Cache

In this section, we consider a more realistic multi-core scenario in which threads can suffer interference delays from two shared resources: bus and second level cache. Such architecture is shown in Figure 3.1. The bus acts as the connection between cores and L2 banks. The use of shared memories in multi-core systems introduces unpredictable and not analyzable worst-case behavior due two factors: *bank access interference* and *storage interference*. In this section we will focus on addressing both problems, enabling multi-core processors to become analyzable.

3.3.1 Bank Access Interference

Caches are normally partitioned into multiple banks to enable parallel operations, i.e., different memory operations can access different banks simultaneously. However, a bank can only handle one memory request at a time. When a bank is serving a memory request, it is inaccessible to any other request for an amount of cycles equal to the bank latency. So, if two memory requests try to access the same bank at the same time, the bus arbiter avoids any conflict by delaying the second access. This kind of effect, called *bank interference* or *bank conflict*, may introduce variability in the execution time of a thread.

An example of a bank conflict is shown in Figure 3.6. Two threads, HRT_1 and HRT_2, want to access the same memory bank (labeled as M_1) at the same time. Since we assume a memory latency of 4 cycles, HRT_2 turns out to be delayed 4 cycles because of a previous request from HRT_1.

In order to control the execution time variation caused by bank interference, we apply the same principle used to avoid bus interference, i.e., determining the maximum delay a memory request can suffer because of bank interference. Assuming the same arbitration policy presented in Section 3.2.1, the UBD is determined combin-

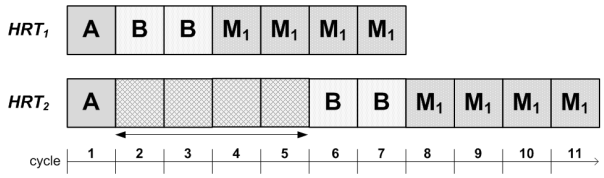


Figure 3.6: Bank conflict example between two HRTs

ing the effects of the requests coming from HRTs and NHRTs. On the one hand, the maximum delay a request from a HRT request can suffer because of NHRTs appears when the former arrives just one cycle after the latter was granted the bus (if both arrive at the same time, the request of the HRT has priority and so it does not suffer any delay). In this case the maximum expected delay is: $UBD = L_{bank} - 1$.
On the other hand, the maximum delay that a request from a HRT can suffer because of other HRTs occurs when it must wait until all other HRT requests finish. In this case the maximum expected delay is: $UBD = (N_{HRT} - 1) \cdot L_{bank}$. Hence, by combining both effects, the maximum delay results in $UBD = L_{bank} - 1 + (N_{HRT} - 1) \cdot L_{bank}$. This can be simplified as follows:

$$UBD = N_{HRT} \cdot L_{bank} - 1 \tag{3.2}$$

As in Formula (3.2), N_{HRT} is the number of HRTs running at the same time inside the processor.

Notice that, as it is commonly the case, a bank access takes longer than accesses to the bus: We consider a L_{bus} of 2 and a L_{bank} of 4 cycles. Hence, the bus latency is overlapped when accessing the bank, as shown in Figure 3.6 (cycles 6 and 7). For that reason, the bus latency does not appear in the formula. However, if the bank latency would be smaller than the bus latency, the bank conflict effect would be hidden because the time required to access a bank would be overlapped by the bus latency. In general Formula (3.2) can be expressed as $UBD = N_{HRT} \cdot max(L_{bank}, L_{bus}) - 1$

3.3.2 Storage Interferences

Storage interferences appear in shared memory schemes when one thread evicts data of another one, potentially delaying the execution time of the second thread. Such time variation makes WCET estimation harder or even infeasible.

Cache locking [78] helps to make caches more analyzable. This technique provides hardware support in order to allow the software to control which cache lines can not be modified by the replacement policy, locking the most frequently used cache lines, and reducing storage interferences. However, this technique requires knowing the whole memory footprint of a thread, being hard to implement in multi-cores. In such case it is necessary to consider all threads that can be co-scheduled into the processor at the same time, in order to prevent that two tasks lock the same lines at the same time. Caches using locking techniques have been shown to have similar behavior of scratchpad memories [78].

Cache partitioning is a well known technique that eliminates completely storage interferences by splitting the cache into private portions, each assigned to a different thread. Our mixed workload environment can benefit from cache partition: Storage interferences between HRTs are avoided by assigning them different partitions of the cache, while non real-time threads can share the same part of the cache. In this thesis, we study two different cache partitioning techniques controlled via software: *columnization* and *bankization*, and their effect on the WCET computation.

In *columnization* [28] the cache is partitioned into ways, giving to each thread a subset of the total number of ways that no other thread can use. In fact this technique only varies the replacement policy: a thread evicts data only in the assigned ways. Cache partitions at level of ways can be implemented with column caching [28]: a bit vector specifies the set of columns (ways) assigned to a given thread. The replacement algorithm is modified to limit replacement to the columns specified by the bit vector.

In *bankization* the cache is partitioned into banks, giving to each thread a subset of the total number of banks that no other thread can use. In any cache access it is required to remap the destination bank of a memory request to one of the banks assigned to the thread. The additional hardware necessary to implement bankization,

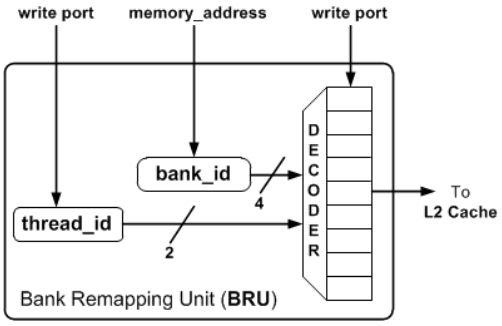


Figure 3.7: BRU

is based on a *Bank Remapping Unit* (BRU) that computes the target L2 bank given the thread identifier and the memory address, as shown in Figure 3.7. The information regarding the destination bank of any memory request is contained inside its address. A given range of bits of the memory address is used to select the destination L2 bank. Since bankization assigns a subset of the L2 banks to a given thread, it is necessary to remap the destination bank of a memory request to one of the banks assigned to the thread. The BRU performs this remapping, i.e., it determines the new destination bank of a memory request. The table inside the BRU (see Figure 3.7) is updated by the RTOS based on the subset of banks assigned to each thread. The remapping table is indexed by the thread id and the original L2 bank id of a memory request. The BRU output is the new bank id.

The main difference between columnization and bankization is that columnization prevents only storage conflicts since different threads can still access to the same bank. As a result, the UBD to use with columnization is the one shown in Formula (3.2). Meanwhile, with bankization we prevent both storage and bank access conflicts, so the UBD to use is given by Formula (3.1). Hence, bankization provides tighter WCET estimation than with columnization. A detailed comparison between columnization and bankization is done in Section 3.5.2.

Inputs		Output
NHRTs	nHRTs	UBD
-	0	0
0	1	0
0	2	4
0	3	8
0	4	12
1	1	3
1	2	7
1	3	11
1	4	15

Figure 3.8: UBD values

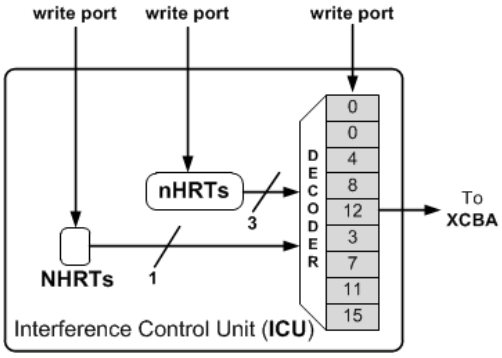


Figure 3.9: ICU

3.4 Computing a Safe WCET Estimation on multi-core Processors

So far we have shown how our multi-core architecture enforces a given UBD that a thread can suffer due to interferences with other threads accessing on-chip shared resources. In particular we have computed the UBD due to bus and cache bank interferences, ensuring that, regardless of the workload any request of a HRT will never be delayed longer than the UBD. This is a necessary feature to make a multi-core architecture time analyzable.

In this section we propose a novel hardware feature: the *WCET Computation Mode*. The *WCET Computation Mode* allows computing safe WCET estimations of HRTs that are going to be executed simultaneously with other tasks on the multi-core architecture. Our multi-core processors has two execution modes: *WCET Computation Mode* and *Standard Execution Mode*. Our processor is set in the *WCET Computation Mode* when computing a WCET estimation for the HRT and in *Standard Execution Mode* otherwise.

3.4.1 The WCET Computation Mode

When analyzing a set of HRTs, the processor is set into *WCET Computation Mode* and each HRT is run in isolation. In this execution mode, the processor delays the execution of every request to a shared resource by UBD cycles. That is, once both, the request from the HRT and the shared resource (in our case the cache and the bus) are ready, the XCBA *freezes* that request by UBD cycles. By doing this, the XCBA artificially introduces the maximum delay that a request from HRT can suffer because of inter-thread interferences, that is the UBD. Hence, the execution profile that results executing the HRT under the *WCET computation* mode takes into account the worst-case delay that the HRT can suffer due to inter-task interferences. This execution profile is passed to RapiTime (our WCET analysis tool) that computes a safe WCET estimation without any single change in the tool.

Once a WCET estimation has been obtained for each HRT, the processor is set back to *Standard Execution Mode*, in which no artificial delay is introduced. Our bus arbiter ensures that the execution time of a HRT that runs in a given workload formed by N HRTs running at the same time, will not be longer than its corresponding $WCET_N$. When running in *Standard Execution Mode* instructions accessing shared resources are executed before their estimated WCET, as it is not always the case that they suffer an inter-task interference. In [14, 83] it has been formally proved that executing an instruction before its estimated WCET, ensures that the WCET estimation derived running in *WCET Computation Mode* is safe. As a consequence, the WCET estimation provided by RapiTime, is a safe upper bound of the execution of the HRTs when they run in the multi-core processor sharing resources with other tasks.

The UBD artificially introduced by our *WCET Computation Mode* is computed using Formula (3.1) or (3.2). The UBD depends on the total number of hard real-time threads running at the same time in the processor (N_{HRT}), that is upper bounded by the number of cores. Thus, depending on the number of HRTs the analyzed thread is going to be co-scheduled with, a different UBD value is used by the *WCET Computation Mode*, resulting in different WCET estimation values. In general, we say that a

HRT that is co-scheduled at the same time with N HRTs, is analyzed using a *WCET Computation Mode* of N, which results in a WCET estimation $WCET_N$.

Our *WCET Computation Mode* allows analyzing each HRT in isolation, i.e., independently from the particular task set in which that task is going to be scheduled. For every HRT we build a WCET-matrix. The WCET-matrix has as many entries as WCET-computation modes times the number of cache partitions a thread can be assigned. In our baseline, we have a total of 25 configurations, 5 WCET-computation modes (including a configuration that disables it) times 5 cache configurations (assigning a power of 2 cache size to each HRT). This WCET-matrix can be computed in *isolation* for each HRT. This process can be easily automated, in fact we do so to run the experiments for this thesis. The next step is to provide the WCET-matrix to the schedulability algorithm, which selects the best allocation of resources for the tasks in the task set. In Chapter 5, we elaborate more the schedulability issues of our proposal.

This WCET Computation Mode can be easily adapted to other processor shared resources as long as it is possible to compute an Upper Bound Delay of the inter-thread interferences. Chapter 4 describes how we adopted the WCET Computation Mode to the off-chip memory system.

Hardware Implementation

The *WCET Computation Mode* requires extra hardware in the XCBA to store all possible UBD values (for our architecture they are shown in Figure 3.8) when analyzing HRT running in a *WCET Computation Mode* of N. To do so, we introduce the *Interference Control Unit* (ICU), shown in Figure 3.9, that contains all precomputed UBD values corresponding to each N-*WCET Computation Mode*. Moreover, in order to generate a tighter WCET estimation, we also include inside the ICU the UBDs that do not take into account the NHRTs. Hence, since N is bounded by the number of cores, the size of the ICU is limited to $2 \cdot N_{cores}$.

To sum up, the *WCET Computation Mode* works as follows. According to the number of HRTs (*nHRTs*) and whether there are NHRTs, the corresponding UBD is

forwarded to XCBA. Then, the XCBA inserts such value into a down-counter that is reset every time a new request is ready. When the counter reaches zero, the request is sent through the bus, effectively delaying each request by UBD cycles. To disable the *WCET Computation Mode* and run the processor in *Standard Execution Mode*, it is necessary to set to zero the *nHRTs* register. An example of an ICU in a 4-core architecture using columnization and a bank latency of 4 cycles is shown in Figure 3.9. For example, when analyzing a HRT that is going to be co-scheduled with 2 more HRTs and one NHRT, the UBD required is: $UBD = N_{HRT} \cdot L_{bank} - 1$ because there is a NHRT. Thus, being $N_{HRT} = 3$ and $L_{bank} = 4$ this results in a UBD of 11 cycles. ICU can be set either by the RTOS or even by the processor vendor since it depends on the architecture.

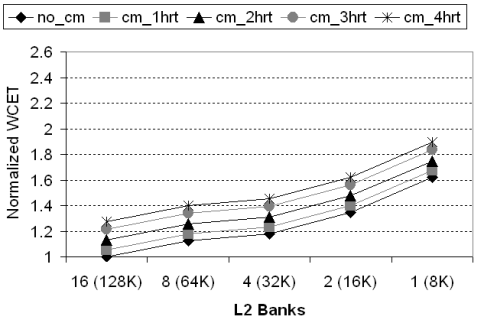
3.5 Results

This section evaluates the effect of *WCET Computation Mode* on the on-chip shared bus and it compares the two cache partitioning techniques: Bankization and Columnization. All WCET estimations have been obtained using RapiTime and normalized to the WCET estimation when the hard real-time task runs in isolation with all the hardware resources in the multi-core architecture and the *WCET Computation Mode* disabled.

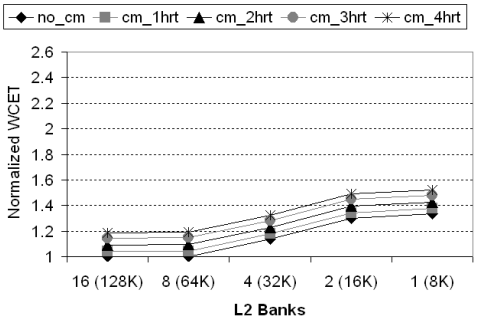
3.5.1 WCET Evaluation

To run our experiments we used the HRTs described in Chapter 2. For each benchmark we compute a WCET estimation varying the *WCET Computation Mode* and the amount of cache assigned to each of them. Bankization is used as a cache partitioning technique. In Figure 3.10 we show the normalized WCET average between all the benchmarks belonging to the same *high*, *medium* and *low* demanding group.

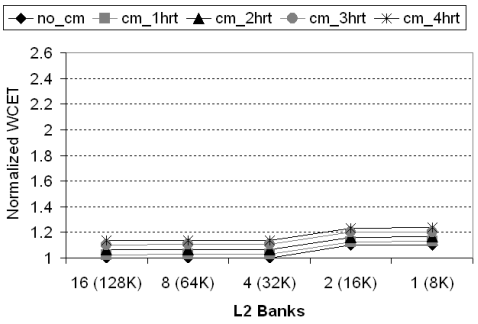
Notice that in all cases, the WCET estimation increment when varying the *WCET Computation Mode* from 1 to 4 is almost the same regardless of the cache size given to



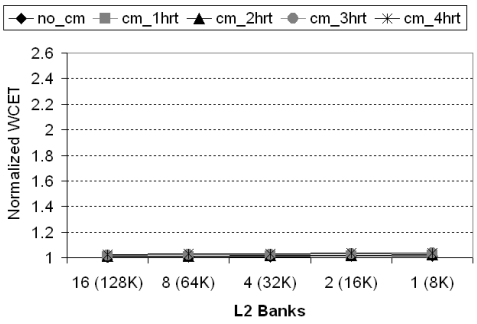
(a) High demanding HRTs (bankization)



(b) Medium demanding HRTs (bankization)



(c) Collision Avoidance App. (bankization)



(d) Low demanding HRTs (bankization)

Figure 3.10: WCET estimation of different HRTs (using bankization)

the task. This is because the bus is accessed before the cache access, so the introduced delay is independent of the cache configuration used.

High demanding benchmarks, shown in Figure 3.10a, are very sensitive to both *WCET Computation Mode* variation and cache partition size reduction. Varying the *WCET Computation Mode* from 1 to 4 and fixing a cache size, the WCET estimation increases from 5% to 27%. When reducing the cache size from 64KB to 8KB the WCET estimation increases with respect to using the whole cache (128KB) from 12% to 62%. Hence, running with 4-*WCET Computation Mode* and 8KB cache size, the WCET estimation increases 89% in comparison of not using this mode and having

the whole cache. Such increment comes from the contribution of *WCET Computation Mode* (27%) and the cache size (62%).

Medium demanding benchmarks, shown in Figure 3.10b, have a smoother behavior with respect to *high* demanding ones. The *WCET Computation Mode* increases the WCET estimation from 3% to 18% when varying it from 1 to 4 respectively. The cache size increases the WCET estimation from 13% to 34% when reducing it from 32KB to 8KB respectively (notice that there is no degradation in performance when reducing the cache size from 128KB to 64KB). Hence, when running with 4-*WCET Computation Mode* and 8KB cache size, the WCET estimation increases 52% in comparison of not using *WCET Computation Mode* and having the whole cache.

Low demanding benchmarks, Figure 3.10c, increases the WCET estimation 2% when running with up to 4 *WCET Computation Mode*, and it has not effect on WCET estimation when reducing the cache size.

The collision avoidance algorithm provided by Honeywell, Figure 3.10d, resembles a *medium* demanding benchmark. The *WCET Computation Mode* increases the WCET estimation up to 14% when running in 4 *WCET Computation Mode*, and the cache size reduction increases the WCET estimation up to 10% when having a 8KB cache. This results in an overall WCET estimation increment of 24% in comparison of not using this mode and having the entire cache.

Unlike single-core scheduling techniques where only one WCET estimation value per cache size is required, a set of WCET estimations are computed using our *WCET Computation Mode* technique, resulting in the *WCET-matrix* presented in Section 3.4. As explained, this matrix is given to the scheduling algorithm that looks the best scheduling. The values presented in Figure 3.10 results in a WCET-matrix of 25 entries, five cache sizes and four *WCET Computation Mode*. Moreover, since each WCET estimation is independent of the workload, any change in a WCET-matrix of a task does not affect any matrix of other tasks.

3.5.2 Bankization vs Columnization

The previous subsection uses bankization as cache partitioning technique to evaluate the WCET increment when using our *WCET Computation Mode*. In this section we compare bankization and columnization in terms of WCET estimation variation and implementation complexity.

Figure 3.11 compares bankization (labeled with *B*-) and columnization (labeled with *C*-) in terms of WCET estimation increment for *high*, *medium* and *low* demanding tasks and the collision avoidance algorithm, when varying the *WCET Computation Mode* from 1 to 4. All the values (in case of bankization and columnization) are normalized to the same WCET estimation obtained running the HRTs with the whole cache (i.e., there is no difference between bankization and columnization) and *WCET computation mode* disabled.

Columnization prevents only storage conflicts since different threads can still access to the same bank. As a result, the UBD to use with columnization is the one shown in Formula (3.2). Meanwhile, with bankization we prevent both storage and bank access conflicts, so the UBD to use is given by Formula (3.1). Hence, bankization provides tighter WCET estimation than with columnization. For *High* demanding tasks (squares in Figure 3.11) the use of columnization increases their WCET estimation from 4% to 16% when varying the *WCET Computation Mode* from 1 to 4 respectively. In case of *medium* demanding tasks (circles) such increment varies from 2% to 9%. Finally, for *low* demanding tasks (diamonds) the WCET estimation increment is up to 1.6% when running with 4-*WCET Computation Mode*. Collision avoidance algorithm (labeled as Hon), has an increment from 2% to 18% when varying the *WCET Computation Mode* from 1 to 4 respectively.

Even though it is clear that columnization involves a bigger WCET estimation than bankization, bankization has an important drawback: It requires bigger cache area and additional hardware in comparison to columnization. Such increment comes from two sides. First, a BRU (see Figure 3.7) is required to remap the destination bank. Second, since the number of banks assigned to a given thread is not fixed, the number of bits that form the memory address tag cannot be fixed. It is then required

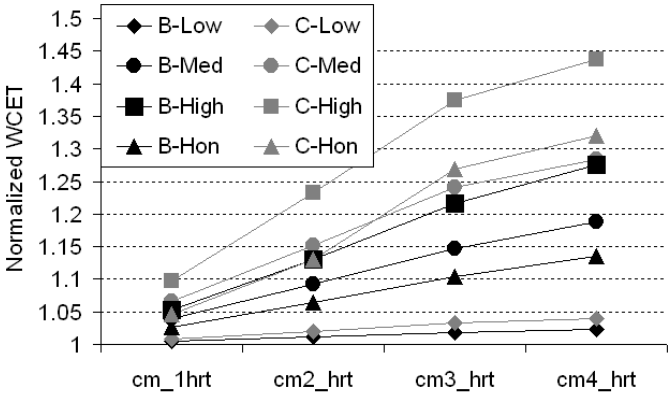


Figure 3.11: Bankization vs Columnization

to reserve space for the maximum tag size, that is when only one bank is assigned to a thread. In our architecture the cache area is increased by a 3% in comparison to columnization.

Depending on the allowed WCET estimation increment and the amount the hardware available a designer of an embedded, hard real-time system can choose one of the two alternatives discussed in this section

3.5.3 Mixed-Application Workload Evaluation

The first objective of our architecture is to ensure that HRTs finish before their deadlines, which can produce a performance degradation of NHRTs. For example, high resource demanding HRTs require reserving a significant part of the cache. As a consequence, when NHRTs are co-scheduled with high-demanding HRTs, NHRTs will be allowed to use less resources than when they run with low demanding HRTs. In this section we analyze the performance, IPC throughput, we obtain for the NHRTs when they run with other HRTs.

We composed 4-thread workloads with 2 HRTs and 2 NHRTs. We use HRTs with different resource demands: *high*, *medium* and *low* demanding groups (labeled as *H*, *M* and *L* respectively). As NHRTs we use benchmarks from MediaBench, MiBench

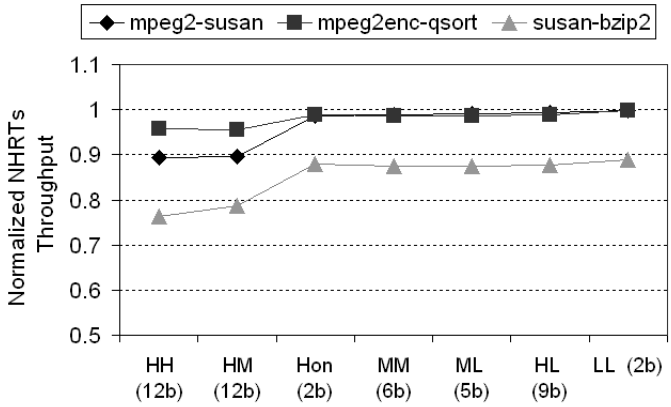


Figure 3.12: Normalized throughput of NHRTs. The numbers in parenthesis are the number of L2 banks required by the HRTs

and SPEC CPU 2006 grouping them as: *mpeg2dec - susan*, *mpeg2enc - qsort* and *susan - bzip2*.

Figure 3.12 shows the throughput of the NHRTs when they run with other HRTs. The throughput is normalized to the case when the NHRTs run with no other HRTs at the same time. In these experiments we have assumed that each HRT has an utilization of 20%, that is that its deadline is only 20% higher than its WCET estimation when it runs in isolation ($d_i/WCET_i = 1.2$). In order to accomplish with this time requirement in the x-axis of Figure 3.12 we show the number of L2 banks that the HRTs must reserve. The number of banks given to the NHRTs is 16 minus the number of banks given to the HRTs, where 16 are the number of banks L2 has in our baseline architecture. In all our experiments, the HRTs finished before their deadlines. In all workloads we observe that, obviously, when the HRTs require less L2 banks to reach their deadline, Low demanding type of HRTs, the NHRTs run faster. In the case of *mpeg2dec-susan* and *mpeg2enc-qsort* benchmarks, they reach an increment of 10% and 5% respectively between LL and HH HRTs, obtaining a throughput of 1 in case of LL (i.e., the maximum we can achieve, because it means we are achieving the same performance of when there are not HRTs in the workload). These results show that our multi-core can execute HRTs meeting deadlines and provide high performance to

Table 3.1: WCET estimation for some EEMBC benchmarks

Task	Resource demand	WCET in isolation (processor cycles)	Normalized WCET in WCET Computation Mode n			
			$WCET_1$	$WCET_2$	$WCET_3$	$WCET_4$
aifftr01	High	9.09×10^8	1.06	1.15	1.26	1.33
a2time01	Low	6.66×10^8	1.001	1.001	1.002	1.003
tblock01	Low	6.40×10^8	1.001	1.001	1.002	1.004

NHRTs. In the case of *susan-bzip2* the variation in the throughput is also 10% but in case of LL the normalized throughput is 0.88, i.e., they do not reach 1. This is because the resources used by HRTs slow down the NHRTs but still they achieve high performance. In general, the performance that NHRTs achieve depends on their cache utilization. If they are high-demanding they will be more affected by the use of cache the HRTs do and vice versa.

Unlike the cache, the bus is not reserved for the HRTs. The bus arbiter just prioritizes the requests from the HRT over the requests of the NHRTs. When the bus is not used by HRTs, NHRTs can use it. The use of the bus done by the HRTs, depends on the particular HRTs.

3.6 Grouping Technique

The *WCET Computation Mode* allows analyzing each HRT in isolation, i.e., independently from the particular task set in which that task is going to be scheduled. Let's define $WCET_k^{taski}$ the WCET estimation for $task_i$ when it runs in *WCET Computation Mode* k. So far we have assumed that all HRTs have the same priority. That is, in the bus scheduling we use a round robin policy among HRTs, and they have priority over NHRTs. In this scenario, if we schedule several NHRTs and M HRTs at the same time, the execution time of each HRT is upper bounded by a WCET estimation, $WCET_M^{taski}$. However, as shown in Figure 3.10, the WCET of tasks with high resource demands is sensitive to the *WCET Computation Mode* in which the task runs. That is, the WCET estimation rapidly increases as we increase the *WCET Computation Mode* in which we run the task. Meanwhile, tasks with low shared resource

demands are almost insensitive to the *WCET Computation Mode* used (less than 2% in the worst-case). To maximize the utilization of the processor, we propose to divide the HRTs into groups. The bus arbiter applies a round robin priority among groups. Inside each group the arbiter also applies a round robin policy. In general, we create g groups of HRTs and in a given group we place n tasks. In this scenario, each task in such group uses a WCET estimation $WCET_{g \cdot n}^{task_i}$. That is, a thread in a group with n tasks, has to use the *WCET Computation Mode* k, where k equals the total number of groups times the number of HRTs in its group. Hence, tasks in populated groups use a higher *WCET Computation Mode* than tasks in smaller groups.

Let's assume we want to schedule several NHRTs with the following EEMBC HRTs: *aifftr01*, *a2time01* and *tblock01*. As shown in Table 3.1, *aifftr01* is a high demanding benchmark, e.g. its WCET is 33% higher when run in *WCET Computation Mode* 4 with respect to its WCET in isolation. Meanwhile, *a2time01* and *tblock01* are two benchmarks with low resource demands.

If we use a round robin policy, we have to use $WCET_3$ for each task, which is high for *aifft01*, 26%. However, if we put *aifftr01* alone in a group and *a2time01* and *tblock01* in another group, each request from *aifftr01* may suffer at most a delay accessing the bus and cache equal to the delay when running in *WCET Computation Mode* 2, which leads to a $WCET_2^{aifftr01} = 1.15(15\%)$, meanwhile *a2time01* and *tblock01* has to run in *WCET Computation Mode* 4, which leads to an increment of their WCET less than 0.5% ($WCET_4^{a2time01} = 1.003$ and $WCET_4^{a2time01} = 1.004$). In this way, the scheduling algorithm can take full advantage of using grouping by considering the resource demands of each HRT, which can be determined analyzing the WCET-matrix of each HRT. Tasks with high resource demands can reduce its WCET estimation by placing them into different small groups, while putting all threads with low demand of shared resources into a single group.

The grouping technique requires small hardware modification to XCBA: Instead of applying a round robin policy between the HRTs, the round robin is applied among different groups and among threads inside each group. Thus, XCBA requires ad-

ditional information: The total number of groups and the group associated to each thread.

3.7 Related Work

Some works already deal with the problem of analyzability in the presence of some shared resources [11, 36, 72, 83].

In [83] Rosen et al. described a solution to implement predictable real-time applications on multiprocessors. They propose a bus scheduling policy based on TDMA (Time Division Multiple Access) based on a previously statically defined scheduling policy. Different time-slots to access the bus are allocated to different processors by static scheduling, i.e., stored in a memory directly connected to the bus arbiter. This technique needs to know the workload a priori, which is the whole set of tasks that run on the system at any given time, in order to avoid situations where the bus contention increases the memory access latency. This solution prevents any deadline miss due to bus conflicts. The architecture, which is described in [53] for a real-time biomedical monitoring and analysis system, is a multi-core processor where each core has its own private memory, connecting all of them with a bus.

The Real-Time Virtual Multiprocessor (RVMP) architecture [36] virtualizes a single in-order super-scalar processor into multiple interference-free different-sized virtual processors. The configuration of the virtual processors can be changed at runtime according to the timing requirements, providing a timing analyzable architecture together with the flexibility of SMT processors. The processor partitioning is determined statically by the real-time scheduling framework that preserves the possibility of analyzing the WCET as in single-cores. The architecture does not include any cache to reduce the level of non-determinism. In this work, the SMT is assumed to have fully-pipelined functional units so that every clock cycle a new access to a shared resource can be performed without any interference.

In [37] the authors propose a real-time multithreading framework, that can be applied on a switch-on-event single-core multithreaded processor. According to the

same authors ([36]), the solution is limited to scalar pipelines with only one of the hardware threads selected for execution on the pipeline at the time. The main characteristic of this processor is that a thread cannot overlap its computation and its access to memory. Instead the memory access of one thread can be overlapped with the computation of other threads. This model cannot be applied in our multi-core approach, in which multiple threads run in parallel. In [36, 83] it is further assumed that each task has a private piece of memory on chip, either a cache or a scratchpad. In this thesis, instead of an interference-free architecture we focus on an architecture in which threads can compete for the hardware shared resources, providing in this way high performance. Moreover, for the HRTs we provide a WCET estimation that is independent on the task set in which that HRT is executed.

In [21,72] the authors describe different aspects of accessing shared resources taking into account cache memories. In [72] they deal with interferences at the bus level between cache accesses and I/O peripheral transactions, concluding that these kinds of inferences cause unpredictable behaviors. They present a theoretical framework able to model the interaction between the CPU and the peripherals accessing the front side bus. In [21] they address the cache partitioning problem (to avoid interference between different cores) as an optimization problem. The solution found by the optimization algorithm identifies the optimal size of each cache partition such that the system worst-case utilization is minimized and real-time schedulability is increased.

3.8 Summary

In this chapter we have proposed the fundamental for our multi-core architecture in which the maximum time that a request from a HRT accessing an on-chip shared resource can be delayed by any other task is bounded. That is, our multi-core processor enforces that a request of a HRT cannot be delayed longer than a given *Upper Bound Delay* (UBD).

We extend our multi-core architecture, which allows determining an UBD, with a novel hardware feature called *WCET Computation Mode* that allows estimating safe

WCET of HRTs running into our multi-core architecture. HRTs are run in isolation with the *WCET Computation Mode* enabled. In this execution mode, the processor artificially delays each HRT request by the worst-case delay that every HRT request can suffer due to the interaction with other tasks when run inside a workload. As a result, the computed WCET estimation is a safe upper bound of the execution of the HRT when it runs in *Standard Execution Mode* together with other tasks in the multi-core processor.

We have focused on the on-chip shared bus and shared cache, while in the next chapter we are going to address the off-chip memory, that is the shared resource with the highest impact on the WCET estimations. Moreover, our proposal can use current WCET analysis tools without requiring any modification, so whatever analysis tool is used in single-core systems can be applied to our multi-core architecture.

CHAPTER 4

Predictable Off-Chip Shared Resource: the DRAM Memory System

This chapter focuses on the off-chip memory system, which is the hardware shared resource with the highest impact on the WCET and hence one of the main challenges for the use of multi-cores in integrated architectures.

4.1 Introduction

An improper design of the memory system may affect system's predictability [88] as well as performance. This effect is specially high in multi-core processors, where inter-task interferences caused by the off-chip shared memory system have the most significant impact on the execution time [22, 66]. Experiments presented in [73] measured a WCET increment of 2.96 times due to memory interferences on a real multi-core processor.

In order to evaluate the impact that inter-thread interferences may have on the WCET estimation of HRT, we have performed the following experiment: we consider

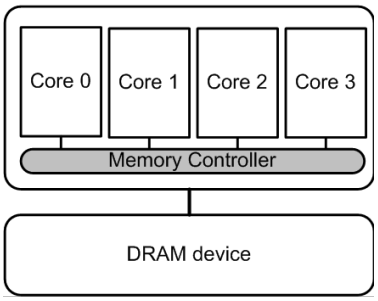


Figure 4.1: Multi-core architecture used as example

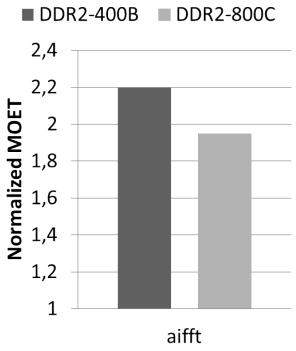


Figure 4.2: MOET of *aifft* running with instances of a memory intensive synthetic benchmark

a processor with 4 cores that are connected through a shared bus to a memory controller that interfaces the cores with a DDR2 SDRAM memory device. We have run the memory-intensive EEMBC Automotive Benchmark [77] *aifft*, that is an algorithm commonly present in any automotive system, co-scheduled with three instances of a memory-intensive synthetic benchmark that constantly accesses the main memory. Figure 4.2 shows the Maximum Observed Execution Time (MOET) increment of the *aifft* caused by memory interferences with respect to its WCET estimation computed assuming no conflicts accessing the memory. In particular we have used two different JEDEC-compliant 256 Mb x16 DDR2 SDRAM devices and we observe an increment up to 1.95x and 2.20x when using DDR2-800C and DDR2-400B respectively. Hence, if the inter-thread interferences between threads are not taken into account when computing the WCET estimation, the execution time of a HRT can go largely beyond its estimated WCET.

Thus, in order to enable a safe use of multi-core processors in integrated architectures, it is mandatory to consider the memory system into the WCET Analysis. If this is not the case, time composability cannot be guaranteed and hence integrated architectures cannot be used. In order to enable the use of multi-core processors in integrated architectures:

1. We present an analytical model, based on a memory controller configuration that implements a close-page row-buffer policy and an interleaved-bank address mapping scheme, to analyze in detail the impact of the SDRAM memory system on the WCET estimation. To do so, our analytical model computes the worst-case delay (UBD), that a memory request can suffer due to memory interferences generated by other co-running tasks. The selection of the memory controller configuration is based on the analysis of six different configurations, varying the row-buffer policy and the address mapping schemes.

2. We propose a *Real-Time Capable Memory Controller* (RTCMC) for multi-core processors. RTCMC is compliant with our analytical model, so it enables the use of multi-cores in integrated architectures. The RTCMC introduces two new features to reduce the overall WCET: (1) A mechanism to consider refresh operations in the WCET estimation, which are one of the main contributors of the low predictability of memory systems, and (2) a mechanism to minimize the impact of memory interferences caused by non hard real-time tasks (NHRTs) over hard real-time tasks (HRTs) in a mixed criticality workload.

Our analytical model is based on generic timing constraints as defined in the JEDEC industrial standard [49], so it can be used with any memory controller configuration, to compare different JEDEC-compliant DDRx SDRAM memory devices. In particular, in this thesis we analyze the UBD of three different JEDEC-compliant 256Mb x16 DDR2-SDRAM memory devices [49]: DDR2-400B, DDR2-800C and DDR2-800E.

The UBD can be used in the WCET analysis in order to take into account memory interferences and obtain a safe and composable WCET estimation for any task. That is, the resulting WCET estimation of a task is independent of the memory behavior of the other co-running tasks because the worst-case memory interference scenario is considered. The UBD information can be used in both WCET analysis tools: measurement- and static-based approaches, requiring no changes to current WCET analysis tools, so the same tools and techniques that are used and valid for single-core

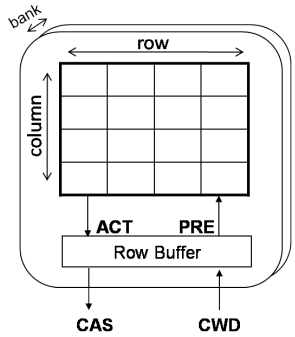


Figure 4.3: DDRx SDRAM memory system

processors can be used in the analysis of multi-core processors. Thus, no effort by the WCET analysis tools developers is required, with a reduction in the overall costs.

4.2 DDRx-SDRAM Fundamentals

In this chapter we focus on DDRx SDRAM off-chip memories[1] [46] (DDR, DRR2 and DDR3) that are compliant with the JEDEC industrial standard [49]. This type of memories is commonly used in high performance computer systems, especially multi-cores, but designers start to introduce them into hard real-time embedded systems as performance requirements increase [92], e.g., both Leon3 processor by Gaisler used in space applications, and MPC8641D processor by Freescale employed in avionics industry, implement a DDR SDRAM memory controller.

A DDRx SDRAM memory system is composed by a *memory controller* and one or more *memory devices*. The memory controller is the component that controls the off-chip memory system acting as the interface between the processor and memory devices, which store the data.

A memory device consists of multiple *banks* that can be accessed independently. Each bank comprises *rows* and *columns* of DRAM cells and a *row-buffer* which caches the most recently accessed row in the bank (see Figure 4.3). A row is loaded into

[1] By default, the term DRAM memory refers to JEDEC-compliant DDRx SDRAM memory.

the row-buffer using a row activate command (*ACT*). Such command *opens* the row, by moving the data from the DRAM array cells to the row-buffer sense amplifiers. Once a row is open, any read/write operation (*CAS/CWD* command) can be issued to read/write a *burst* (of size *BL*) of data from/to the row-buffer, with a latency of $BL/2$ memory cycles, i.e. the memory is double data rate. Finally, a precharge command (*PRE*) *closes* the row-buffer, storing the data back into the memory cells. The memory controller can use two different policies to manage the *row-buffer*: *close page* that precharges the row after every access and *open page* that leaves the row in the *row-buffer* open for possible future accesses to the same row.

The DRAM-access protocol defines strict timing constraints [46], specified by the JEDEC industrial standard [49] and provided by memory vendors, that dictate the minimum interval of time between different combinations of consecutive DRAM commands. Such timing constraints strictly depend on the memory characteristics. Table 4.1 summarizes the most relevant timing constraints of three JEDEC-compliant 256Mb x16 DDR2-SDRAM memory devices considered within this thesis: 400B, 800C and 800E. The values are expressed in memory cycles, except the clock rate of the memory that is represented in nanoseconds.

The time a request to a DRAM system memory takes to be execute depends on the *memory device* and the *memory controller*. In the memory device, the variability is originated by the *DRAM-access protocol*, which defines the different *timing constraints* and so the minimum interval of time between different combinations of DRAM commands of a memory request. In the memory controller, the variability depends from the *Row-Buffer Management policy*, the *Address-Mapping Scheme* and the *Arbitration policy*.

In SDRAM memories, for data integrity, the data must be periodically read out and restored to the full voltage level with *refresh* operations (*REF* commands). Every t_{REFI} cycles a refresh command (REF) is automatically sent to all banks, refreshing one row per bank. This operation takes t_{RFC} cycles to be completed.

Table 4.1: Timing constraints of 256Mb x16 DDR2 devices: 400B, 800C and 800E [49] (values in memory cycles)

Time	Description	400B	800C	800E
t_{CK}	Clock rate in ns	5ns	2.5ns	2.5ns
t_{CAS}	CAS to bus transfer delay	3	4	6
t_{RCD}	ACT to CAS/CWD delay	3	4	6
t_{RP}	PRE to ACT delay	3	4	6
t_{RC}	ACT to ACT delay	11	22	24
t_{RAS}	ACT to PRE delay	8	18	18
t_{BURST}	Data bus transfer	4	4	4
t_{CWD}	CWD to bus transfer delay	2	3	5
t_{CCD}	CAS to CAS (same bank) delay	2	2	2
t_{RTP}	Last 4-bit prefetch of CAS to PRE delay	2	3	3
t_{WR}	End CWD to PRE delay	3	6	6
t_{WTR}	End CWD to CAS delay	2	3	3
t_{RRD}	ACT to ACT (different bank) delay	2	3	3
t_{RFC}	Time required to refresh a row	15	30	30
t_{REFI}	REF to REF delay	1560	3120	3120

4.3 An Analytical Model to Compute the UBD of a Memory Request

In a multi-core architecture several threads run in parallel and each can generate simultaneously different requests to memory. Under this scenario the timing behavior of a memory request is characterized by the *Request Inter-Task Delay* and the *Request Execution Time*. The former represents the delay the memory request can suffer - due to interferences with other requests generated by co-running tasks running on different cores - before getting the access granted to the memory device. The latter identifies the time a memory request takes to be completed, once it cannot suffer interferences with the other threads' requests. The goal of our analytical model is to define an

Upper Bound Delay (UBD) to the *Request Inter-Task Delay*, so that by taking it into account during the WCET analysis of a HRT, the WCET estimations are independent from the rest of the co-running threads, and so they enable time composability.

First, we are going to define the Request Inter-Task Delay, and then we are going to describe how to bound it. The Request Inter-Task Delay depends on:

- Number of *scheduling-slots*[2] a thread has to wait before being granted access to the memory. The channel to access the memory is shared between all of them, and so they interfere with each other. Among the different memory requests that access simultaneously the memory, the memory controller is in charge of scheduling the next request. The arbitration policy implemented into the memory controller and the structure of the internal queues used to buffer the memory requests inside the memory controller determines how many *scheduling-slots* (rounds) a thread may need to wait to get access to such shared resource.

- Duration of each scheduling-slot: the *Issue Delay*. The *Issue Delay* is the time interval between the issue of two consecutive requests, i.e. from the instant a request is issued until the next one can be issued. Moreover in this work, we define the Upper Bound for the Issue Delay, that we called Longest Issue Delay (t_{LID}).

  The Issue Delay depends on the DRAM device used, on the specific timing constraints (see as a reference Table 4.1), on the *Row-Buffer Management* policy and the *Address Mapping* scheme implemented in the memory controller. Depending on the row-buffer management policy used (i.e. open page or close page) and on the previous request, the sequence of DRAM commands that the memory controller issues to the memory device in order to serve a memory request can vary. That is, in DDRx DRAM memory systems the time a request takes to be executed may depend on the previous requests. For example, a read operation has a shorter duration if the previous request was also a read than

[2]When using an arbitration policy, we define as *scheduling-slot* the round, or the time-slot of the arbitration phase in which a new request is selected by the arbiter.

if the previous request was a write. This significantly complicates the WCET analysis of HRTs because the duration of a memory request may depend on the other co-running tasks, and in a multi-core, it is likely the scenario where the previous request of a certain request R is originated by a different thread. As a consequence, its execution time depends on the other co-running threads, which breaks the principle of time composability. The address-mapping scheme defines how a physical address is mapped to banks, rows and columns in the DRAM device. This can potentially originate a bank conflict if two subsequent requests access simultaneously the same bank.

In order to ensure time composability of tasks executed on multi-cores, we propose an analytical model based on generic timing constraints defined in the JEDEC industrial standard [49], that can be used to compute the Upper Bound Delay (UBD) of the Request Inter-Task Delay. We also show that, by considering the UBD during the WCET analysis of a task, the resulting WCET estimation is independent from the workload, and hence the task becomes *time composable*.

4.3.1 Analysis of Different Memory Controller Configurations

As pointed in the previous section, the Request Inter-Task Delay depends on different configuration parameters of both the memory controller and the memory device. Making a complete analysis of all possible combinations of parameters is infeasible given the high number of combinations. We focus on a subset of those configurations, though our analysis can be easily extended to all the cases.

In particular this section compares six different memory controller configurations: two row-buffer policies (*open-page* and *close-page* row-buffer) and three address mapping schemes (*shared-bank*, *private-bank* and *interleaved-bank* address mapping schemes). The shared-bank scheme maps memory addresses as follows: a cache line is stored into a single memory bank and all tasks share all the memory banks. In the private-bank scheme (used in [54]), the address mapping scheme maps each cache line into the memory bank owned by the task and the other tasks cannot access it. In the *interleaved-bank* [11] address mapping scheme, a cache line is split among all banks,

Table 4.2: Longest Issue Delay for different memory controller configurations and three different JEDEC-compliant 256Mbx16 DDR2 SDRAM devices: 400B, 800C and 800E

	400B	800C	800E
close-page/interleaved-bank	21	23	27
close-page/private-bank	21	23	28
close-page/shared-bank	53	89	89
open-page/interleaved-bank	21	26	27
open-page/private-bank	9	11	13
open-page/shared-bank	28	35	43

so each memory request accesses all the banks in sequence (we consider a 4-banks DRAM device) and this way DRAM commands can be effectively pipelined. In the last scheme described, the access granularity is equal to $BL \cdot N_{\text{banks}} \cdot W_{\text{BUS}}$ bytes, where N_{banks} is the number of banks in the DRAM device, and W_{BUS} the bus width in bytes [11].

For all the memory controller designs we assume that for every memory request a cache line of 64 bytes is transfered (i.e. the typical size for a second level cache line in high performance embedded processors [92]). Each of the six memory controller design configurations is analyzed for three DDR2 DRAM memory devices defined in the JEDEC industrial standard (400B, 800C and 800E); for a total of 18 cases.

Table 4.2 shows the results (expressed in memory cycles) of comparing the 6 memory controller configurations for 3 different memory devices. We observe that shared-bank and private-bank schemes benefit from using open-page policy since, once the row-buffer has been activated, a set of bursts can be transferred to read/write the complete cache line. Instead, when using close-page policy, the row-buffer is precharged after every read/write operation, enlarging the duration of the memory request. This is not the case for the interleaved-bank scheme that provides a lower Longest Issue Delay when using a close-page policy. The reason is the use of the auto-precharge feature, while the open-page policy requires sending explicitly a PRE

command through the command bus, and so it results in a bus conflict for the 800C device.

When comparing the different address mapping schemes: private-bank reduces the Longest Issue Delay with respect to shared-bank because bank interferences that are originated due to tasks that access simultaneously the same memory bank are removed. In the private-bank scheme, in fact, the different HRTs share only the data and command bus of the DRAM device while in shared-bank all the components of a DRAM device are shared among the threads.

The Issue Delay has a direct impact on the UBD and so on the WCET of a HRT: the higher the Issue Delay that a memory request can suffer due to memory interferences is, the higher the WCET of the HRT. From a WCET point of view, the private-bank address scheme with open page policy provides the lowest Issue Delay. However, this approach does not provide enough system flexibility as it partitions statically the memory. It would be not possible for different tasks to use the memory according their requirements, and moreover this approach would require at least one memory bank per core, which clearly presents scalability issues at architectural implementation level. For example, in a n-core multi-core processor each thread would receive a partition of $1/n - th$ of the total memory size, disregarding the memory requirements of each thread and hence leading to a poor utilization of the memory because the memory will be partitioned independently of the memory-footprint of the different HRTs.

For this reason, we select the configuration with the second lowest Longest Issue Delay but that guarantees flexibility: close-page row-buffer management policy and interleaved-bank address mapping scheme, for which we provide a complete analysis of the UBD. However it is worth noticing that, the different steps required to compute the UBD model of other memory controllers are very similar to the ones provided in this chapter and can be easily derived.

The first contribution of this chapter is shown in Table 4.2: a comparison of the Longest Issue Delay for different memory controller and memory device configurations of DDRx SDRAM off-chip memories. This allows one to design hardware from

the WCET point of view, and not from the average performance point of view as usually the case.

4.3.2 Defining an Upper Bound to the Issue Delay

We define the Longest Issue Delay (t_{LID}), Upper Bound for the Issue Delay, using the generic timing constraints defined in the JEDEC standard (see Table 4.1) for a memory system that implements a close-page row-buffer management policy and an interleaved-bank address mapping scheme.

t_{LID} is determined by: (1) the minimum time interval between two consecutive row activations of the same bank, that we define *time issue bank* (t_{IB}), and (2) the data bus serialization that imposes the duration of the data transfer associated with a request. t_{IB} is at least equal to t_{RC} cycles, that is the timing constraint that determines the minimum time interval between two row activations issued to the same bank. This guarantees that, before activating a given bank, the previous request, issued to such bank, has been completed. However, depending on the type of the previous unfinished request [46], i.e. either a read or a write, t_{IB} is different, and it is defined as following:

- When the previous request is a read: $t_{\text{IBR}} = max\{(t_{\text{RCD}} + max(t_{\text{BURST}}, t_{\text{RTP}}) + t_{\text{RP}}), t_{\text{RC}}\}$
- When the previous request is a write: $t_{\text{IBW}} = max\{(t_{\text{RCD}} + t_{\text{CWD}} + t_{\text{BURST}} + t_{\text{WR}} + t_{\text{RP}}), t_{\text{RC}}\}$,

Both t_{IBR} and t_{IBW} are defined as the maximum between t_{RC} and the sequence of timing constraints associated to the commands issued by the memory controller to perform respectively a read and a write operation.

The data bus serialization does not allow different banks to be simultaneously activated: at least a number of cycles equals to t_{ACTB}, that we define as $max\{t_{RRD}, t_{BURST}\}$, have to be past. This way t_{RRD} is satisfied and the data can be transfered from/to a given bank because the data bus is available, after the transmission from the previous bank is finished (t_{BURST}). In addition to that, it is also required to take into account

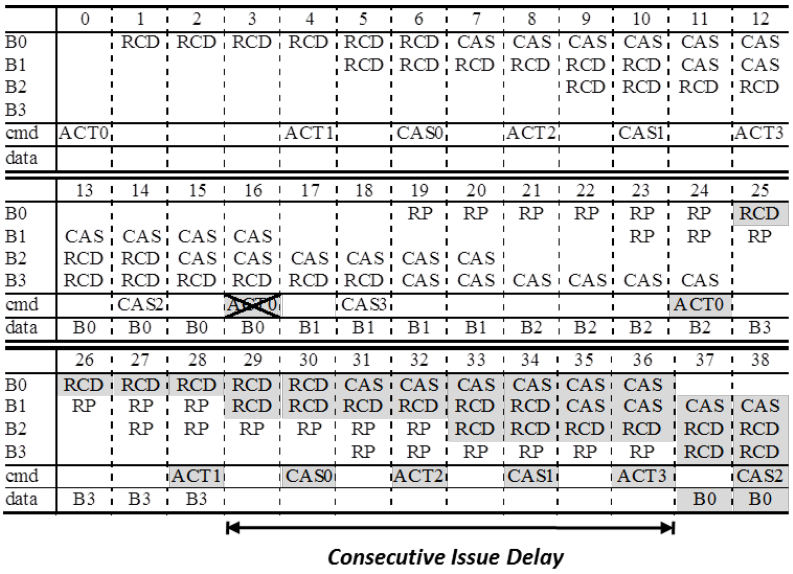


Figure 4.4: Timing of two consecutive read operations (one in white and one in gray) using a 4-bank JEDEC 256Mb x16 DDR2-800E SDRAM device

if two consecutive operations are or not of the same type: in case of *write after read*, requests are delayed due to both, the minimum time interval between the issue of a CWD and a CAS command and the t_{WTR} timing constraint. t_{WTR} accounts for the time that DRAM requires to allow I/O gating to overdrive the sense amplifiers before the read command can start, switching the direction of the bus. Thus, a *write after read* involves an additional delay of t_{WTR} + t_{CAS}. In case of *read after write*, requests are delayed by one extra cycle, because the duration of t_{CWD} is always defined as t_{CAS}-1 cycles [49] generating a shift on the data bus of one extra cycle.

To sum up, t_{LID} requires one to consider both the previous issued request and the current one, resulting in 4 different expressions:

- The *read-to-read issue latency* $t_{\text{LIDRR}} = max\{t_{\text{ACTB}} \cdot N_{\text{banks}}, t_{\text{IBR}}\}$ cycles.
- The *read-to-write issue latency* $t_{\text{LIDRW}} = max\{t_{\text{ACTB}} \cdot N_{\text{banks}} + 1, t_{\text{IBR}}\}$ cycles.
- The *write-to-write issue latency* $t_{\text{LIDWW}} = max\{t_{\text{ACTB}} \cdot N_{\text{banks}}, t_{\text{IBW}}\}$ cycles.

- The *write-to-read issue latency* $t_{\text{LIDWR}} = max\{(t_{\text{ACTB}} \cdot N_{\text{banks}}) + t_{\text{WTR}} + t_{\text{CAS}}, t_{\text{IBW}}\}$ cycles.

The worst possible scenario can be considered by defining:

$$t_{\text{LID}} = max\{t_{\text{LIDRR}}, t_{\text{LIDRW}}, t_{\text{LIDWW}}, t_{\text{LIDWR}}\} \tag{4.1}$$

By taking into account t_{LID} into the WCET analysis, the WCET estimations are time-independent from the requests sent to memory by the co-runner tasks, because the worst-case scenario is always considered. This is the key point to provide safe and composable WCET estimations in a multi-core processor. Even though such technique may introduce some pessimism (our experiments, presented in Section 4.5, show less than 29% with actual real-time applications provided by Honeywell into the WCET estimations.

An example for a 4-bank DDR2-800E device [49] is shown in Figure 4.4: in particular it shows the command bus (*cmd*), the data bus (*data*) and the bank status (B0 – B3) for two consecutive read requests (the first one in white and second one in grey).

In the example, each bank is activated every t_{BURST} cycles (at cycles 0, 4, 8 and 12) as in this case $t_{ACTB} = t_{BURST}$, so the data can be transferred in consecutive cycles (from cycle 13 to cycle 28). However, if a new request is ready to be served (in grey), it cannot be issued t_{BURST} cycles after activating B3 (crossed cell in cycle 16 of Figure 4.4) because the t_{IBR} of B0, which in this case is equal to t_{RC} cycles, would be violated. Instead, the new request must wait the *Consecutive Issue Delay* (t_{CID} cycles), being not issued until cycle 24. Observe that the t_{CID} also affects the data bus efficiency, reducing it down to $(t_{ACTB} \cdot N_{banks})/t_{LIDxx} \cdot 100\%$. In general, t_{CID} can be expressed as $|t_{LIDxx} - (t_{ACTB} \cdot N_{banks})|$. Note that, if we compute t_{CID} considering DDR2-400B SDRAM device instead of DDR2-800E device, it equals 0. Hence, by using the DDR2-400B device, a new request can be issued t_{BURST} cycles after activating B3 of the previous request achieving a data bus efficiency of 100%.

Our model considers single-DIMM, single-rank and single 4-bank device DDRx SDRAMs, as this is currently implemented in high performance embedded systems

[46]. Instead, high-performance processors for servers, laptops, market, implement more performance-oriented techniques (e.g. request bundling), and use more complex DDR2 or DDR3 DRAM memory systems in which additional timing constraints should be considered (e.g. t_{RTRS}, t_{FAW}). However, before applying any of these techniques and technologies to real-time systems, it is required to analyze the WCET impact instead of performance improvements.

4.3.3 Refresh Operations

As already mentioned, refresh operations are one of the main contributors of the low predictability of memory systems [20]. During a refresh operation no other command can be issued, hence enlarging the latency of a memory request. Since this may delay the duration of any memory request, as a consequence, it can have effects on the WCET estimations.

A possible solution to take into account this delay in the computation of the t_{LID} would be to consider that every single request is affected by a refresh operation: $t_{LID+REF} = t_{LID} + t_{RFC} - 1$.

However, it is clear that applying $t_{LID+REF}$ to every memory request would lead to an overestimated WCET, because a task suffers in the worst-case only a refresh every t_{REFI} cycles. In Section 4.4 we show how we enhanced our real-time capable memory controller with a special hardware feature to reduce the refresh impact. In particular we synchronize the start of the execution of a HRT with the memory refresh. By doing so, the impact of refreshes during WCET analysis is smaller or equal than during normal execution.

4.3.4 Bounding the Request Inter-Task Delay: the UBD

In order to compute safe and composable WCET estimations in multi-core processors, in addition to determine a safe upper bound for the Issue Delay, it is required to consider the arbitration policy and the internal request queue structure implemented in the memory controller. This can be represented by defining the function f_{LARB} that determines the maximum number of scheduling-slots (or Longest Arbitration)

that a request may wait before getting the access granted, where the duration of each scheduling-slot is, in the worst case t_{LID}.

It is then possible, to define a safe upper bound for the overall Request Inter-Task Delay that Longest Request Inter-Task Delay (t_{LRITD}) or more simply the Upper Bound Delay (UBD) as a function of t_{LID}, $UBD = f_{LARB}(t_{LID})$.

In this work we consider a *round robin* arbitration policy [70], although other arbitration policies could be considered (like the one used in [11], TDMA, etc.). This analysis can be easily extended to other arbitration policies.

In the case of round robin, the number of inter-tasks interferences (i.e. arbitration slots) are upper bounded by the maximum number of tasks that can access simultaneously the memory (equal to the number of cores). Moreover, in order to isolate intra-task interferences (interferences originated from requests of the same task) from inter-task interferences (coming from requests of different tasks), our model considers one request queue per task. By doing this, the memory request scheduler considers only the top of each queue and not all the requests that are pending to be served from each thread. The worst-case scenario occurs when all HRTs that are executed simultaneously in the processor try to access the memory at the same time, and the HRT has to wait up to the total number of different co-running HRTs No_{HRT} (i.e. at most the total number of cores). Thus, the $f_{LARB}(x)$ function is defined as $f_{LARB\_HRT}(x) = (No_{HRT} - 1) \cdot x$ where x is the duration of each scheduling-slot, being in this case t_{LID}.

Regarding NHRTs, despite of their lower priority, it may happen that a request coming from a HRT arrives just one cycle after a request from a NHRT was issued to main memory, so the request from the HRT has to wait $f_{LARB\_NHRT}(x) = t_{LID} - 1$.

The total delay that a memory request can suffer is the sum of both effects: $UBD = f_{LARB\_HRT}(x) + f_{LARB\_NHRT}(x) = No_{HRTs} \cdot t_{LID} - 1$ where x is defined as t_{LID}.

Table 4.3: Request Execution Time for different memory controller configurations and three different JEDEC-compliant 256Mbx16 DDR2 SDRAM devices: 400B, 800C and 800E

	400B	800C	800E
close-page/interleaved-bank	16	18	16
close-page/private-bank	43	70	67
close-page/shared-bank	37	70	67
open-page/interleaved-bank	16	19	17
open-page/private-bank	16	16	19
open-page/shared-bank	16	16	19

4.3.5 Request Execution Time

The Request Execution Time (RET) is the amount of time a request takes to be completed, once it is ready and it cannot suffer interferences with the other threads, i.e. after UBD cycles in the worst-case. In particular it is the time necessary to transfer the data of a memory request on the bus from/to the memory banks (starting from the first until the last bit).

In Table 4.3 we show the Request Execution Time in memory cycles for the different configurations that we explore. Request Execution Time depends on two factors: t_{ACTB} that is the interval between the activation of different banks and t_{BURST} that is the time required to transfer a burst over the data bus.

4.3.6 Including the Effect of the Off-chip Memory Requests into the WCET Analysis

As explained in previous sections, in a multi-core processor, the timing behavior of memory requests is characterized by Request Inter-Task Delay and Request Execution Time. Therefore the maximum memory turn-around time for a memory request can be expressed as $t_{\text{mem}} = UBD + RET$.

In order to include the effect of the off-chip memory into the WCET estimation, the WCET analysis must be modified to take into account the t_{mem} computed using

our analytical model. Depending on the technique used to estimate the WCET we can apply different strategies:

- **Measurement-based techniques:** It is required to implement the *WCET computation mode* [70] in the memory controller. When computing the WCET estimation of a HRT, the processor is set in this mode and the HRT under analysis is run in isolation. Under this mode, the memory controller artificially delays every memory request by UBD cycles, so the HRT considers the worst-case delay that each memory request can suffer due to memory interferences of the other co-running tasks. Once the WCET analysis of all the HRTs is completed the processor is set to *Standard Execution Mode* in which no artificial delay is introduced.

- **Static-based techniques:** It is required to introduce our analytical model into the model used for the processor. In particular, instead of considering a fixed latency, whenever accessing main memory the processor model should consider that each memory request requires t_{mem} to be completed: UBD cycles to be issued (i.e. the worst-case needs to be considered to provide a safe WCET) and RET cycles to be executed.

4.4 The Real-Time Capable Memory Controller

This section presents our Real-Time Capable Memory Controller (RTCMC): a memory controller designed from a WCET point of view, and not from the performance point of view as it is generally the case in embedded and high performance systems. RTCMC is based on our analytical model, allowing to bound the effect that memory interferences have on the WCET. To that end, it implements a close-page management policy, an interleaved-bank address mapping scheme and a round robin arbitration policy with private request queue per core (an overall diagram of the memory controller structure is shown in Figure 4.5). RTCMC also implements special hardware features to deal with memory refresh operations and it minimizes the effect of NHRTs over HRTs, reducing the WCET estimation of HRTs; requests from HRTs

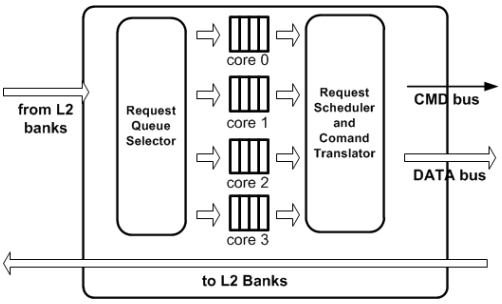


Figure 4.5: Structure of the our memory controller

are, in fact, allowed to preempt on-flight memory requests from NHRTs at memory bank boundaries.

4.4.1 Reducing the effect of NHRTs on HRTs

This section presents a new hardware mechanism that allows memory requests from HRTs to preempt on-flight NHRTs memory requests, reducing the memory interference caused by NHRTs on HRTs. Our technique exploits the independent functionality of the different banks. In particular, in the interleaved-bank address-mapping scheme, every memory request is composed by a sequence of independent accesses to all memory banks, so before issuing a NHRT request to a bank, our memory controller checks if there is any pending HRT request. If this is the case, the scheduler will prioritize the HRT request, stopping the issue of the remaining banks of the NHRT and assigning those issue slots to the HRT.

For example, let us assume that a request from a NHRT accesses bank B0 but before accessing B1, a request from a HRT arrives. In this scenario, RTCMC stops the NHRT request and gives the issue slot to access B1 to the HRT request. Once the HRT request is completed (sequence B1,B2, B3 and B0), the NHRT request is resumed starting from the bank it was preempted, i.e. B1. This example is shown in Figure 4.6. It is important to remark that our technique always maintains the same bank access order (for this work banks B0, B1, B2 and B3), exploiting the benefits of the multi-bank memory devices.

Figure 4.6: Data bus utilization of 2 different memory requests. A HRT request (in gray) preempts a NHRT request (in white). For each element it is indicated from which bank (Bn) the burst is transmitted

The use of this technique requires to re-define our analytical model:
concretely $f_{LARB\_NHRT}(x)$. It is in fact, not required to wait until the whole NHRT request is finished ($t_{LID} - 1$ cycles, in our case) but it is required to wait until the activation of the next bank. Thus, the worst-case scenario occurs when a HRT arrives one cycle after the NHRT request has activated the last bank (B3), as it has the longest bank-to-bank activation interval. In that case the HRT request must wait t_{ACTB} + t_{CID} cycles to activate first bank B0 (see Figure 4.4), resulting in: $f_{LARB\_NHRT}(x) = t_{ACTB} + t_{CID} - 1$. Therefore, the new UBD that a HRT can suffer is: $UBD = (No_{\text{HRTs}} - 1) \cdot t_{LID} + t_{\text{ACTB}} + t_{CID} - 1$

Note that the new $f_{LARB_{NHRT}}$ does not depend on t_{LID} anymore. Moreover in certain DRAM devices like DDR2-400B, the $f_{LARB\_NHRT}(x)$ becomes equal to $t_{ACTB} - 1$ cycles because t_{CID} is 0.

Our technique reduces the impact of NHRTs on HRTs from 17 cycles to 3 cycles in case of DDR2-400B, from 22 to 10 cycles in case of DDR2-800C and from 26 to 14 cycles in case of DDR2-800E, representing a reduction of 82%, 55% and 40% respectively.

4.4.2 Refresh Operations

As pointed out in Section 4.3, refresh operations can delay HRT memory requests, and this may increase the WCET of a task. To consider the delay introduced by refresh operations, a possible solution would be to account such delay in every memory request. However, such approach involves a huge overestimation of the WCET, because a task suffers exactly N_{REF} refresh interferences and not one for every memory request.

The number of refresh operations occurring during the WCET of a HRT is defined by the following iteration: $N_{REF}^{k+1} = \lceil (WCET_{\text{NOREF}} + N_{REF}^{k} \cdot t_{RFC}) \div t_{REFI} \rceil$ where $WCET_{\text{NOREF}}$ is the WCET estimation without considering the impact of refreshes and N_{REF} is the total number of refreshes.

Theoretically it is possible to add the delay suffered due to memory refreshes on top of the estimated WCET. Thus, the new WCET could be computed as: $WCET_{\text{TOTAL}} = WCET_{\text{NOREF}} + N_{\text{REF}} \cdot t_{\text{RFC}}$. However, depending on the approach used to perform the WCET analysis, i.e. static-based or measurement-based, it would be impossible to compute $\text{WCET}_{\text{NOREF}}$. Static-based approaches are based on abstract processor models, so $\text{WCET}_{\text{NOREF}}$ could be computed. While, measurement-based approaches are based on measurements on real processors that execute the tasks, and so refresh operations cannot be disabled. In this latter scenario it is impossible to compute $\text{WCET}_{\text{NOREF}}$. Moreover, the instant or the point of the program in which the refresh occurs may have a different effect on the WCET. Moreover, the particular time instant in which the refresh operation occurs cannot be statically determined because it depends on the exact instant in which the application starts.

To reduce such WCET overestimation, we propose an alternative solution: synchronizing the start of a HRT with the occurrence of a refresh operation during analysis and execution time, so in both cases the start of the task is delayed until the first refresh operation takes place. By doing so, refresh commands will produce the same interferences during the analysis and the execution as they will occur exactly at the same instant with respect to the start of the task. Hence, our technique only requires one to take into account the delay introduced due to the synchronization, resulting in a tight WCET. In the worst-case the task is launched one cycle after the memory has completed a refresh command, and so it must wait (t_{REFI} - 1) before starting to execute. The overall WCET is: $WCET_{\text{TOTAL}} = WCET + t_{\text{REFI}} - 1$ where $WCET$ is the measured WCET estimation.

The technique proposed for dealing with refresh operations assumes that the worst-case path is known and that an input test for such application path can be provided

during the WCET analysis. Later in this section, we explain the hardware support necessary to synchronize the start of a task with refresh operations.

4.4.3 Accounting UBD in the WCET Analysis

To consider the UBD provided by our analytical model in the WCET analysis, RTCMC introduces the *WCET computation mode* already described in Chapter 3.

The UBD computed by our analytical model only depends on the total number of HRTs that access the shared resources, which is bounded by the number of cores, and not on the sequence or history of memory request accesses. Thus, by using it in the WCET computation mode, RTCMC enables computing a safe and *time composable* WCET estimation that is independent from the workload. For instance, if we consider an UBD computed assuming 4 HRTs (WCET-mode 4) in a 4-core processor, the WCET estimation computed for the HRT under analysis is safe for *any* workload in which the HRT runs on this processor.

One of the main advantages of RTCMC is that existing WCET analysis tools already used in single-core systems can be used to compute WCET estimations from the UBD estimations. As a matter of example, we used RapiTime (the original version unchanged) for the WCET analysis of HRTs running in our multi-core architecture.

Once the WCET analysis for all the HRTs is completed, the processor is set to *Standard Execution Mode* in which no artificial delay is introduced. In [14, 83] it has been formally proved that even if instructions execute before their estimated time in the worst case, the computed WCET is a safe upper-bound of the real WCET.

4.4.4 Hardware Implementation

RTCMC requires simple hardware modifications, with respect to a common memory controller, to implement the new features that make it time-analyzable and the tasks composable.

First, in order to allow HRTs to preempt requests from NHRTs, the memory controller implements different request queues: one per core for HRTs and a common

one for all NHRTs, so NHRTs can save the data already transmitted before being pre-empted without the need of re-starting the request. Moreover, since the first accessed bank for a memory request does not require to be bank B0 but any other bank, a multiplexer is required to order the transmitted data properly.

In order to consider the refresh operation in the WCET estimation, the processor starts to fetch instructions from a new task when a refresh operation finishes. In order to implement this, the fetch unit of each core is enhanced with the following hardware mechanism: once the OS schedules a new program onto a core, the fetch unit does not start fetching instructions until a signal coming from the memory controller is high. Such signal is controlled by the memory controller, and it is enabled every time the refresh operation ends, while is low otherwise. This mechanism ensures the synchronization of the execution of an application with refresh operations.

The WCET computation mode only requires a small lookup table to store the precomputed UBD values of the different modes and a countdown counter to store the appropriate UBD that it is decremented every clock cycle. When the counter reaches zero, i.e. after UBD cycles, the memory request scheduler issues the memory request that has been artificially delayed as it would be in the worst-case.

RTCMC completes the design of our real-time capable multi-core architecture: in Chapter 3 we described our solution to take into account on-chip interferences and the shared cache, here we address off-chip interferences. The overall picture of the proposed architecture is shown in Figure 4.7. In this thesis, all hardware resources are accounted to provide a composable WCET estimation. Hence, enabling the use of multi-cores in Integrated Architectures.

4.5 Results

In this thesis we model our RTCMC interfaced with three different JEDEC-compliant 256Mb x16 DDR2 SDRAM devices: DDR2-400B, DDR2-800C and DDR2-800E, each composed by a single DIMM, single rank and a single 4-banks memory device (see Table 4.1). We assume a CPU frequency of 800MHz, being a CPU-SDRAM

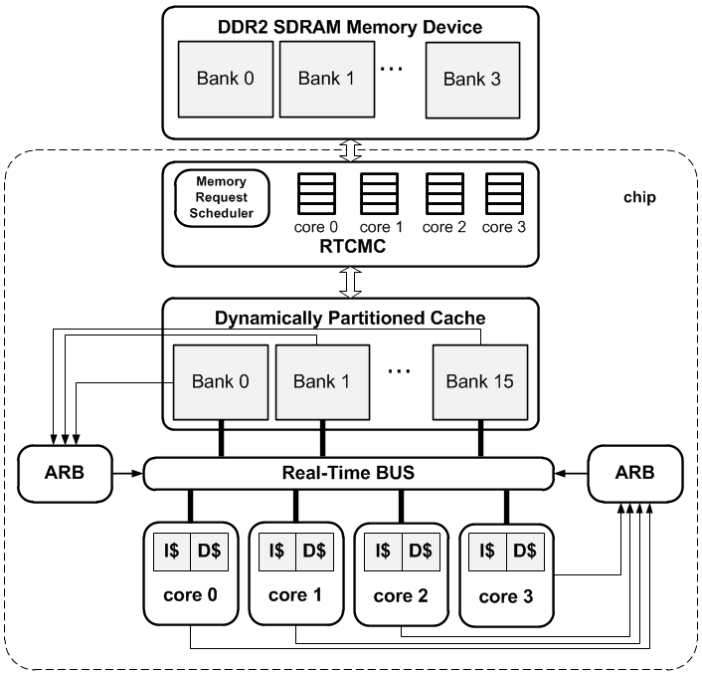


Figure 4.7: Our multi-core architecture

clock ratio of 2 in case of DDR2-800C and DDR2-800E and 4 in case of DDR2-400B.

We run several experiments with mixed application workloads (composed by HRTs and NHRTs) in order to show that hard real-time constraints of HRTs are satisfied while providing performance to NHRTs with the resources not used by HRTs. Our goal is to select memory-hungry NHRTs so that they interfere as much as possible with the HRTs. For the experiments shown in this chapter, we built workloads composed by 2 HRTs running on 2 cores and 2 NHRTs running on the other 2.

In order to study the impact of memory interferences on WCET estimations, we modified the synthetic benchmark called *opponent*, already described in Chapter 2, in such a way that each store instruction systematically misses in L2 cache (L1 cache implements a write-through policy) and so it always accesses the DRAM memory system. Thus, HRTs that are co-scheduled with such task will suffer a tremendous impact on their execution time. We show the impact that inter-task memory interfer-

Table 4.4: UBD in ns of DDR2-400B, 800C and 800E memory devices, considering No_{HRT}=4

	400B	800C	800E
t_{LID}	21	23	27
UBD	63	69	81
UBD (in ns)	**315**	**172.5**	**205.5**

ences have on the WCET estimation when using RTCMC, based on the UBD provided by our analytical model. We also compare the impact of the DRAM memory system with the impact of the inter-task on-chip interferences on the WCET estimation [70]. As main metrics we use the capability to provide tight WCET estimations to HRTs when running in a multi-core processor together with NHRTs and the level of performance that NHRTs achieve with the resources not used by the HRTs. Finally, in subsection 4.5.5 we compare RTCMC with a proposal of the state of the art, Predator [10, 11].

We also show an use case of our analytical model: in particular, we show how our model allows selecting the best DRAM device from a WCET point of view.

4.5.1 Use Case for the Analytical Model: Selecting the Best DRAM Device from a WCET Point of View

Our analytical model can be used to determine the best memory device from the WCET point of view. In this section we compare DDR2-400B, DDR2-800C and DDR2-800E devices. Let us consider an *execution workload* composed by four HRTs (No_{HRT}=4) running simultaneously. It is important to remark that No_{HRT} is the number of HRTs running at the same time inside the processor, which is upper bounded by the number of cores (4 in our baseline setup), and not the total number of HRTs that form the complete task set[3]. By considering the clock rate of each DRAM device (t_{CK}), our analytical model can determine which DRAM device introduces the highest

[3]The Operating System scheduler will be in charge of selecting which are the four HRTs that run simultaneously at every instant.

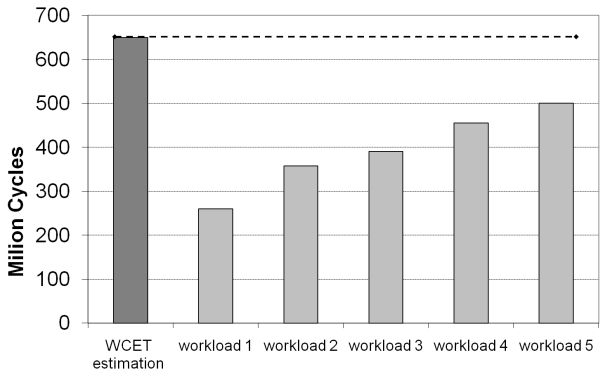


Figure 4.8: WCET estimation and execution time of CA on 5 different workloads

delay in the WCET estimation of a HRT. Table 4.4 shows the results of this comparison. The DDR2-400B DRAM device has the highest impact on the WCET, delaying every request by 315 ns, while the DDR2-800C has the lowest impact, delaying each request by 172.5 ns, representing a reduction of 44% with respect to DDR2-400B. The DDR2-800E delays every request by 205.5 ns.

4.5.2 Composable WCET Estimations

Time composability enables Incremental Qualification in Integrated Systems, which allows upgrading or changing a given function in a system without this affecting the WCET of other functions running on the same hardware. This section evaluates the execution time variation of the CA application in the 4-core architecture that has been previously described.

Figure 4.8 shows the execution time of CA when it runs in 4 different workloads each composed of three more tasks. The dark grey bar shows the WCET estimation for the CA based on the UBD (as described in Section 4.3).

We observe that depending on the interferences the other tasks of the workload have with CA, the observed execution time is different. The difference between the Observed Execution Time of the worst workload, $workload5$ and the WCET estimation is less than 29%. Although the $workload5$ is composed by several *opponent* benchmarks, which have a very aggressive memory behavior, we cannot ensure that

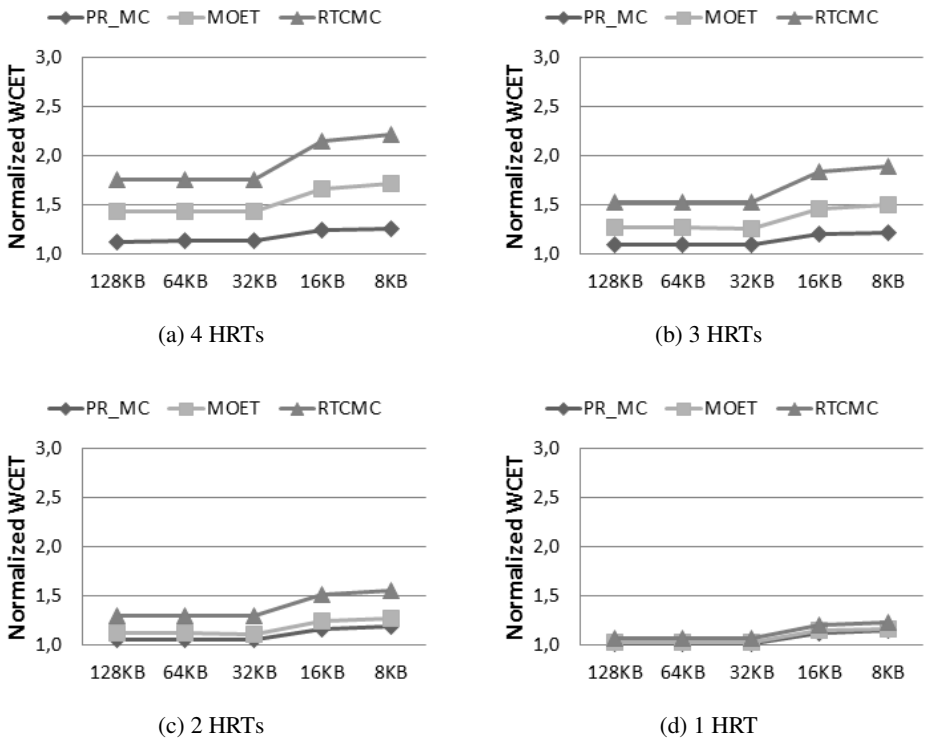
the *opponent* represents the worst-case scenario that CA could suffer. Building such actual worst-case opponent is hard or even impossible as it depends on the underlying processor and memory architecture and the particular HRT under study, requiring an enormous amount of time to be built and tested. Hence, a new workload composed of more memory hungry tasks, could lead to a higher execution time of CA. Any CA's execution time would be always lower than its WCET estimation, as, by design, we know which is the maximum delay CA can suffer and we take it into account in the WCET estimation.

The main conclusion we can extract is that the WCET provided is safe and time composable. At deployment time, all the other tasks with which CA runs, can be changed or updated without affecting the WCET of CA. This is a key feature of Integrated Architectures to heavily reduce design and deployment costs.

4.5.3 Tightness of the WCET estimations

In this section, we further evaluate the tightness of our WCET estimations under a wide set of configurations. Figures 4.9 and 4.10 show the WCET estimation provided by RapiTime for the CA application as we vary (1) the number of HRTs running simultaneously, (2) for each HRT count, we vary the private cache partition size assigned (from 128KB to 8KB) to show how the WCET changes as the pressure on the memory system increases. In order to ease the comparison of the WCET under different configurations, all WCET estimations are normalized to the WCET estimation in isolation, without conflicts accessing all the hardware shared resources (L2 cache, bus and DRAM memory).

For each configuration we compute the WCET estimation under two different scenarios: (1) assuming a Private DRAM Memory Controller for each HRT and so having interferences only in the on-chip resources [70], which has a high hardware cost in term of chip pins (labeled as *PR_MC*); and (2) on-chip and memory interferences are considered at the same time, our memory controller is shared among the different cores (labeled as *RTCMC*). For the latter scenario, in order to evaluate whether the WCET estimations are tight, we measure the Maximum Observed Execution Time



(a) 4 HRTs

(b) 3 HRTs

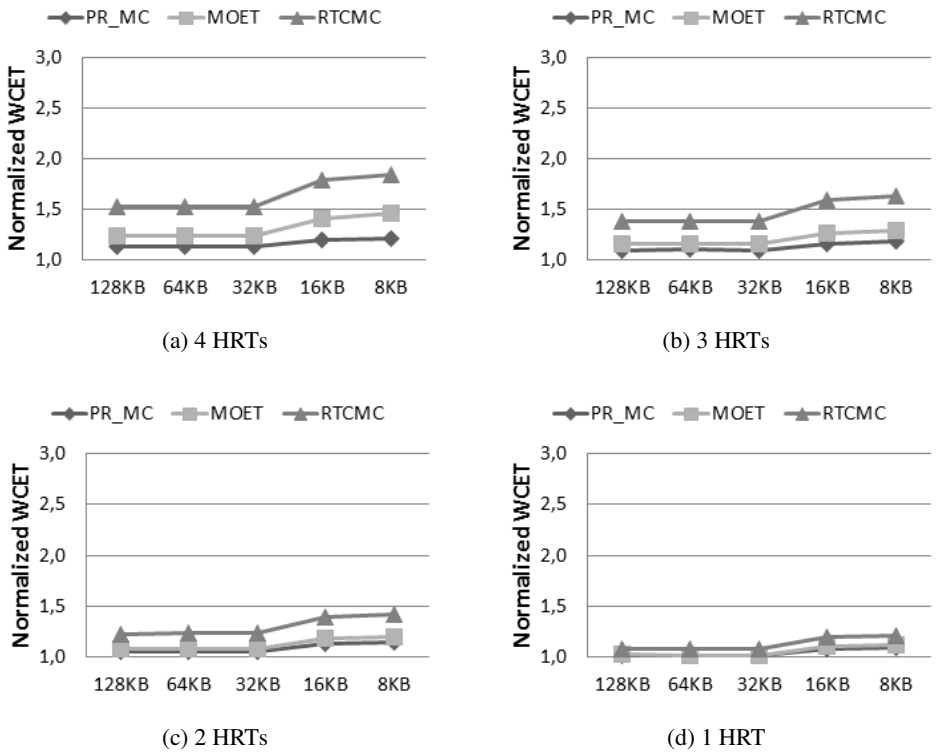
(c) 2 HRTs

(d) 1 HRT

Figure 4.9: Normalized WCET estimation for the CA application using JEDEC DDR2-400B

(labeled as *MOET*) of the HRT when running in a high memory-intensive workload composed by several *opponents*. For example, in Figure 4.9(a) for the 8KB configuration we observe that when the memory controller is not shared the WCET estimation is *PR_MC* is 1.25. When the memory controller is shared and we use RTCMC the WCET estimation is around 2.22, being the MOET 1.72.

By comparing *PR_MC* and *RTCMC*, we observe that memory interferences have a tremendous impact on the WCET estimation, significantly higher than on-chip interferences. In the scenario with the biggest amount of memory accesses, i.e. assigning 8KB of L2 and 4 HRTs in the workload (Figures 4.10(a) and 4.9(a)), the memory interferences increase the WCET estimations from 1.25 (PR_MC) to 2.22 (RTCMC) using DDR2-400B and from 1.19 (PR_MC) to 1.83 (RTCMC) using DDR2-800C,



(a) 4 HRTs

(b) 3 HRTs

(c) 2 HRTs

(d) 1 HRT

Figure 4.10: Normalized WCET estimation for the CA application using JEDEC DDR2-800C

which represents an increment of 0.97x and 0.64x respectively. Obviously, as memory pressure decreases, i.e. the size of the L2 cache partition assigned to the HRT increases and/or WCET computation mode decreases (Figures 2 (b) and 3(b)), the impact of memory interferences also decreases.

Even though memory interferences have such high impact, when comparing the MOET and the WCET of the CA application in the highest memory-intensive workload, i.e. assigning a cache partition of 8KB and running with 3 HRTs *opponents* (Figures 4.10(a) and 4.9(a)), we observe moderate increments: 29% (from 1.72x to 2.22x) in the case of DDR2-400B and 23% (from 1.41x to 1.83x) in the case of DDR2-800C. Note that, since the actual WCET of a HRT is bigger than any MOET and equal or smaller than any WCET estimation ($MOET \leq WCET_{actual} \leq WCET_{estimation}$)

[86], the WCET estimation obtained using RTCMC is less than 29% and 23% bigger than the actual WCET for DDR2-400B and DDR2-800C respectively. The average difference in percentage between MOET and RTCMC is 22% for DDR2-400B and 20% for DDR2-800C.

Table 4.5 shows the results for the experiments described above when using as HRTs the EEMBC Automotive benchmarks in the High (H) and Low (L) groups under the worst possible scenario, running in a workload with 4 tasks. Experiments are shown for two different DRAM memory devices: DDR2-400B and DDR2-800C.

Memory interferences introduce an increment in the WCET estimation of 167% (from 2.04x to 4.84x) for the high group, 64% (from 1.51x to 2.63x) for the medium group and 15% (from 1,03x to 1,18x) using DDR2-400B. When using DDR2-800C these increments are 97%, 49% and 10% for high, medium and low groups respectively.

The WCET estimations are quite tight with respect to the MOET. For HRT counts of 2 and 4 and all cache configurations (8KB to 128kB) the WCET estimations are 37%, and 3% higher in average than the MOET for the high, and low EEMBC groups respectively, when using DDR2-400B. For the same groups the increments are 29%, and 3% respectively when using DDR2-800C. Note that, as the memory pressure decreases, i.e. less HRTs access the memory, such difference also reduces. Medium EEMBC group resembles the results of CA application.

Finally, it is important to remark that the fact of delaying every memory request by UBD cycles when computing the WCET estimation, allows our memory controller to be time composable. However, the UBD can be used in non composable systems as well, for example by applying it only to certain memory requests if information about the actual inter-task interferences that a given task can suffer is available. By doing this, the WCET estimation can be reduced at the price of breaking the time composability property as the WCET estimation will be fully dependent of the composition.

Table 4.5: Normalized WCETs for EEMBC using DDR2-400B and DDR2-800C respectively (H=High, M=Medium, L=Low) in 4 HRTs workload

		DDR2-400B					DDRS2-800C				
		128KB	64KB	32KB	16KB	8KB	128KB	64KB	32KB	16KB	8KB
H	PR_MC	1.27	1.40	1.46	1.67	2.04	1.27	1.34	1.38	1.49	1.67
	MOET	1.11	1.44	1.67	2.30	3.67	1.07	1.24	1.37	1.75	2.53
	RTCMC	1.35	1.94	2.25	3.10	4.84	1.32	1.65	1.83	2.32	3.29
M	PR_MC	1.18	1.18	1.32	1.47	1.51	1.18	1.18	1.26	1.35	1.37
	MOET	1.09	1.07	1.43	1.78	1.94	1.05	1.05	1.24	1.46	1.53
	RTCMC	1.26	1.28	1.82	2.44	2.63	1.24	1.25	1.57	1.93	2.04
L	PR_MC	1.02	1.02	1.03	1.03	1.03	1.02	1.02	1.02	1.03	1.03
	MOET	1.08	1.09	1.10	1.10	1.12	1.05	1.05	1.06	1.06	1.07
	RTCMC	1.12	1.13	1.15	1.15	1.18	1.09	1.09	1.11	1.11	1.13

4.5.4 NHRT Performance Impact

The main goal of our architecture is to allow estimating tight, safe and composable WCETs for HRTs. The second goal is to get the maximum performance for the NHRTs that will benefit from the resources not used by the HRTs. In this section we analyze the IPC throughput we obtain for the NHRTs. We composed workloads with 4 threads, 2 HRTs and 2 NHRTs. We use HRTs with different memory demands: *high*, *medium* and *low* groups. We generate a memory-intensive pair *mpeg2dec* - *qsort*, which frequently misses in L2; and a CPU-intensive pair *susan* - *bzip2*. We study the throughput of the NHRTs when they run simultaneously with other HRTs, normalized to the case when the NHRTs run with no other HRTs at the same time. For memory-intensive NHRTs the normalized throughput ranges from 70% when they run with HRTs from low group, up to 50% whey they run with memory-intensive HRTs. CPU-intensive NHRTs do not show any performance degradation. Their normalized throughput ranges between 97% and 99% as their number of L2 misses is reduced. We observe that, our memory controller effectively allows NHRTs to exploit all the resources not used by the HRTs. As a rule of thumb, we can claim that the higher the

pressure of HRTs on the memory controller is, the lower the bandwidth available for NHRTs and hence the lower the performance are.

4.5.5 Comparing RTCMC and Predator

This section compares RTCMC with Predator [10, 11]. Predator is a memory controller for multi-processor system-on-chip designed for data-flow applications (e.g. some multimedia applications). It cannot be applied directly to hard-real time environments, which is the focus of RTCMC. If so, it would require the WCET analysis tool to be changed to provide worst case bandwidth estimations. Although the applications of both memory controllers, RTCMC and Predator are different, in this section we provide some quantitative comparison.

Predator guarantees an user-defined bandwidth to tasks based on user-defined fixed task priority. Task priorities are defined in a strict monotonic fashion, i.e. two different tasks cannot have the same priority. Based on the assigned bandwidth and priority, Predator provides a bound on the maximum latency of a memory request. The lower the priority of a task is the higher the upper bound delay is. In fact, according to the results presented in [11] assigning a priority of 3 will involve an upper bound delay 8 times higher than assigning a priority of 0. Defining the proper bandwidth is crucial because if a thread has higher bandwidth requirements that what expected by the user, the upper bound delay of a memory request may increase, involving potential deadline misses. Predator implements the same address mapping scheme used in this thesis: interleaved-bank. Predator focuses on streaming/multimedia real-time applications, in which bandwidth requirements can be easily determined. However, in other application domains, such as control-based applications, bandwidth requirements are unknown, hence we target different systems. RTCMC approach requires neither knowing the bandwidth requirements nor assigning a fixed priority to each thread allowing RTCMC being applied to control-based applications. Moreover, our analytical model allows quantifying the impact of the DRAM device, being suitable to any JEDEC-compliant DRAM device: our solution defines the UBD based on the generic DRAM timing constraints and the number of HRTs running simultaneously

Table 4.6: Normalized WCET estimation of CA application using UBDs provided by RTCMC and Predator [11]

RTCMC				PREDATOR			
	128KB	32KB	8KB		128KB	32KB	8KB
1 HRT	1,00	1,07	1,23	priority 0	1,36	1,67	1,98
2 HRTs	1,30	1,30	1,56	priority 1	1,70	2,31	2,99
3 HRTs	1,53	1,53	1,89	priority 2	2,37	3,59	5,02
4 HRTs	1,76	1,76	2,22	priority 3	4,77	7,50	11,19

in the processor. With RTCMC requests coming from different HRTs suffer the same UBD, however, although not presented in this chapter, RTCMC also allows defining priorities by applying, a two-level round-robin policy for HRTs, as shown in Section 3.6 for accessing the on-chip shared bus.

In order to provide a more accurate comparison between RTCMC and Predator, we used the UBDs provided by Predator to compute the WCET estimation of the CA application using the WCET computation mode technique. Concretely, we computed 4 different UBDs, one per each priority value, using the formulas presented in [11] and considering a workload composed by four CA applications with the same bandwidth requirements, and also varying the size of the cache partition (from 128KB to 8KB) for each priority. Only DDR2-400B SDRAM device is used because that is what authors use in [11]. Table 4.6 shows the WCET estimation of Predator with respect to the WCET running in isolation without inter-task interferences, i.e, the same baseline of the other experiments. We observe that in the highest memory-intensive scenario, i.e. assigning a cache partition of 8KB, Predator increases the WCET estimation of the highest priority HRT by 1.98x and by 11.19x for the lowest priority HRT. Instead, by using RTCMC with the same L2 cache partition size, memory interferences increase the WCET estimation of CA application with WCET computation mode 4 only by 2.22x. Therefore, although Predator reduces the WCET estimations of the HRTs with priority 0 it increases them for HRTs with priority 1, 2 and 3 with respect to RTCMC.

4.6 Related Work

Memory interferences have the highest impact on the WCET estimations of HRTs running on multi-cores, though they have not been widely studied. Many works deal with the execution of HRTs in multi-core/SMT processors [37] [70] [87] [14] [83] [41] focusing on how on-chip interferences affect the WCET analysis but without taking into account off-chip memory interferences. Predator [11] the most similar study done in this field, is a memory controller with which we compare RTCMC in Section 4.5.5. However, the goal of Predator's authors is to design a memory controller real-time capable for multimedia/streaming applications and for multi-processor system-on-chip that guarantees an user-defined bandwidth to tasks based on user-defined fixed task priority. Task priorities are defined in a strict monotonic fashion, i.e. two different tasks cannot have the same priority. The PRET machine [54] is a processor designed from a predictable point of view where the memory is statically partitioned, each thread gets different memory banks and the accesses are managed through a *memory wheel*. A TDMA-like policy is used to manage the memory wheel such that each thread can access to its bank during a static-assigned slot. However, the DRAM-access protocol is not taken into account.

In [37] it is proposed a switch-on-event single-core multi-threaded processor for real-time systems. The chapter presents a coarse grain timing analysis of the impact of bank and bus memory interferences on the WCET estimation without taking into account the DRAM-access protocol. Other approaches [66] [44] [80] [79] focus on the DRAM-access protocol from a high performance point of view, with Quality of Service (QoS) capabilities but without upper bounds on the latency of DRAM commands so they are not real-time capable. Nesbit et al. [66] propose a memory controller in which each task is provided with an average assigned bandwidth regardless of the load placed on the memory system from other threads.

Akesson et al. [12] presented an approach for composable resource sharing based on latency-rate servers. As case study they describe an SRAM memory controller. Their solution, is orthogonal to our, it would be possible to use our analytical model in

combination with their arbitration scheme to design an hard real-time capable memory controller.

The impact of DRAM refresh on task execution times with the focus on how predictability is adversely affected leading to unsafe hard real-time system design is shown in [20]. The authors propose an approach to overcome this problem through software-initiated DRAM refresh that they tested on a single core board.

Pitter and Schoeberl [74] [75] [76] introduce a real-time Java multiprocessor called JopCMP. It is a symmetric shared-memory multiprocessor and consists of up to 8 Java Optimized Processor (JOP) cores, an arbitration control device, and a shared memory. All components are interconnected via a system on chip bus. No details about the memory controller and the off-chip memory are provided.

4.7 Summary

In this chapter we have proposed an analytical model that (1) helps understanding the impact that different JEDEC-compliant DDRx SDRAM memory systems have on the WCET estimation of HRTs running in a multi-core. (2) Provides an Upper Bound Delay to the maximum effect of inter-task memory interferences. This enables the computation of WCET estimations for any HRT independently from the workload on which this task runs, making WCET estimations composable. (3) Allows designing hardware to reduce WCET (instead of reducing average case execution). We exemplified the use of our model by evaluating six different memory controller designs for three different JEDEC-compliant 256Mbx16 DDR2 SDRAM.

We propose a Real-Time Capable Memory Controller (RTCMC) for multi-core architectures. RTCMC is compliant with our analytical model, hence allows estimating the Upper Bound Delay (UBD) that each memory request can suffer, hence enabling the use of multi-cores in Integrated Architectures. RTCMC includes a new feature to deal with refresh operations, that are one of the main contributors to the variability in the low predictability of memory systems.

Overall, we have studied the problems related to inter-thread interferences accessing the main memory and proposed a possible approach to reduce the WCET of a HRT running in a multi-core processor.

CHAPTER 5

IA^3: Interference-Aware Allocation Algorithm

Once we designed a real-time capable multi-core processor in order to execute multi-programmed workload, it is necessary to assign the different tasks to the different cores in an efficient way. To that end, this chapter describes our allocation algorithm proposal that take advantage of multiple values of WCETs to consider different level of interferences, trying to minimize the resources used by the HRTs and hence, maximize the resources left to execute the NHRTs.

5.1 Introduction

A common problem addressed by system designers is the allocation of tasks to the different cores of a multi-core or multi-processor platform. Given a set of independent HRTs (defined as the *task set*), there is a huge number of different combinations on how executing those tasks on the given processor. Those combinations are determined by the order in which the different tasks are executed, the release time of the different jobs, the resources allocated to such tasks and the timing constraints they have (like the *deadline*).

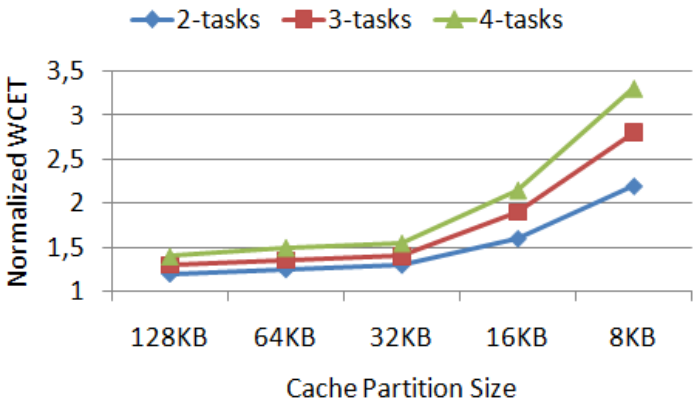


Figure 5.1: Normalized WCET estimations of *aifftr* under fifteen execution environments, defined by the number of simultaneous HRTs and the cache partition size assigned to each task, taking as a baseline the WCET estimation computed assuming no inter-task interferences, i.e. running in isolation

We identify as one of the main aspects that the designer should consider when defining the task allocation, the set of inter-task interferences that affect the execution of a task. Such inter-task interferences are determined by the *execution environment* in which the task runs. The execution environment is defined as the environment in which the task is executed; in a multi-core processor, it depends on both the workload and the hardware configuration. Regarding the *workload*, the scheduling/allocation algorithm determines the tasks that will execute simultaneously, defining the patterns of access to shared resources. Regarding the hardware configuration, in some architectures [28, 70], like the one proposed in this thesis, the hardware provides the software with some 'levers' to configure the arbitration policy of buses, the amount of cache assigned to each task, etc. Hence, the execution environment of a task depends on the workload with which the task runs and on the hardware configuration. Therefore, the execution of a HRT under different execution environments leads to different inter-task interference scenarios, and hence different WCETs for the task.

Despite the impact that the execution environment has on the WCET estimation, most approaches to hard real-time task scheduling consider only a single WCET value for each HRT: the highest among all execution environments in which the HRT can

run. Two notable exceptions to this are the works of Kato et al. [52] and Jain et.al. [47], which consider multi-case execution times. The research presented in this thesis differs in that our approach focuses on the WCET estimations of each HRT considering only the parameters that determine the execution environment in which a task runs, rather than on estimations of the execution time variations due to details of the various tasks when scheduling them in a simultaneous multi-threading environment. Hence our approach enables system composition based on information about individual components (i.e. applications or tasks) whereas previous work does not.

In this chapter we propose IA^3: an off-line *Interference-Aware Allocation Algorithm* that uses a set of WCET estimations corresponding to all the execution environments in which this task may run. IA^3, which is based on the *first fit decreasing heuristic* [31], introduces two novel concepts: the *WCET matrix* and the *WCET-sensitivity*.

The WCET-matrix is an n-dimensional vector space per HRT, where each dimension represents the parameters that determine different execution environments that may affect the WCET of the HRT. Thus, each execution environment has a WCET estimation associated with it. For example, in Figure 5.1, the WCET-matrix of the *aifft* task is a 2-dimensional vector space that considers the cache partition size and the number of HRTs in the workload in which *aifft* runs. The WCET-sensitivity complements the WCET-matrix by making the IA^3 algorithm aware of the impact of changing the execution environment on the WCET estimation. The WCET-sensitivity is derived from, and represents the variation across, the WCET-matrix.

The abstraction provided by the WCET-matrix and the WCET-sensitivity allows IA^3 to generate a very efficient partition that reduces the amount of resources (e.g. number of cores, cache partition size assigned to each core, etc.) required to schedule a given task set. Reducing the number of processors and hence the weight of the system is of significant benefit in many embedded systems, particularly those in avionics and space systems. Moreover, in mixed-criticality workloads, reducing the resources assigned to HRTs, allows the NHRTs to use more hardware resources and hence increase their quality of service.

We illustrate the concept of the IA^3 by using the multi-core architecture proposed in this thesis which has two main features: (1) the hardware allows the software to configure the size of cache partition assigned to each running task; (2) the inter-task interferences that a HRT may suffer depend only on the *number* of HRTs running simultaneously and not on the *particular* HRTs.

5.1.1 Notation

This chapter focuses on the *allocation problem* in homogeneous multi-core processors, considering a *partitioned* scheduling approach in which once a task has been assigned to one core it is not allowed to migrate. Moreover, we consider a *sporadic task model*, in which the arrival times of jobs of the same task are separated by a minimum inter-arrival time, referred to as the task's period. The arrival times of jobs of different tasks are assumed to be independent. At any point in time a job can execute on at most one core and, at most m HRTs can run simultaneously in an m-core processor.

In general, an allocation algorithm assigns a *task set* τ composed of n independent tasks $(\tau_0, ...\tau_n)$ to a set of m identical cores $(s_1, .., s_m)$. Each task, τ_i, is characterized by a WCET estimation C_i, a period P_i, and a hard deadline D_i, assuming *implicit deadlines*, i.e. $D_i = P_i$. The utilization of a task, u_i, is defined as $\frac{C_i}{P_i}$ and it ranges between $0 \leq u_i \leq 1$. The utilization of a task set, u_{sum}, is defined as $\sum_{\tau_i \in \tau} u_i$. The static partition generated by the allocator assigns a subset of tasks $\gamma_k \subseteq \tau$ to core s_k with a *cumulative utilization* $u^k_{sum} = \sum_{\tau_i \in \gamma_k} u_i \leq 1$. However, finding an optimal allocation is an NP-hard problem in the strong sense [39] and so non-optimal solutions derived from the use of bin-packing heuristics are typically used [35,55,69].

In a partitioned scheduling scheme, once tasks are allocated to cores, an on-line *uniprocessor* scheduling algorithm is used. In this thesis we consider non-preemptive EDF scheduling. A γ_k is *schedulable* using a non-preemptive EDF scheduling algorithm if the following conditions [50] are satisfied: (1) $u^k_{sum} \leq 1$ and (2) $\forall \tau_i \in \gamma_k \mid \forall L : P_1 < L \leq P_i, L \geq C_i + \sum_{j=1}^{i-1} \lfloor \frac{L-1}{P_j} \rfloor C_j$.

However, any other non-preemptive uniprocessor scheduling algorithm can be used. Note, we consider non-preemptive scheduling as this is used in many commercial systems, particularly those in aerospace applications. Non-preemptive scheduling significantly reduces the difficulty involved in obtaining valid estimates of the WCET, as the use of preemption can introduce new inter-tasks interferences due to the cold start when the task is resumed.

We consider the analyzable multi-core processor described in Chapters 3 and 4 as the target processor in which task sets are allocated. Under this architecture, the execution environment in which a given HRT can run is identified by two parameters: (1) the cache partition size assigned to each core and (2) the number of HRTs running simultaneously at any point in time.

Therefore, under this architecture, there are as many execution environments as possible cache partitions multiplied by the number of simultaneous HRTs, upper bounded by the total cache size and the number of cores respectively.

5.2 Our Proposal

This section describes in detail the algorithm we propose to generate a more efficient allocation, but focusing first on the novel concepts we introduce: the *WCET-matrix* and the *WCET-sensitivity*.

5.2.1 WCET-matrix and WCET-sensitivity

Current scheduling approaches typically consider only a single WCET value per task to perform the allocation, which usually corresponds to the highest WCET estimation in all execution environments in which the HRT can run [47]. Instead, in this thesis we propose an allocation algorithm that considers a set of WCET-values, each corresponding to a different execution environment in which the task can run.

Definition 1. *Given a HRT, the WCET-matrix is the collection of WCET estimations of this HRT when it runs on a processor under different execution environments.*

	16KB	32KB	64KB
1 HRT	1	1,2	1,4
2 HRTs	1,3	1,5	1,8
3 HRTs	1,4	1,8	2,3

Figure 5.2: Example of WCET-matrix

	from 16KB to 32KB	from 32KB to 64KB
1 HRT	0,2	0,2
2 HRTs	0,2	0,3
3 HRTs	0,4	0,5

Figure 5.3: Example of WCET-sensitivity

Thus, each WCET estimation inside the WCET-matrix is identified by the execution environment parameters that result in that WCET. For example the number of simultaneous HRTs that run in a multi-core processor, and the size of the cache partition assigned to each task.

Note, not all HRTs are affected in the same way due to inter-task interferences. For example, when comparing the WCET estimation of a memory intensive HRT that requires a lot of accesses to main memory in an execution environment with two, three or four HRTs running simultaneously, its WCET estimations will vary more than the WCET estimations of a CPU bounded HRT that does not access the main memory.

Definition 2. *Given a HRT, the WCET-sensitivity measures the WCET estimation variation among the different execution environments that comprise the WCET-matrix.*

Thus, a HRT with low WCET-sensitivity means that the variation of the WCET estimations under different execution environments is small, while a HRT with high WCET-sensitivity means that different execution environments in which the task is executed make the WCET change significantly.

Figures 5.2 and 5.3 show two simple examples to better understand the concepts of WCET-matrix and WCET-sensitivity. The two matrices consider an execution environment where it is possible to configure the number of simultaneously running HRTs and the size of the cache partition. Each element of the WCET-sensitivity matrix is obtained by subtracting two adjacent elements of the WCET-matrix.

Therefore, by considering the WCET-matrix and the WCET sensitivity, the allocation algorithm can be aware of how the execution environment impacts on the WCET

estimation of each task, thus enabling a more efficient allocation to be found. Current partitioning allocation approaches, e.g. first-fit decreasing, prioritize tasks by sorting them based on their utilization, so tasks with higher utilization are allocated first. Instead, WCET sensitivity enables partitioning based on the execution environment required, such that tasks with higher resource demand are allocated first.

5.2.2 Re-defining Terminology and Notations

The introduction of the WCET-matrix and the WCET-sensitivity makes it necessary to extend the terminology and notations provided in Section 5.1.1.

Our interference-aware allocation algorithm not only assigns a task set τ composed of n independent tasks $(\tau_1, ...\tau_n)$ to a set of m identical cores $(s_1, .., s_m)$, but also selects the execution environment in which each task will run based on the WCET-matrix and the WCET-sensitivity. For that, we define configuration φ as the set of tuples $(< e_1, a_1 >, ..., < e_n, a_n >)$, such that each task $t_i \in \tau$ has an associated tuple $< e_i, a_i >$ that specifies the execution environment e_i in which t_i runs, and the core a_i to which t_i is allocated. It is important to remark that, in a partitioned scheduling approach such as the one considered in this thesis, tasks are statically assigned to different cores, forcing each task to be executed exclusively on one core. Therefore, all tasks that have been assigned to a given core are forced to have the same execution environment.

The relationship between the WCET estimation and the execution environment can be defined as an n-dimensional vector space $M \subseteq \mathbb{N}^{z+1}$ of $n = z + 1$ dimensions, in which the first z dimensions identify all possible parameters that define an execution environment and the $(z + 1)$-th dimension identifies all possible WCET estimations of a HRT under the different execution environments. A vector $\hat{v} =< v_1, ..., v_{z+1} >$ associates each WCET-estimation (v_{z+1}) with its corresponding execution environment ($v_1, ..., v_z$). The WCET-matrix of each task $\tau_i \in \tau$ is a vector sub-space $WM \subseteq M$ that corresponds to the set of $\hat{v}$ in which this task can run. We require that all WCET-matrices are such that: Given a task τ_i and two vectors $\hat{v}_1$ and $\hat{v}_2$ both from the same task, if the execution environment of $\hat{v}_1$ requires less resources

than $\hat{v}_2$, the associated WCET estimation of $\hat{v}_1$ is equal to or higher than the associated WCET estimation of $\hat{v}_2$.

Similarly, the variation that the WCET estimation has among all different execution environments can be defined as a n-dimensional vector space $S \subseteq \mathbb{Z}^{z+1}$. A vector $\hat{w}$, defined as $\hat{v}_1 - \hat{v}_2$ where $\hat{v}_1$ and $\hat{v}_2 \in M$, measure the WCET estimation variation when changing the execution environment from the one defined in $\hat{v}_2$ to the one defined in $\hat{v}_1$. Hence, the WCET-sensitivity of each task $\tau_i \in \tau$ is a vector subspace $WS \subseteq S$ that corresponds to the set of $\hat{w} = \hat{v}_1 - \hat{v}_2$ in which $\hat{v}_1$ and $\hat{v}_2$ belong to τ_i and where they represent all the different execution environments.

It is important to remark that configuration φ must contain a valid set of execution environments. If we consider the previous example in which different cache partitions can be assigned to each core, the size of each cache partition in the configuration φ must be equal to or less than the total cache size. Moreover, the maximum number of simultaneous HRTs considered must be equal to or less than the total number of cores. In this case we say that configuration φ is *valid*.

5.2.3 The Algorithm

The main objective of IA^3 is to determine a valid set of φ to schedule a task set τ that minimizes the resources assigned to HRTs and hence enables the rest of the resources to be assigned to NHRTs, thus maximizing the hardware utilization and taking full advantage of multi-core processors.

The IA^3 is composed of two phases: the *common partitioned* phase, in which the same execution environment is considered in all cores, and the *WCET-sensitivity* phase, in which different execution environments are assigned to different cores. Figure 5.4 shows the pseudo-code implementation of IA^3 considering the multi-core processor described in this thesis, in which the execution environment is identify by: (1) the number of HRTs that run simultaneously (No_{HRT}) and (2) the size of the cache partition assigned to each core (p). Thus, the WCET-matrix of a task is a vector subspace of $z = 2$ dimensions with $m \cdot x$ vectors (where x is number of cache partition sizes, and m is the number of cores), such that given a $\hat{v} =< k, p, c >$ it associates

the WCET estimation c with the cache partition p and the number of cores k used by HRTs. Moreover, given two different vectors $\hat{v_1} =< k_1, p_1, c_1 >$ and $\hat{v_2} =< k_2, p_2, c_2 >$, the WCET-sensitivity is defined as $\hat{w} =< k_1 - k_2, p_1 - p_2, c_1 - c_2 >$, where $c_1 - c_2$ is the WCET estimation variation changing the execution environment from $< k_2, p_2 >$ to $< k_1, p_1 >$. We note that IA^3 can be generalized for additional dimensions (Section 5.5 considers execution environments identified by different parameters).

Concretely, IA^3 takes as input the task set τ, the total number of cores m and the different cache partition sizes $cache\_partitions$. As output it provides a list of valid configurations ϕ that schedule τ.

The IA^3 iterates over all $z = 2$ dimensions of the WCET-matrix, i.e. over No_{HRT} (line 2) and p (line 7), using a *depth-first search* approach [30] in increasing order, i.e. from the execution environment in the WCET-matrix that leads to the lowest WCET estimation for each task ($No_{HRT} = 1$ and p_1 = biggest size of cache partition), to the one that leads to the highest WCET estimation ($No_{HRT} = m$ and p_x which refers to the smallest size of cache partition). Note this ordering is used because we are interested in solutions that use the least processing capacity, thus maximizing the available spare capacity for NHRTs. At the beginning of every new iteration of the outermost loop (line 2), No_{HRT} determines the number of cores available (n_av_cores) for the HRTs (line 3) and a working copy (τ') of the original task set (τ) is set (line 4). Note that the selected starting point, i.e. the one that leads to the lowest WCET estimation, aims to determine the minimum number of processors that could possibly result in a solution, and then refine the solution from there.

During the *common partitioning* phase, the IA^3 tries to allocate the task set τ' using n_av_cores cores, assigning to each core the execution environment defined by No_{HRT} and p, and generating configuration φ_{ffdtmp}.

Input τ: task set
m: total number of cores,
$cache\_partitions$: cache partition sizes
Output $list_\varphi$: list of valid φ

```
1  list_φ := ∅;
2  for No_HRT in [1:m]
3    n_av_cores:=No_HRT;
4    τ' := τ;
5    tmplist_φ := ∅;
6    φ_wcstmp , φ_ffdtmp := ∅;
7    for each p_j in cache_partitions_sizes (decreasing order)
8      // Common Partition Phase
9      <φ_ffdtmp ,∅> := ffd(τ', n_av_cores , No_HRT ,p_j ,∅);
10     if (schedulable(τ ,φ_ffdtmp ∪ φ_wcstmp))
11       if (valid(φ_ffdtmp ∪ φ_wcstmp)
12         add-to-list(φ_ffdtmp ∪ φ_wcstmp ,tmplist_φ);
13       endif
14     else
15       if (p = max(cache_partitions))
16         break;
17       endif
18       // WCET sensitivity Phase
19       wst := compute-sensitivity(τ' ,No_HRT ,p_j ,p_j-1);
20       < φ_1c ,τ' > := ffd(τ' ,1 ,No_HRT ,p_j-1 , wst);
21       φ_wcstmp := φ_1c ∪ φ_wcstmp;
22       n_av_cores:= n_av_cores − 1;
23       <φ_ffdtmp ,∅> := ffd(τ', n_av_cores , No_HRT ,p_j ,∅);
24       if (schedulable(τ, φ_ffdtmp ∪ φ_wcstmp) )
25         if (valid(φ_ffdtmp ∪ φ_wcstmp)
26           add-to-list(φ_ffdtmp ∪ φ_wcstmp ,tmplist_φ);
27         endif
28       else
29         break;
30       endif
31     endif
32   endfor
33   φ_min:=find-minimum-φ(tmplist_φ),
34   add-to-list(φ_min , list_φ);
35 endfor
36 return list_φ;
```

Figure 5.4: Pseudo-code implementation of the IA^3 considering the multi-core architecture presented in this thesis with $z = 2$.

To do so, the IA^3 applies a first-fit decreasing heuristic [31] (ffd function) using the WCET-estimations c_i such that $\hat{v}_i =< k_i, p_i, c_i >\in WM$ (the WCET-matrix), $k_i = No_{HRT}$ and $p_i = p_j$ (line 9)[1]. The last parameter of the ffd function (in this case $\emptyset$) defines the criteria used to sort tasks before allocating them: $\emptyset$ indicates to use the WCET estimations. Then, if the resulting φ_{ffdtmp} (joined with the result of previous *WCET sensitivity* phases) can schedule the complete task set τ (line 10) and it is a valid configuration (line 11), i.e. the amount of resources required by τ do not exceed the actual amount of resources available in the processor, it is stored (line 12).

Let us assume that the execution environment e_2 (defined by No_{HRT} and p_j) cannot schedule τ while e_1 (defined by No_{HRT} and p_{j-1}) can. Note that the size of the cache partition considered in e_1 (p_{j-1}) is bigger than the one considered in e_2 (p_j). In this case, the IA^3 starts the *WCET-sensitivity* phase (line 18), in which it identifies those tasks whose WCET estimations suffer a higher impact when changing the execution environment from e_1 to e_2. To do so, the IA^3 computes the WCET-sensitivity $\hat{w} = \hat{v}_1 - \hat{v}_2$ for all tasks in τ', such that the first 2 dimensions of $\hat{v}_1$ and $\hat{v}_2$ are equal to e_1 and e_2 respectively, and it sorts all the tasks by the 3rd dimension of $\hat{w}$ (line 19). The task order[2] is stored in wst. Note that, since the amount of resources assigned to e_2 is less than e_1, the 3rd dimension of $\hat{w}$ will be always bigger than or equal to 0.

Then, the subset of tasks $\gamma \in \tau'$ with higher sensitivity are allocated to one core, fixing e_1 as the execution environment of that core. To do so, the ffd function uses the WCET estimations c_i such that $\hat{v}_i =< k_i, p_i, c_i >\in WM,\ k_i = No_{HRT}$ and $p_i = p_{j-1}$ and the task order defined by wst (line 20). As a result the ffd function provides the partial configuration φ_{1c}, which only contains the set of tuples $< e_i, a_i >$ assigned to γ, being $e_i = e_1$, and a new task set τ' that replaces the previous one, such that for all $t_i \in \tau'$, $t_i \notin \gamma_k$ and $t_i \in \tau$. This new φ_{1c} is then added to other allocations that have been computed during previous WCET sensitivity phases (φ_{wcstmp})(line

[1] $\hat{v}_i$ exists as all execution environments considered by IA^3 have a corresponding $\hat{v}_i$ per task

[2] We also checked ordering tasks using the $sensitivity/period$ rather than just $sensitivity$ criteria, expecting that tasks with a large decrease in utilization by virtue of having more cache, could be assigned on a core with a larger cache partition. However, both heuristics have very similar results.

21) and the number of available cores is reduced (line 22). By doing so we expect to assign the tasks γ with the highest WCET-sensitivity to a core with an execution environment e_1, so the new task set τ' may be scheduled with e_2 or less resources on the remaining cores.

Finally, the ffd function is called with the remaining tasks contained in the new τ', the current available cores and the execution environment e_2, resulting in a new partial configuration φ_{ffdtmp} (line 23). Then, it is checked whether the complete configuration made up of φ_{ffdtmp} and φ_{wcstmp} can schedule the complete task set τ (line 24). If the unified φ is valid and it can schedule τ, it is added to the $tmplist_{\varphi}$ (line 25). If τ is not schedulable, the cache partition exploration finishes (line 29).

Note that the IA^3 uses two different first fit decreasing sorting criteria: one uses the WCET estimation and it does not modify τ' (lines 9 and 23 with $\emptyset$ in the input and output), and one that uses the WCET sensitivity and generates a new τ' (line 20).

For a given No_{HRT} IA^3 generates at most one valid configuration. That is, fixing No_{HRT}, the algorithm selects the configuration, if there is one, that minimizes the total cache used. This is performed at the end of every iteration of the p_j loop, by calling the function `find-minimum-`φ (line 33) that, given the list of φs stored during the exploration of cache partition sizes p_j, selects the one that requires the smallest amount of cache. However, different number of cores (No_{HRT}) can lead to multiple configurations (φs), so the same τ can be scheduled using different execution environments and different task-to-core mappings (line 34). Then, the embedded system designer can select the proper solution according to the system constraints.

It is important to remark that the IA^3 explores invalid configurations, i.e. with more resources than the ones available in the processor, in order to reach a valid one. For example, it can be the case that the IA^3 explained above explores a configuration in which the sum of the cache partition sizes assigned to each core exceeds the total cache size. Then, the goal of the WCET sensitivity phase is to identify those tasks that require a higher cache partition and group them into a core, such that the remaining tasks may be assigned to cores with a smaller cache partition. By doing this a valid

configuration in which the sum of the cache partition sizes does not exceed the total cache size can be reached.

5.2.4 Example: Applying IA^3 to a four-core processor with partitioned cache

This section explains all the steps of the IA^3 implementation described in Section 5.2.3 considering the following input parameters: a task set τ, $m = 4$ and $cache\_partitions = [64, 32, 16, 8]$.

Let us assume that the first iteration of the common partition phase (No_{HRT} = 1, $p = 64$) results in a φ_{ffdtmp} that cannot schedule τ (line 10) (no WCET sensitivity phase has been executed, so φ_{wctmp} is empty). Then, since $p_1 = 64$, i.e. the biggest cache partition size (line 15), IA^3 breaks out of the p_j loop (line 16). The same conditions are verified in the next iteration (No_{HRT} = 2, p_1 = 64). Instead, with No_{HRT} = 3 and $p_1 = 64$, τ is schedulable (line 10), but the resultant φ_{ffdtmp} is not stored because it is not valid (line 11) (φ_{ffdtmp} assigns 64 KB to each core but the total available cache size is 64 KB). The same happens in the next iteration (No_{HRT} = 3 and $p_2 = 32$). Note that in these two cases the IA^3 is exploring invalid configurations. Instead, when performing the common partition phase with No_{HRT} = 3 and $p_3 = 16$, τ is not schedulable with the resultant φ_{ffdtmp}. In this case, the condition stated in line 15 does not hold, so the WCET sensitivity phase starts.

In the *WCET-sensitivity* phase, the WCET-sensitivity vectors $\hat{w}$ are computed (line 19), such that No_{HRT} = 3, $p_3 = 16$, and $p_2 = 32$, storing the sort criteria in wst. Table 5.1 shows an example of the computation of the WCET-sensitivity of the three highest sensitive tasks in τ', please note that the first z dimensions of $\hat{w}$ are not considered. Then, the IA^3, using a first-fit decreasing heuristic, allocates the tasks with the highest WCET-sensitivity to a single core, and it fixes No_{HRT} = 3, p_2 = 32 as the execution environment of this core (line 20). This allocation is stored in φ_{1c} and the set of tasks that have not been allocated are stored in τ'. Then, φ_{1c} is joined with the results of the previous WCET sensitivity phases, φ_{wcstmp} (which in this case is empty), and the number of available cores is reduced to 2. Finally, a first fit decreas-

Table 5.1: WCET-sensitivity computation ($\hat{w}$) of 3 tasks considering $k_1 = 3$, $k_2 = 3$, $p_1 = 16$ and $p_2 = 32$. Ordered by the 3-rd dimension

	$\hat{v}_1$	$\hat{v}_2$	$\hat{w}$
task$_6$	$< 3, 16, 132 >$	$< 3, 32, 101 >$	$< -, -, 31 >$
task$_2$	$< 3, 16, 100 >$	$< 3, 32, 75 >$	$< -, -, 25 >$
task$_5$	$< 3, 16, 256 >$	$< 3, 32, 235 >$	$< -, -, 21 >$

ing algorithm is applied on the remaining 2 cores, with No_{HRT} = 3, $p_3 = 16$ and the new task set τ'. The resultant φ_{ffdtmp} is joined with φ_{wcstmp}, i.e. considering 3 cores, with respectively 32, 16 and 16 KB of cache partition. Let us assume that τ is schedulable. Thus, since the new φ is valid, it is added to $tmplist_{\varphi}$ (line 26).

In the next iteration (No_{HRT} = 3 and $p_4 = 8$) the common partition phase is started again, but considering 2 available cores and the new τ' (line 9). In this case, the resultant φ_{ffdtmp} is joined with the previous φ_{wcstmp} (line 10), i.e. considering 3 cores, with respectively 32, 8 and 8 KB of cache partitions. Let us assume, τ is schedulable and the resultant φ is valid, so it is added to the list $tmplist_{\varphi}$. However, in the next iteration (No_{HRT} = 3 and $p_5 = 4$) the common partition phase results in a φ that cannot schedule τ so the WCET sensitivity phase starts again. In the WCET sensitivity phase, the resultant joined φ_{ffdtmp} and φ_{wcstmp} is 3 cores, with respectively 32, 8 and 4 KB of cache partition, which cannot schedule τ so IA^3 breaks out of the p loop (line 29).

Once the cache size dimension p_j has been fully explored, the function $find - minimum - \varphi$ finds, among all the stored φ (3 cores, with respectively 32, 16 and 16 KB of cache partition and 3 cores, with respectively 32, 8 and 8 KB of cache partition), the one with the smallest amount of cache used, i.e. 3 cores, with respectively 32, 8 and 8 KB of cache partition. Thus, in order to reach this valid configuration, the IA^3 has explored invalid ones, i.e. assigning to each core 32 KB. Then, a new iteration starts with $No_{HRT} = 4$. The same steps are carried out, but they are omitted here due to lack of space.

It is important to remark that this implementation can lead to multiple allocation solutions. That is, the same τ can be scheduled, for example, using a configuration with three cores and cache partitions equal to 32, 8, 8 KB, and with a configuration with four cores but with less cache, allowing the embedded system designer to select the best solution according to the system constraints.

5.3 Test Methodology

Evaluating the effectiveness of the IA^3 requires a methodology of generating task sets. However, in order to perform an effective evaluation of our proposal, it is important to consider tasks with different levels of WCET-sensitivity. To that end, we first characterized the WCET-sensitivity of the EEMBC Autobench [77] (see Chapter 2) benchmark suite under different execution environments considering the multi-core architecture introduced in Chapter 3 and 4.

In order to characterize their WCET-sensitivity, we computed the WCET estimation of each EEMBC benchmark considering that each request to a shared resource is delayed by UBD cycles, under 20 different execution environments, as a result of assigning different cache partition sizes (128 KB, 64 KB, 32 KB, 16 KB and 8 KB) and varying the number of HRTs that access simultaneously the shared bus and the memory controller, i.e. different $UBDs$ corresponding to No_{HRT} from 1 HRT to 4 HRTs. Hence, the WCET-matrix of each benchmark results in a 2-dimensional vector space indexed by the cache partition size and the number of simultaneous HRTs.

All WCET estimations were computed using our multi-core simulator infrastructure and RapiTime [8] tool, as already described in Chapter 2.

Once the WCET-matrix of each benchmark had been generated, we analyzed the results according to WCET-sensitivity, i.e. how the WCET estimations vary across the different execution environments, creating 3 sensitivity groups called *High, Medium* and *Low Sensitivity*. Each group, whose WCET-matrix is the average of the WCET-matrices of all tasks that form the group, is formed by the following EEMBC bench-

Table 5.2: The WCET-sensitivity range of each sensitivity group considered in the generation of the tasks' WCET-matrix

	Cache Partition	Number of simultaneous HRT
High Sensitivity	0.10 - 0.25	0.10 - 0.50
Medium Sensitivity	0.07 - 0.14	0.05 - 0.18
Low Sensitivity	0.00 - 0.03	0.00 - 0.01

marks: High (*aifftr*,*aiifft*,*cacheb*), Medium (*aifirf*, *iirflt*, *matrix*, *pntrch*), Low (*a2time*, *basefp*, *bitmnp*, *canrdr*, *idctrn*, *puwmod*, *rspeed*, *tblook*, *ttsprk*).

For each sensitivity group, we identified the WCET-sensitivity between two adjacent execution environments of the WCET-matrix. To do so, starting from the execution environment with 1 HRT and 128KB of cache, we compute the WCET-sensitivity as we reduce the cache size, and increase the number of simultaneous HRTs. Table 5.2 shows the range of the WCET-sensitivity when the cache size and the number of simultaneous HRTs changes in each group.

5.3.1 Randomly-Generated Tasksets

We randomly-generated task sets based on the WCET-sensitivity ranges shown in Table 5.2: Starting from an *initial WCET* (corresponding to an execution environment with 32KB of cache partition and 1 HRT) we generate a complete WCET-matrix by applying the variations among the different execution environments that resembles the WCET-matrix of a given sensitivity group. In order not to generate identical WCET-matrices, we use a uniform random-generator that considers the maximum and the minimum variance of the corresponding groups.

We assume two different classes of initial WCETs: a *High Utilization* class, corresponding to a task with an utilization ranging between 0.3 and 0.6; and a *Low Utilization* class, with an utilization ranging between 0.1 and 0.3. The initial WCET is also generated using an uniform random-generator that considers the corresponding utilization ranges. For these simple experiments, we are interested only in the effect

Table 5.3: Percentage of type of tasks in the task set considering the initial WCET class and the sensitivity group

	High u (30%).	Low u (70%)
High Sensitivity (20%)	$20\% \times 30\% = 6\%$	$20\% \times 70\% = 14\%$
Medium Sensitivity (30%)	$30\% \times 30\% = 9\%$	$30\% \times 70\% = 21\%$
Low Sensitivity (50%)	$50\% \times 30\% = 15\%$	$50\% \times 70\% = 42\%$

of different execution environments on the partitioning. We therefore fixed all of the task periods to have the same value (and the deadlines equal to the periods).

Finally, in order to generate task sets composed of tasks with different requirements, we assume that 30% of the tasks belong to the high utilization class and 70% belong to the low utilization class. Moreover, from the characterization of the EEMBC Automotive benchmark suite we assume that 20% of the tasks in the taskset have a high sensitivity, 30% have a medium and 50% have a low one. Note that, each task can belong to a high or a low utilization class, and its WCET-matrix can have a WCET-sensitivity that resembles a high, medium or low sensitivity group. Table 5.3 shows the percentage of tasks considering the sensitivity group and the initial WCET. The total utilization of the generated tasksets, i.e. the target U_t, is computed using the WCET value of each task when running in the initial execution environment.

5.4 Results

This section evaluates the IA^3. To do so, we have generated 10 series of 10,000 task sets, each composed of 10 tasks, using the methodology explained in Section 5.3. Concretely, for each series, we have fixed a target utilization U_t (i.e. u_{sum} computed considering the initial WCET of each task in the task set t), ranging from 2.9 to 3.9 with an increment step of 0.1. To do so, we used the simple naive and unbiased method described in [32], [33]: we generate 9 out of 10 random WCET matrices in high and low utilization ranges and then check if the remaining task with utilization equal to $U_t - u_{sum}$ belongs to the low utilization class; if not we discard the task

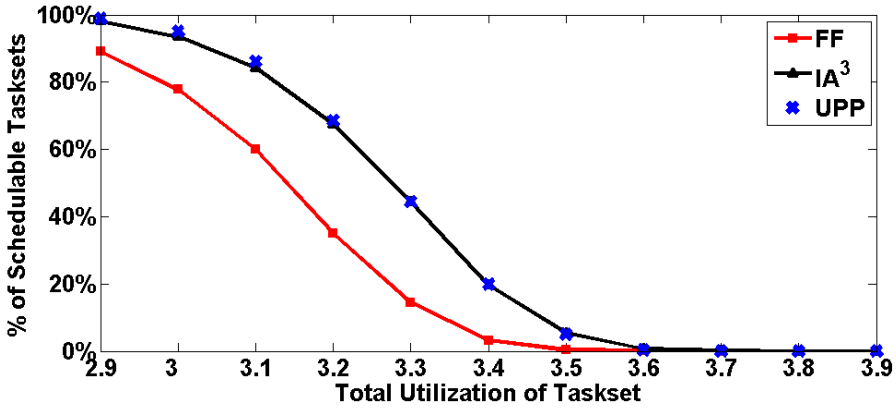


Figure 5.5: Percentage of schedulable task sets when applying *FF, UPP* and *IA^3* in a 4-core processor, ranging the total utilization from 2.9 to 3.9 with an increment step of 0.1

set and start the process again. Moreover, we also consider a real task set composed of the sixteen EEMBC Autobench benchmarks (described in Section 5.3), ranging its total utilization from 2.7 to 3.3 with an increment step of 0.1.

The target multi-core architecture considered is a 4-core processor with a 128 KB partitioned cache. The architecture allows assigning dynamically to each core a private partition of cache ranging in size from 4 KB to 128 KB.

We also compare IA^3 to an idealized scheme that assumes each core has the same cache partition size, and checks a necessary feasibility condition (labeled *UPP*). In this case we simply check that, given an execution environment with k cores and a cache partition size assigned to each core identified by p, that for all $\hat{v} =< k, p, c >$, the cumulative utilization $\sum_{u \in \hat{v}} \leq k$. If this test fails, then the configuration could not possibly be schedulable as it has a higher utilization that the available processor capacity. *UPP* represents an upper bound on the maximum possible performance of a global scheduling approach with no migration overheads.

Moreover, we also compare IA^3 with a partitioned scheduling algorithm that uses a First-Fit Decreasing Heuristics (labeled as *FF*) [31] that assigns to each core the same execution environment such that the resources used are minimized, i.e. the

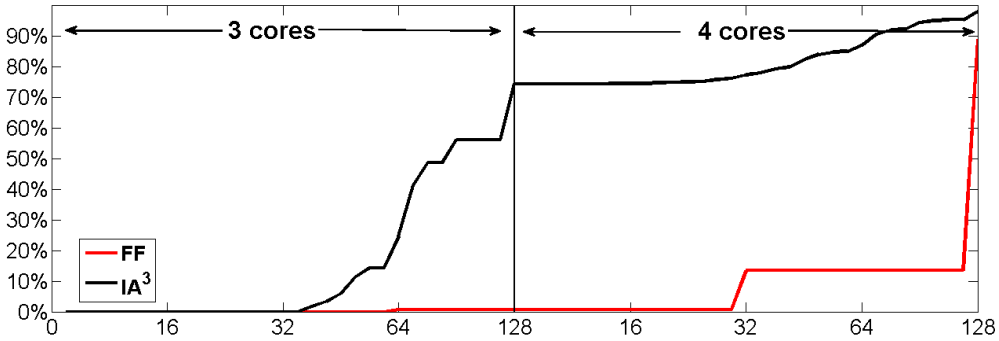


Figure 5.6: 10,000 Task set with Total Utilization 2.9

cache partition assigned and the number of cores. For *FF* we are effectively applying only the *common partitioning* phase of the IA^3.

Figure 5.5 compares the percentage of schedulable task sets using IA^3, *FF* and *UPP*, increasing the U_t from 2.9 to 3.9. As expected, our allocation algorithm is able to consistently schedule more task sets than *FF*, while achieving almost the same ratio of scheduled task sets as the hypothetical upper bound given by *UPP*. On average, when compared to *FF*, IA^3 is able to schedule 20% more tasksets, with a maximum difference of 32% when considering $U_t = 3.2$.

However, the benefit of IA^3 is not only the number of schedulable task sets, but the resources used to schedule them. Figures 5.6 and 5.7 show the cumulative distribution function of the resources required to schedule task sets with a U_t equal to 2.9 and 3.3 respectively, when using IA^3 and FF. Concretely, the figures show the percentage of scheduled task sets with a certain number of cores (X-axis division on Figure 5.6) and the size of cache partition assigned (in KB) to each core. For example, when considering a total utilization equal to 2.9 (Figure 5.6), the IA^3 is able to schedule more than 70% of the tasksets with only 3 cores, while, in the case of *FF*, less than 5% of the tasksets have been scheduled with 3 cores. With a total utilization of 3.3, both schemes require 4 cores to schedule all tasksets. However, IA^3 is able to schedule more than 30% of the tasksets without requiring the complete cache size, i.e 128 KB. Therefore, IA^3 generates a very efficient partition scheme by assigning different environments to different cores and so reducing the resources required. Reducing the

amount of resources assigned to HRTs is particularly beneficial in *mixed-application workloads*, because the resources not used by HRTs can be assigned to NHRTs. For example, in Figure 5.6, 50% of the tasksets have been scheduled using only 3 cores and less than 96 KB of cache partition, so the rest of the resources, i.e. one core and 32 KB of cache partition can be assigned to NHRTs. By contrast, in the case of *FF*, only 15% of the tasksets are able to be scheduled with less than 64 KB, but requiring 4 cores, and so giving less resources to NHRTs.

Finally, Table 5.4 shows the number of cores and the total cache partition size required to schedule the real task set composed of the sixteen EEMBC benchmarks when using FF, UPP and IA^3, with utilization U_t scaled from 2.7 to 3.3. IA^3 requires consistently less resources than the other schemes. Notice that IA^3 performs better than UPP because UPP assumes that all cores have an equally sized cache partition to ensure, after a task migration, that the destination core has the same cache partition as the source one.

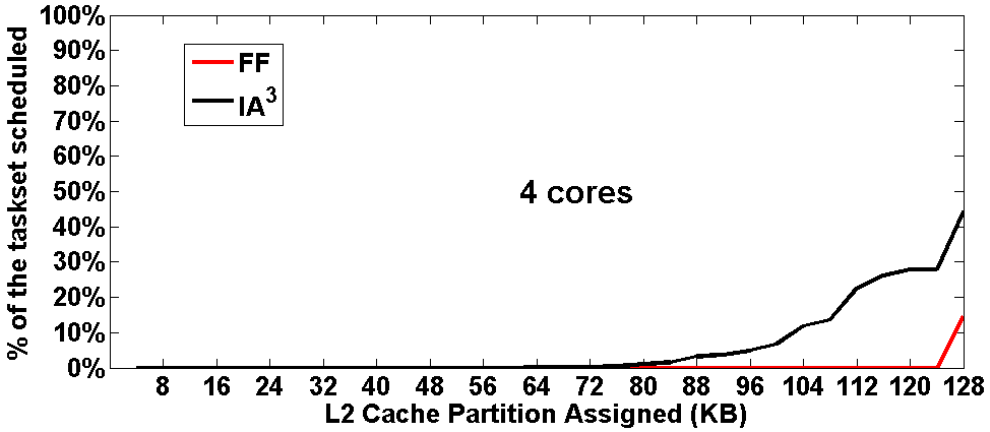


Figure 5.7: 10,000 Task set with Total Utilization 3.3

5.5 Additional Considerations

The IA^3 permits a WCET-matrix of z dimensions, $z \subseteq \mathbb{N}$. However, in this thesis we have considered only a simple WCET-matrix of $z = 2$ dimensions that includes the

Table 5.4: Number of cores and cache partition size (in KB) required to schedule a task set composed of the 16 EEMBC benchmarks when applying *FF, UPP* and *IA*^{3}, ranging the total utilization from 2.7 to 3.3 with and increment step of 0.1

	FF		UPP		IA^3	
U_t	cores	cache (KB)	cores	cache (KB)	cores	cache (KB)
2.7	4	32	3	48	**3**	**40**
2.8	4	32	4	32	**3**	**52**
2.9	4	32	4	32	**4**	**20**
3.0	4	64	4	32	**4**	**28**
3.1	4	64	4	64	**4**	**32**
3.2	4	64	4	64	**4**	**32**
3.3	*not schedulable*		4	64	**4**	**40**

number of simultaneous HRTs and the size of the cache partitions assigned to each core, which improves task set schedulability when compared to a classical partitioned scheme. However the benefit of making task allocation aware of the impact that different execution environments have on the WCET is potentially much bigger. For example, the WCET-matrix can potentially be applied to heterogeneous multi-core processors, in which each core has a different computational power. In this case, an extra dimension is required to measure the WCET of the different tasks under the different cores, Thus, the IA^3 will consider a WCET-matrix of $z = 3$, assigning to the most powerful cores those tasks with the highest WCET-sensitivity.

An important future research line of IA^3 is to consider a preemptive scheduling approach. Some works introduce a factor in the WCET estimation [13,43,84] in order to consider the impact of inter-tasks interferences introduced by preemption due to the cold start when the task is resumed. These factors can be introduced in the WCET-matrix to allow IA^3 to use preemption. Similarly, the WCET-matrix can consider other execution environment parameters such as the number of TDMA-slots assigned to a HRT, different frequencies at which the processor can run, etc.

However, it is important to notice that adding additional dimensions to the WCET-matrix increases the computational complexity of IA^3 because the current implementation of the algorithm effectively searches over all combinations of parameters (e.g. number of cores, and cache partition size). The complexity can be reduced by applying a search algorithm optimization, for example, Emberson et. al. [38]. However, as WCET estimates need to be obtained for each task in all the execution environments of the WCET-matrix, in practice the dimension of the matrix will be limited to a small number of important factors such as cache size, number of simultaneous HRTs, computational speed, etc.

5.6 Related Work

There are two main strands of research in multiprocessor scheduling [34], reflecting the ways in which tasks are allocated to cores. Partitioned approaches allocate each task to a single core, dividing the problem into one of task allocation (bin-packing) followed by single processor scheduling. In contrast, global approaches allow tasks to migrate from one core to another at run-time. In partitioned scheduling finding an optimal task allocation is an NP-hard problem in the strong sense [39] and so non-optimal solutions derived from the use of bin-packing heuristics are typically used.

Dhall and Liu [35] proposed two heuristic assignment schemes: *rate-monotonic next-fit* and *rate-monotonic first-fit* based on the next-fit and first-fit bin-packing heuristics. In both schemes, tasks were sorted in decreasing order of their periods and assigned to a so-called current core until the schedulability condition was not achieved. Davari and Dhall [31] proposed a variation of the rate-monotonic first-fit, the *first-fit decreasing utilization*, in which tasks were sorted by their utilization. Similarly, Oh and Son [68] proposed a *best-fit decreasing utilization* in which tasks were allocated to the cores that would then have the least remaining utilization. However, all these approaches did not consider the effect that interferences have on the WCET estimation. Jain et al. [47] proposed a scheduler for a two-way simultaneous multithreading processor in which the WCET estimation for each soft real-time task depends on the

co-running task. In this case, there is a single hardware configuration so the execution environment of each task only depends on the particular soft real-time task with which it is executed. This approach has not been applied to hard real-time systems.

Global scheduling approaches have also considered the effect of the execution environment on WCET estimation. Holman et al. [43] realized that scheduling all processors simultaneously can result in a heavy bus contention at the start of the scheduling quantum due to reloading data into caches. To reduce such bus contention they presented a new global scheduling model that distributes more uniformly the bus traffic by shifting the scheduling decisions of the different tasks. Anderson et al. [13] proposed a new task organization, called *megatasks* as a way to reduce the miss rate in shared caches on multi-core platforms. Megatasks artificially inflate their utilization factors in order to consider cache contention. Both techniques have good results in the average case, reducing the bus and cache contention. However, interferences remain uncontrolled, and they are still unknown in the worst case.

Kato et al. [52] considered the use of multiple execution time estimations (EET) using U-Link Scheduling to schedule a task set in a soft real-time simultaneous multi-threading (SMT) environment. Concretely, they provided the EET of a task in an SMT execution environment based on WCET estimation in a single-threading execution and the execution variation introduced by the details of the other co-scheduled tasks. The research presented in this thesis differs from that of Kato et al. in that our approach focuses on the WCET estimations of each HRT considering only the parameters that determine the execution environment in which a task runs, rather than on estimations of the execution time variations due to details of the various tasks when scheduling them in a simultaneous multi-threading environment.

We note that multiple execution times per task have also been considered as part of approaches to improve quality of service on single processors [48].

The work of Pellizzoni et al. [73] considers the effect of contention for access to main memory on WCETs of tasks. This is done by abstracting the accesses of each interfering core into an arrival curve which combined with peripheral traffic gives a delay bound for the task under analysis. Again this means that the analysis is depen-

dent on the individual characteristics of the tasks, rather than just the characteristics of the execution environment.

Banus et al. propose a dual priority algorithm to schedule real-time tasks in a shared memory multiprocessor using a global scheduler for both periodic and soft-aperiodic tasks. Migration is allowed between processors [17]. The same authors in [16] propose a technique to allocate aperiodic tasks after having statically allocated the periodic hard real-time tasks.

To the best of our knowledge, IA^3 is the first task allocation algorithm that considers a matrix of WCET estimations for the different execution environments in which tasks can run.

5.7 Summary

In this chapter we presented IA^3, an new off-line interference-aware allocation algorithm for multi-core processors. The IA^3 is based on two novel concepts: the WCET-matrix and the WCET-sensitivity.

The WCET-matrix is a z-dimensional vector space that contains the WCET estimations of each task under different execution environments. Each dimension represents a different execution environment parameter (e.g. number of simultaneous HRTs, size of the cache partition assigned to each core, etc.) that makes the WCET estimation vary. The WCET-sensitivity measures the impact of changing the execution environment, i.e. modifying the parameters that define it, on the WCET estimation. These two concepts allow IA^3 to consider not just a single WCET estimation but a set of WCET-estimations generating a more efficient partitioning.

For the target multi-core architecture considered in this thesis an execution environment can be identified based on the number of simultaneous HRTs and the size of the cache partition assigned, and so having a WCET-matrix of $z = 2$ dimensions.

In our simple experiments, comparing IA^3 with a classical first-fit decreasing partitioning scheme under the target architecture, IA^3 is able to schedule, on average, 20% more tasksets. Moreover, when considering the amount of resources used, IA^3

schedules more than 70% of the tasksets with only 3 cores, while the first-fit partitioning approach is able to schedule only 5% of the tasksets with 3 cores. Finally, using IA^3 50% of the tasksets that are scheduled using 3 cores require less than 96 KB of cache, and so leaving one free core and 32KB of cache to NHRTs. Instead, using first-fit only 15% of the tasksets that are scheduled in four cores require less than 64 KB, which does not leave any core to NHRTs.

CHAPTER 6

A First Step Towards Predictable Parallel Applications

To exploit the full benefits of multi-core processor and increase the performances, applications can be parallelized. So far, parallel programs have been developed in high performance computing but not in hard real-time systems. This chapter is a first attempt of achieving higher performance for HRTs by supporting predictable parallel hard real-time applications.

6.1 Introduction

In the previous chapters of this thesis we have been focusing on multi-programmed workloads composed of HRTs and NHRTs, however higher performance can be achieved using multi-core architectures if applications are made *multi-threaded* by exploiting Thread-Level Parallelism (TLP): applications are split into threads that run in parallel on different cores and synchronize whenever they need to communicate. This chapter focuses on the software-pipelined parallel programming technique, i.e. a particular implementation of the producer-consumer programming model. In the software-pipelined approach, a stream of data is passed through a sequence of pipeline stages

and each stage performs a step of the overall computation. These stages can be implemented as individual threads. Since the different stages that compose the application access different portions of data, the stage threads can be executed in parallel on different cores of a multi-core processor.

However, each thread may also suffer from inter-thread interferences like accesses to shared resources, which can potentially reduce the performance benefits brought by TLP. The shared resource with the highest impact on the WCET is the main memory (see Chapter 4). Therefore, the inter-thread interferences generated by simultaneously executed threads when accessing the memory may destroy the advantage of TLP. That is also the case of the software-pipelined approach. In fact, the data that has been processed by stage n is stored in memory, so it can be accessed by the next stage $n + 1$. Concurrently, stage n starts to process a new portion of data that has been stored also in memory by stage $n - 1$, originating memory contention between the two stages.

Shared caches have traditionally been used to alleviate the impact of accessing the main memory in multi-core processors. However, their use in hard real-time scenarios complicates considerably the WCET analysis or make it even infeasible, as shared caches generate storage interferences: a thread evicts data of another one, originating additional misses and it potentially delays the execution time of the second thread, and so the benefits of caches are lost. Partitioned caches (see Chapter 3) have been proposed to increase the predictability of shared caches in multi-core processors by assigning private portions of the cache to each thread and so eliminating the storage interferences. However, cache partitioning techniques are not suitable for the software-pipelined pattern, due to the fact that moving the data among the different stages of the application, would require to copy such data among the different cache partitions in order to have the proper data stored in the correct cache partition.

The most relevant related work is the one presented in [81] where the authors provide a high-level preliminary WCET analysis of the communication and synchronization patterns for a data parallel application.

This chapter proposes a software/hardware *cache partitioning* approach that exploits the benefits of TLP by reducing the memory contention in hard real-time pipelined parallel applications guaranteeing a predictable timing behavior. The hardware technique, called *bankization* (see Chapter 3), which allows to dynamically assign private cache banks to cores at run-time, has been extended with a software interface which ensures a correct behavior of the cache partition remapping mechanism and provides the foundation for developing multi-threaded applications using the bankization technique. By doing so, the data generated by one stage and stored into its corresponding cache partition can be easily accessed by another stage executed in another core, without requiring the copy/movement of the entire chunk of data among the different cache partitions.

6.2 The Software-Pipelined Parallel Programming Model

Among the different existing parallel programming models [51] we selected the software-pipelined parallel programming technique because it is simple and it is the one that can take higher advantage of our hardware proposals. In particular we were able to provide a solution that could allow the execution of parallel applications satisfying the real-time constraints by effectively using the *bankization* cache partitioning mechanism.

The software-pipelined parallel programming technique, i.e. a particular implementation of the producer-consumer programming model, is a very popular programming model in which a stream of data is passed through a sequence of pipeline stages, where each of them performs a part of the overall computation. With the exception of the first stage of the pipeline, which is fed with the original input data, the execution of each following stage requires the output of the preceding one. Hence, the data moves through the pipeline in discrete clocking steps. The length of such a clocking step and herewith the performance of the pipeline, is defined by the longest pipeline stage i.e., by the pipeline stage with the highest WCET in case of a hard real-time system.

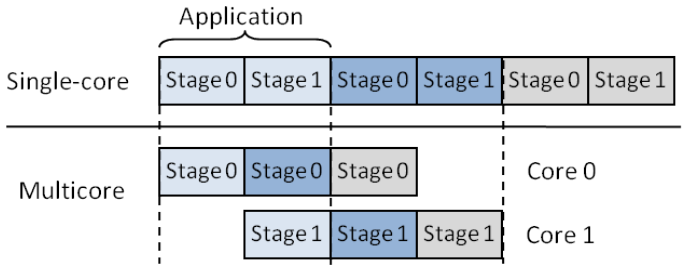


Figure 6.1: The original application is split into two stages. Each iteration of the application is represented in a different color (light-blue, blue and grey). In a single-core processor, each stage runs one after the other. Instead, in a multi-core processor, each stage can run in a different core

Improving the execution speed of the longest pipeline stage also improves the overall performance of the pipeline while improving any other pipeline stage is useless.

As an example, we consider the two-staged pipeline shown in Figure 6.1. In the case of a single-core, the processor has to execute the two stages sequentially and also multiple times to process a stream of incoming data portions. On a multi-core system, the two pipeline stages can be executed simultaneously working on different iterations of the incoming data stream. This means that the second iteration of the pipeline stage 0 starts processing the second data portion of the input stream while, in parallel, stage 1 performs the second processing part of the first data portion.

By doing so, it would be reasonable to expect that, without considering the first and the last stages of the execution, the execution time of the application can be reduced by a factor of two. However, because of inter-thread interferences accessing the main memory, the performance benefits of parallel pipelining can be significantly reduced. That is, the data processed by stage 0 is stored in main memory, so stage 1 can access it. At the same time, stage 0 starts to process a new stream of data also stored in main memory, generating memory contention between the two stages.

Traditionally, caches have been used to alleviate the impact of memory interferences. If we consider the example above, the data processed by the stage 0 in core 0 should be available to the stage 1 in core 1 without requiring to access the main

memory, resulting in a reduced number of accesses to main memory and hence memory interferences. However, concurrently to the execution of stage 1, stage 0 is also executed for the next iteration, generating *storage interferences* among the two stages.

These storage interferences lead to an unpredictable cache state at any point in time of the execution. Hence, a tight WCET estimation is infeasible as the cache must be assumed to produce cache misses only.

6.3 Our Proposal

In this section we propose a new technique to propagate data through a software-pipeline using our multi-core processor described in Chapters 3 and 4 that implements dynamic cache partitioning. Besides a performance increase compared to a single-threaded version of the same application, the pipelined version is still suitable for hard real-time systems.

6.3.1 Bankization: A Dynamically Partitioned Cache

Cache partitioning has been proposed to completely eliminate storage interferences by splitting the cache into private portions, each assigned to a different thread. Unfortunately, this makes each partition private to each thread and so not visible to the rest of the threads. Thus, if we consider the example of Figure 6.1, and we assign a different cache partition to each stage, the storage interferences would be eliminated, but the data generated by stage 0 would be invisible to stage 1. A naive solution is to copy the data from the cache partition assigned to stage 0 to the cache partition assigned to stage 1. However, the overhead introduced could reduce the performance benefits of the pipelining.

The dynamically cache partitioning technique called *bankization* that is integrated in our architecture (see Figure 4.7) allows to assign a subset of the total number of cache banks to each thread so that no other thread can access it (see Chapter 3 for more details).

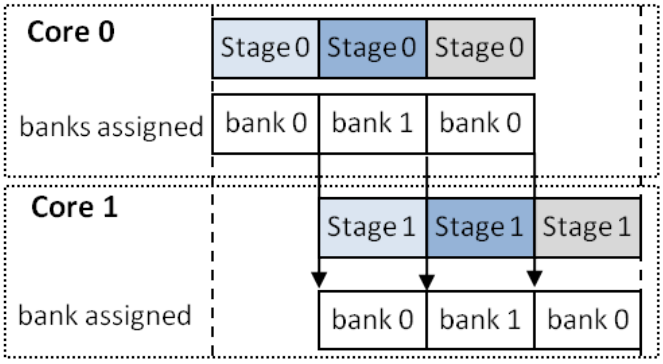


Figure 6.2: A two-stage pipeline application running in a two-core processor. Bankization allows to reassign the bank used by stage 0 to stage 1

To do so, bankization introduces the *Bank Remapping Unit* (BRU) that determines the bank assigned to a given thread based on the thread identifier and the accessed memory address (the destination bank of any memory request is contained inside its address). By changing the bank assignment defined inside the BRU, bankization allows to remap the banks assigned to the different threads at runtime such that the data visible to one thread can be accessed by another one without requiring the movement of data among banks.

Let us consider the pipelined example shown in Figure 6.2. The first stage of the pipeline, stage 0 (light-blue box) is assigned to an empty bank: bank 0, such that no other thread can access it. Then, once stage 0 has finished, bank 0 is re-assigned to stage 1 (light-blue box), such that it can access the data stored by stage 0. At the same time, a new empty bank (bank 1) is assigned to a new execution of stage 0 (blue box) such that it can compute a new stream of data without interfering with bank 0. Once stage 1 (light-blue box) has finished its computation, bank 0 can be re-assigned to a new execution of stage 0 (grey box) as an empty bank, and so all the content will be invalidated.

6.3.2 System Software Bankization Interface

For the work presented in this Chapter we have used the MERASA system-level software [91]: it contains functionalities of a real-time operating system (RTOS) and it provides the proper interfaces to implement our techniques.

The memory management of the MERASA system-level software minimizes interferences of different threads by providing a flexible two-layered memory management. The first layer allows pre-allocation of memory regions to threads while the second layer is in charge of memory management inside each thread. The pre-allocation of the heap regions is used to guarantee the isolation of different threads' memory regions [59]. After the pre-allocation of memory regions to the threads, the dynamically partitioned cache is configured accordingly.

In parallel programming models, such as the software pipelined model considered in this Chapter, multiple threads of the application need access to a shared heap memory region for data exchange. Using the pure bankization technique would not allow to access shared data because memory regions are visible only to a single thread.

However, in order to guarantee the correct functional behavior of bankization as well as to provide time predictability we introduced two types of heap regions: *private heap* and *shared heap*. Both types of regions only allow accesses of one thread at a time (during one clocking step) but the shared regions can be exchanged between pipeline stages. To reach that, it is required to: (1) isolate the shared heap regions from the private heap regions such that accesses to the shared heap do not collide with private heap accesses inside the same bank, and (2) synchronize the accesses to the shared data. We enhanced, in collaboration with University of Augsburg, the MERASA system-level software to address both requirements.

First, we provide a mechanism to split the heap memory region of the application into two independent regions: the *private heap region* and the *shared heap region*. The private region uses a real-time capable two-level allocation mechanism [59] to maximize the isolation of different threads' memory regions. Instead, in the shared heap region, which allows the access of multiple threads, the two-level allocation is reduced to one real-time capable allocation level. It is important to remark that the

shared allocation during the application's execution must be performed in a mutually exclusive way to keep the state of memory consistent. Nevertheless, we are able to guarantee timing predictability, as the number of potentially waiting threads, i.e. the list of HRTs is bounded by the number of cores in our architecture.

Moreover, in order to avoid storage interferences among different accesses to the private and shared heap regions, we extended the BRU (explained in Chapter 3) such that both regions are mapped into different banks. To do so, it is required to consider only few most significant bits of the address defined by the linker, that distinguish the two regions. The information about the heap region type is set using the *set_privateheap_size* and *set_sharedheap_size* functions that control the BRU based on the given cache partition size of each region.

Second, we guarantee that a bank used by one thread is not remapped until this thread has finished. However, such a synchronization is not done at a single stage basis but at the application basis. In other words, the bank remapping has to be done once all stages that run simultaneously have finished. Such a constraint is fundamental to allow the computation of the WCET estimation, as the WCET of each parallel phase can be expressed as the maximum WCET of all different stages. Section 6.4 analyses in detail the WCET estimation of a pipelined parallel application. We have integrated the following functionalities into the MERASA system-software, which will ensure the correct behavior of the cache by synchronizing the bank remapping at the end of all threads:

- *init_pipeline*: This function sets the number of stages and it assigns to the first stage/thread of the pipelined parallel application its corresponding set of banks based on the cache size given to each stage. The size is forced to be a multiple of the size of a cache bank. Moreover, it initializes two barriers for synchronization which are used in the *clock_edge* function. This functions uses the *set_sharedheap_size* to identify the set of banks as shared heap region.

- *start_pipeline*: It stalls each thread until the data coming from the previous thread is available. This function is composed of a loop controlled by the num-

ber of stages that have not being executed yet for the first time. Each iteration, the *clock_edge* function is called.

- *stop_pipeline*: This function does the same as start_pipeline but at the end of the execution. Thus, it stalls each thread until all the subsequent threads finish.

- *clock_edge*: It is the core of the software-pipelining technique and ensures the correct setting of the bankization parameters and the synchronous start of all pipeline stages (threads). Two barriers are required for this purpose. The first barrier ensures that all stages have finished their computation. When all stages have reached that barrier, the remapping of the banks is performed. After that, the second barrier ensures that the new bank assignment has finished and all threads start execution of the next iteration. The first pipeline stage is the one in charge of generating the banks' reassignment.

Moreover, the MERASA system software is equipped with time predictable synchronization primitives, which implement the barriers required for the bankization interface [82].

Figure 6.3 shows an example of using the software pipelining interface in a n-stage pipelined parallel application. The pipelining mechanism is initialized using *init_pipeline* (line 2). At the beginning of each stage, e.g. stage n, the function *start_pipeline* (line 9) stalls it until its previous stage, stage $n-1$, finishes. This function call is also responsible of reassigning the new bank when the computation starts with the data produced by stage $n-1$. Once the stage has performed its computation (line 11), it is stalled again in the *clock_edge* (line 12) waiting all stages to finish in order to perform the remapping. Finally, when the pipeline finishes, *stop_pipeline* (line 14) stalls all stages until the last stage finishes.

```
1  void main() {
2    init_pipeline(n, 4096);
3    stage 0 in core 0;
4    stage 1 in core 1;
5       ...
6    stage n in core n;
7  }
8  stage n {
9    start_pipeline(n);
10   while (true) {
11     do_stage n;
12     clock_edge(n);
13   }
14   stop_pipeline(n);
15 }
```

Figure 6.3: An *n*-stage pipelined parallel application using the bankization interface

6.4 WCET Analysis of a Software-Pipelined Parallel Application

6.4.1 WCET Estimation Without Cache

In the WCET analysis of a pipelined application it is required to consider the longest execution time of all the stages that compose the application, considering the impact that inter-thread interferences have on the WCET due to the parallel execution of multiple stages. However, the impact that interferences have along the execution is different depending on the execution phase in which they are: the initialization phase (*prologue*), finalization phase (*epilogue*) and the central phase of the execution (*kernel*).

Figure 6.4 shows a four-stage pipelined application that is executed five times (each color represents a different iteration). Let us assume that $WCET_n$ is the computed WCET estimation of the stage n. However, this WCET is not the same along the execution of the application, because the number of stages that run simultaneously

changes in the prologue and epilogue phase. Therefore, it is required to consider the WCET in the different phases. Let us assume that the $WCET_n(m)$ is the WCET of stage n when running with m simultaneous stages. In other words, $WCET_n(m)$ considers the impact that inter-thread interferences have on the WCET estimation stage n introduced by the other stages. Thus, the WCET estimation of each stage can be represented as:

- During the prologue phase, stage 0 runs alone and simultaneously with one and two more stages (stage 1 and 2). Hence, the WCET of the prologue ($WCET_{pro}$) can be expressed as:

$$\begin{aligned} WCET_{pro} = WCET_0(0) + \\ max\{WCET_0(1), WCET_1(1)\} + \\ max\{WCET_0(2), WCET_1(2), WCET_2(2)\} \end{aligned} \tag{6.1}$$

- During the epilogue phase, stage 3 runs alone and simultaneously with one and two more stages (stage 1 and 2). Hence, the WCET of the epilogue ($WCET_{epi}$) can be expressed as:

$$\begin{aligned} WCET_{epi} = WCET_3(0) + \\ max\{WCET_2(1), WCET_3(1)\} + \\ max\{WCET_1(2), WCET_2(2), WCET_3(2)\} \end{aligned} \tag{6.2}$$

- Finally, during the kernel phase, all stages run simultaneously. Hence, the WCET of the kernel ($WCET_{kernel}$) can be expressed as:

$$\begin{aligned} WCET_{kernel} = 2 \times max\{WCET_0(3), WCET_1(3), \\ WCET_2(3), WCET_3(3)\} \end{aligned} \tag{6.3}$$

  were 2 is the number of iterations executed during the kernel phase.

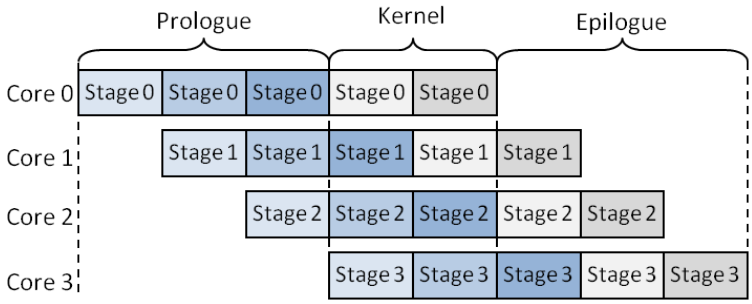


Figure 6.4: The prologue, kernel and epilogue phase of a 4-stage pipelined application that runs five times, each represented with a different color (light-blue, blue, dark-blue, light-grey and grey)

Therefore, the overall WCET of a pipelined application can be expressed as:

$$WCET_{pipe} = WCET_{pro} + WCET_{epi} + WCET_{kernel} \tag{6.4}$$

6.4.2 Considering The Cache into the WCET Estimation

However, when introducing the bankization technique in the computation of the WCET of each stage, it is required to consider the state of the cache at the end of the previous stage. That is, because each stage consumes the bank used by the previous one, a WCET analysis chain is created, in which the analysis of each state is used to feed the analysis of the next stage. It is worth noting that the analysis of a given stage does not require to know the state of the cache in all previous stages but just the previous one. Hence, we can define $WCET_n(m, cache_{n-1})$ as the WCET estimation of stage n, being n any stage except stage 0, running with m simultaneous stages and considering the cache state at the end of stage $n-1$. In case of stage 0, the WCET remains as $WCET_0(0)$, because it starts the computation of the streaming with the cache empty.

Therefore, the computation of the WCET estimation in every pipeline phase results in:

- The prologue phase:

$$WCET_{pro\_b} = WCET_0(0) + max\{WCET_0(1), WCET_1(1,0)\} + max\{WCET_0(2), WCET_1(2,0), WCET_2(2,1)\} \quad (6.5)$$

- The epilogue phase:

$$WCET_{epi\_b} = WCET_3(0,2) + max\{WCET_2(1,1), WCET_3(1,2)\} + max\{WCET_1(2,0), WCET_2(2,1), WCET_3(2,2)\} \quad (6.6)$$

- The kernel phase:

$$WCET_{kernel\_b} = max\{WCET_0(3), WCET_1(3,0), WCET_2(3,1), WCET_3(3,2)\} \quad (6.7)$$

Therefore, the overall WCET of a pipelined application when using bankization can be expressed as:

$$WCET_{pipe\_b} = WCET_{pro\_b} + WCET_{epi\_b} + num_{iterations} * WCET_{kernel\_b} \quad (6.8)$$

It is important to remark that the bankization interface ensures that the end of the stage n and the beginning of stage $n + 1$ are synchronized. By doing so, we can ensure that the WCET estimation of multiple stages running in parallel is equal to the highest WCET estimation among the different stages. Note that, in case of not using bankization, such synchronization among the stages is also required.

6.5 Results

This section presents the impact of the bankization technique on the WCET estimation of the two-stage pipelined parallel version of the two applications described in Section

2.4.1: LU Decomposition and Stereo Navigation. We consider the WCET estimation of each application under three different scenarios. For each scenario, we define n as the number of iterations of each application ($n = 5$ for LU Decomposition and $n = 4$ for stereo navigation).

1. *Single-thread version* ($WCET_{st}$): The two stages run one after the other into the same core. We assume that no other threads run concurrently, so the benchmark does not suffer from any inter-thread interferences from other applications. A private cache partition of 32 KB is assigned to this core. The WCET expression used is: $WCET_{st} = n \times [WCET_0(0) + WCET_1(0, 0)]$

2. *Pipelined parallel version without cache* ($WCET_{pipe}$, see equation 6.4): Each stage runs in a separate core. Again, no other threads run in parallel, so each stage may suffer only from inter-thread interferences from the other stage. The cache has been disconnected, so all memory requests go directly to the memory controller through the bus.

3. *Pipelined parallel version with bankization* ($WCET_{pipe\_b}$, see equation 6.8): Same as above but with cache and bankization technique enabled. So the cache partition used by stage 0 can be consumed by stage 1. A private cache partition of 16 KB has been assigned to each core.

6.5.1 LU Decomposition

Figure 6.5 shows the WCET estimation when running the two-stage LU decomposition application into two different scenarios: Using the bankization technique ($WCET_{pipe\_b}$) and not using a shared cache ($WCET_{pipe}$). Values are normalized to the WCET estimation of the single-threaded version of the LU decomposition application ($WCET_{st}$).

As expected, without a cache ($WCET_{pipe}$) the memory access interferences destroy completely the benefits brought by the pipelined model and increase the WCET estimation by up to 30% with respect to the single-threaded version. Instead, if using the bankization technique, the memory interferences are considerably reduced.

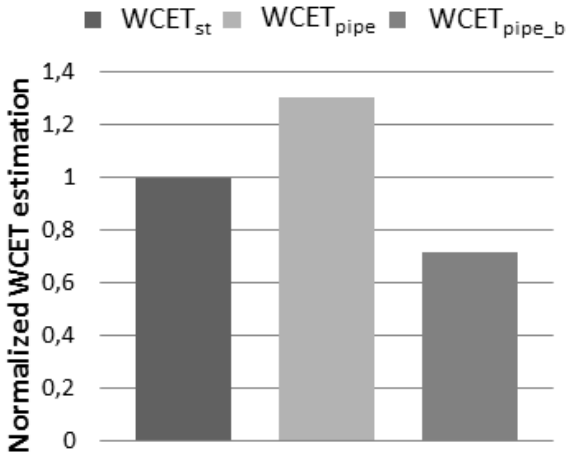


Figure 6.5: WCET estimation of the two-stage LU decomposition application using no shared cache ($WCET_{pipe}$) and bankization ($WCET_{pipe\_b}$). Values are normalized to the WCET estimation of the single-threaded version ($WCET_{st}$)

As a consequence the WCET estimation is reduced by 28% compared to the single-threaded version, taking full advantage of the parallel execution.

To better understand the benefits of bankization, Table 1 shows the WCET estimation increment of each stage of the LU decomposition (stage 0 and 1) during the three phases of the application (prologue, epilogue and kernel) if the shared cache is not used ($WCET_{pipe}$, see equations 6.1, 6.2 and 6.3) and if cache and bankization is enabled ($WCET_{pipe\_b}$, see equations 6.5, 6.6 and 6.7). Values are normalized to the WCET estimation of the corresponding stage in the single-threaded version: stage 0 for the prologue phase, stage 1 for the epilogue phase and the sum of both stages for kernel phase (note that in this case, stages do not suffer inter-thread interferences). Not using bankization, impacts negatively on the WCET estimation of each stage, increasing the WCET estimation in all phases, including the kernel phase whose WCET estimation is higher that the sum of the WCET estimation of the two stages in the single-threaded version. However, in case of using bankization, although the WCET estimation increases in the prologue and epilogue phase by 0.1% and 0.5% due to having a smaller cache (16KB) with respect to the single-threaded version (32KB).

Table 6.1: WCET estimation increments of the LU decomposition during the three phases using no cache ($WCET_{pipe}$) and bankization ($WCET_{pipe\_b}$). Values are normalized to the WCET estimation of the single-threaded version using cache

	$WCET_{pipe}$	$WCET_{pipe\_b}$
stage 0 (*prologue*)	24.06%	0.10%
stage 1 (*epilogue*)	30,07%	0,50%
max{stage 0, stage 1} (*kernel*)	22,73%	-33,43%

The benefit comes from the kernel phase that reduces the WCET estimation by 33% due to executing both stages in parallel.

6.5.2 Stereo Navigation

Figure 6.6 shows the WCET estimation of the two-stage stereo navigation application in the two different scenarios: Using the bankization technique ($WCET_{pipe\_b}$) and not using a shared cache ($WCET_{pipe}$)). Values are normalized to the WCET estimation of the single-threaded version of the stereo navigation application ($WCET_{st}$).

Similarly to the LU decomposition case study, the interferences introduced by main memory when not using a cache ($WCET_{pipe}$) destroy completely all the benefits brought by the pipelined parallel model and increases the WCET estimation by up to 48% with respect to the single-threaded version. Instead, when using the bankization technique, the memory interferences are considerably reduced, taking full advantage of the parallel execution and so allowing the WCET estimation to be reduced by 15% compared to the single-threaded version.

6.6 Summary

This chapter proposes a software/hardware cache partitioning approach that exploits the benefits of the software pipelined parallel programming model to effectively reduce inter-thread interferences when accessing the main memory, and so the WCET estimation with respect to the single-threaded programming model. Our approach

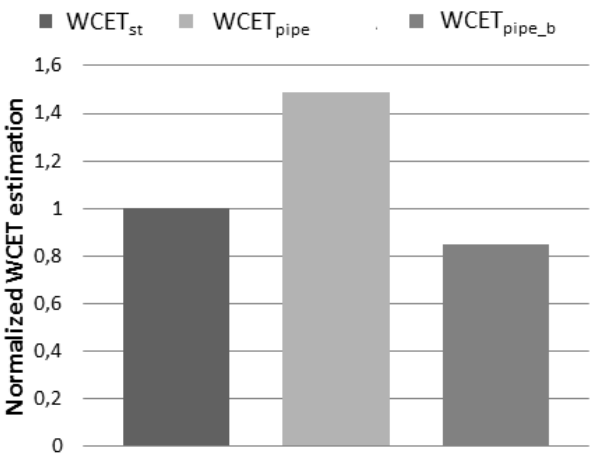


Figure 6.6: WCET estimation of the two-stage stereo navigation application using no shared cache ($WCET_{pipe}$) and bankization ($WCET_{pipe\_b}$). Values are normalized to the WCET estimation of the single-threaded version ($WCET_{st}$)

extends the bankization mechanism proposed in Chapter 3: a dynamically cache partitioning technique that allows to assign at run-time a private set of cache banks to a thread that no other can use, by providing an interface to guarantee the correct functional behavior of bankization as well as to provide the time predictability required by hard real-time applications. The programming model splits the heap memory region into two isolated regions: the *shared heap region* and the *private heap region*, such that accesses to the shared heap do not collide with private heap accesses. The used system-level software provides a set of synchronization primitives that ensures that a bank used by one thread is not remapped until this thread has finished. Hence, we can compute the WCET estimation of a software-pipelined parallel application.

In this Chapter we describe a first attempt towards the execution of hard real-time parallel applications; the work done in this chapter represents the foundations for the research that will be carried out in an European FP7 Project called parMERASA that has been accepted recently.

CHAPTER 7

WCET On-line Monitoring in an Automotive Environment

This chapter describes a proposal to verify the timing correctness of HRTs without requiring any modification in the processor: we design a hardware unit which is interfaced with the processor and integrated into a functional-safety aware methodology. This unit monitors the execution time of a block of instructions and it detects if it exceeds the WCET.

7.1 Introduction

Systems used in modern cars, are a typical example of hard real-time systems, in fact they are in charge of controlling the functionality and the safety of airbags, brakes, chassis control, engine control and in the future they will play a crucial role in x-by-wire cars.

It is commonly the case, in automotive systems, that different subsuppliers provide the different IPs of a System-On-Chip. In such scenario it is not possible to modify an IP designed by another supplier. Hence, we propose a timing and functional-safety aware methodology to combine software tools and a special hardware unit to support

time correctness at system level. We design a hardware unit which is interfaced with the IP of the core and integrated into the functional-safety aware methodology. This unit monitors the execution time of a block of instructions and it detects if it exceeds the WCET. In other words it monitors any WCET violation due to timing faults. Concretely, we show how to handle timing faults on a real industrial automotive platform. We implemented our complete flow and ran some experiments to check its effectiveness simulating our hardware platform at RTL.

This chapter also describes why timing dependability issues are of primary importance in automotive systems: we provide a formalization of the problem by defining timing faults and how they are considered by current certification standards. We indeed propose the fundamental notions behind timing correctness to be integrated in an automotive standard.

7.2 Background on Timing Issues

Automotive systems must satisfy ISO 26262 [5] functional safety norm, for example in terms of how to detect, correct or tolerate errors before that they cause a failure leading to a potential critical hazard, or in terms of how to guarantee the coexistence of different functions having different safety integrity levels without any mutual interference. Potential failures due to timing issues are also subject of several requirements of ISO 26262. At the same way, the increasing complexity of automotive systems is the main driver of AUTOSAR [3], the open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. Timing correctness is one of the topics in AUTOSAR as well. However, both ISO 26262 and AUTOSAR are describing timing issues without the level of detail that such important and emerging problem would require. For example, ISO 26262 does not include any specific guideline about how to prevent, detect or tolerate timing faults.

In the automotive domain there are three main standards that designers use to follow when designing their systems: IEC 61508 [4], ISO 26262 [5] and AUTOSAR [3].

The first one is related to the development of safety-related electronic systems, while the last one describes an open and standardized automotive software architecture. IEC 61508 was recently replaced by ISO 26262 to comply with needs specific to the application sector of Electrical and/or Electronic (E/E) systems within road vehicles. Among the three standards the one with major focus on timing issues is AUTOSAR.

7.2.1 Definition of a Timing Fault

As pointed out in the AUTOSAR [3] standard, a timing fault, at system level, refers to a process or service that is not delivered or completed within the specified time interval. The execution time of a task is not only restricted by the computing power of a processing unit but also depends on interferences generated by other tasks. The reasons for such interferences can be twofold:

1. Inter-thread interferences [70,71] generated accessing the same hardware shared resource if more than one task runs at the same time on a processor (valid only for multicore or multiprocessor systems).

2. Interrupt delay: whenever interrupts triggered by external devices (e.g. hardware watchdogs, peripherals) preempt the current task, this may lead to an unforeseeable delay of execution.

If, due to one of the reasons listed above, the execution time of one or more tasks in the system takes longer than what considered when defining the global scheduling of the system, a *timing fault*[1] is generated. The system, then, can react in different ways: task retry, function retry, function priority re-ordering, function replacing, microcontroller reset and microcontroller reconfiguration. The timing faults, as specified in the ISO 26262 for other types of faults, can be caused by:

1. **Systematic faults**: on the one hand, they are originated, if the designer underallocates the time budget for a task, i.e. considering an unsafe WCET that is

[1]During system integration designers consider the Worst Case Execution Time (WCET) of a task: the maximum possible execution time along all the different paths of the program.

smaller than the Maximum Observed Execution Time (MOET). On the other hand, they occur, if the designer changes the tasks that are scheduled together, so that different interference scenarios accessing shared resources occur [70, 71]. This generates timing faults, due to a longer execution time, with a consequent WCET violation. This kind of faults can be avoided by using a strong verification flow with the aid of tools that help the designer in the timing characterization of the tasks.

2. **Hardware faults**: random faults on hardware resources, e.g. the memory controller, that do not alter the data but delay the execution of the tasks can originate timing faults. These random faults cause, for instance, a wrong scheduling sequence in the memory controller of the requests to the memory. Such faults cannot be eliminated, but they can be reduced or limited if they are considered in the WCET analysis. Due to the technology scaling the number of hardware faults is increasing and such faults are becoming one of the primary issues for chip designers. However most of the works in research have been focusing on solutions that guarantee the functional correctness of a circuit (even in the presence of faults) but they do not deal with timing correctness that, in critical systems, is as important as functional one.

Timing faults, if not detected, can or cannot lead to the violation of the safety goal. The safety goal from a timing perspective is the WCET. Safe faults are the ones that do not involve a violation of the WCET, i.e. even in presence of the timing fault the execution of a task takes shorter than its WCET. If the timing fault leads to an execution time longer than the WCET then it violates the safety goal. In Figure 7.1 we define a flow to classify the faults.

7.2.2 Timing Fault Models

Regarding the timing correctness of an automotive critical task, examples of *timing fault models* are:

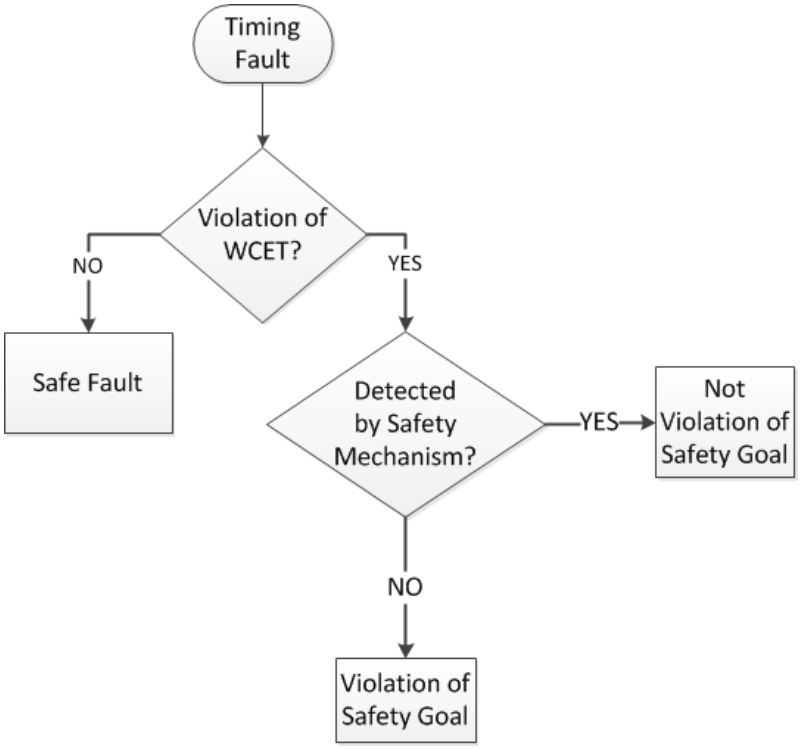


Figure 7.1: Flow to classify timing faults

1. Interrupt handling: external interrupts preempting a task in execution could lead to timing faults.

2. Caches: since the cache is a memory that does not have a constant access time (it depends on the current state of the memory), different states of the cache could lead to different execution times and so to different timing faults.

3. DMA/Memory controller: a problem in the memory controller, e.g. in the arbitration scheme, that could involve different memory access patterns, could lead to timing faults. Moreover, most memory controllers, especially the ones for FLASH memories, contain buffers and state machines that could cause timing faults.

4. Inter-thread interferences on shared resources: inter-thread interferences accessing shared resources [70, 71] could cause a thread to execute longer than what expected and consequently could cause timing faults.

5. Faults (systematic or random) of the CPU that could cause timing faults include:

   (a) Instruction Latency: if due to a fault, the latency of instructions changes, the total execution time of a task could suffer a delay, and so it could result in a timing fault.

   (b) High performance features, e.g. out of order execution, branch prediction, etc, use complex logic and buffers that, in case of a fault, could cause a delay in the execution (e.g. a different instruction issue sequence) with consequent timing faults.

6. Common-cause faults (either systematic or random) could originate timing faults and depending on the design could or not be detected by the safety mechanism: for e.g. if a fault is present in the clock tree, it can be detected if the safety mechanism and the CPU use different clock trees; otherwise not. Similarly a temperature increase could slow down the CPU and would be detected. An error in the configuration of the registers of the safety mechanism could also lead to a failure.

7.3 Our Solution: the TaCMU

YOGITECH's *faultRobust* technology provides a set of IPs [58] and methodologies [57] for the analysis, detection and correction of faults affecting the different parts of the electronic equipment or SoC. With respect to the faultRobust IPs (fRIP), each of them protects a particular component such as CPU, memory system and peripherals. Among the fRIPs, fRCPU is a scalable and flexible family of IPs for protection of CPU sub-system. fRCPU is not a replication of the CPU and it is architecturally and functionally diverse from the CPU. It is smaller than the CPU since it covers only what it is really relevant in the CPU sub-system to reach the highest safety integrity level (SIL3 according IEC 61508, ASIL D according to ISO 26262). Its main functions are fault detection and fault diagnosis. An example of fRCPU is fRCPU_armcm3, i.e. the specific fRCPU to be used in conjunction with an ARM Cortex-M3 processor [40]. fRCPU_armcm3 is designed to detect hardware random faults that could affect the ARM Cortex-M3 processor during its normal operation, such as permanent and transient faults. It also offers additional functions to support the detection of software systematic faults. In this chapter we propose a *timing and functional-safety aware* methodology to combine software tools and a special hardware unit integrated into the fRCPU_armcm3 to verify the time-correctness of an application. The special hardware unit is called *Timing-aware Coverage Monitor Unit* (TaCMU), it checks that the execution time of every *Block of Instructions* (BI) in the application is shorter than the corresponding WCET. A BI can be a basic block, a sequence of instructions, a function, or a set of functions; the level at which checks are performed depends on the granularity level fixed by the software developer. The timing and functional safety aware methodology we propose is composed by the following main steps:

1. The software developers write the source code of the program.
2. The source code is then automatically instrumented by the WCET analysis tool, that inserts, according to the granularity level set by the programmer, the instructions used to build the trace.
3. After that, the source code is compiled.

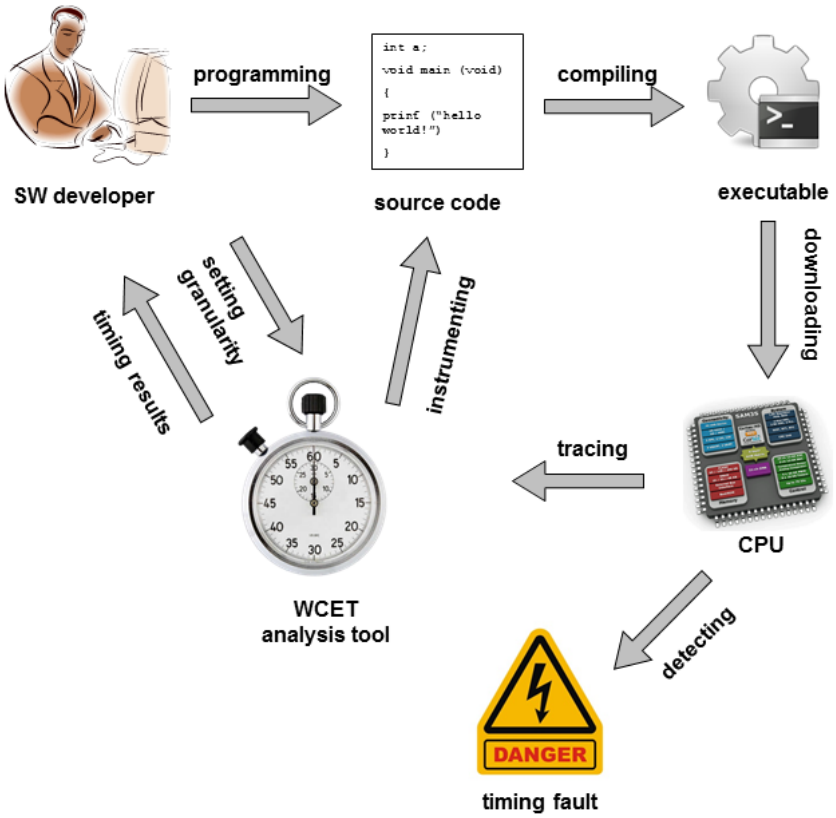


Figure 7.2: Diagram of the Timing and Functional-Safety aware methodology

4. The executable is downloaded on the CPU.

5. Then it is executed disabling the TaCMU (this is necessary to prevent any error for timing failures while performing the analysis) and an execution trace is captured.

6. The trace is processed by the WCET analysis tool, which provides the user with the results of the WCET and the MOET for each block of instructions, according to the granularity level set by the designer.

7. The user can decide how to set the timing constraints for each block selecting the MOET, or the WCET or applying a safety factor to the MOET.

8. The value selected for each block is then inserted into the source code.

9. The program is compiled again.

10. When it is executed, with TaCMU enabled, the TaCMU checks for every block of instructions that the execution time between the first instruction and the last one of the block is smaller or equal than the value set in the source code as WCET. If this is not the case an error is generated. The overall scheme of our flow is shown in Figure 7.2.

The advantages of our proposal for the designers are twofold:

- On the one side, the designer can benefit from a framework that: 1) can automatically insert/control watchdogs-like timers, 2) provides an integrated environment to analyze the timing behavior of an application, 3) allows to set the *watchdog-like timers* according to application/user requirements.

- On the other hand, the program flow monitoring assures that the program execution has completed the major parts of the program, and that it has completed them in the correct order.

The only disadvantage for the designer, that we identified, is the following: the source code instrumentation and annotation involve an overhead in terms of code size.

However this overhead can be easily controlled by the final users by leveraging the granularity of the timing annotations. For example, if the user instruments the source code to check the WCET at basic block level it has a consequent higher impact in terms of code size with respect to an annotation at function level.

7.3.1 Hardware Implementation

The overall hardware structure of TaCMU includes:

- A Stack structure to support timing monitoring of nested blocks of instructions.
- A tick counter to have an internal reference of the current time.
- A mechanism to save the stack structure in case of task context switch.
- Support to read the timing annotations set in the task source code.
- A mechanism to raise an exception in case of a timing fault.
- Comparators to check the violation of the WCET.

7.3.2 Software Environment

The software platform to support TaCMU includes:

- The integration of the WCET analysis tool inside the timing and functional safety aware methodology.
- An extension of the fault injector already included in the faultRobust faultInjector (fRFI, see [57]) to introduce timing faults and verify the safety mechanism.
- An instrumentation tool able to insert instructions to specify the WCET of the block of instructions.

7.3.3 CPU-Independence

An important property of TaCMU is that it is CPU-independent and it does not require any modification to be used with a different CPU than the ARM Cortex-M3. The TaCMU is coupled together with the fRCPU and it exchanges a small amount of data with it. On the other side, the TaCMU takes benefit of the tightly-coupled interface between the CPU and fRCPU described in [40]. The information it requires are the timing footprints of the running task (i.e. the WCET of different block of instructions that the designer want to monitor). In addition to that, it requires triggering the check points when they are reached during the execution. TaCMU can be easily adapted to any CPU (e.g. ARM A9) as long as the fRCPU for that CPU exists: there is no dependency with the configuration of the CPU and with the features it contains, and it only requires few special registers that need to be configured by software.

7.3.4 Safety Considerations about TaCMU

As for any other safety mechanism in the CPU, the ISO 26262 norm requires to analyze the safety aspects of the TaCMU itself, since it can also suffer from random hardware faults that, combined with another independent hardware or systematic fault in the mission function, could lead to an undetected timing fault (the so-called *latent fault*). In case of a fault in the TaCMU, there are two scenarios that can occur:

- A timing fault is detected even if it was not present.
- A timing fault that is present it is not detected.

To avoid the second scenario (and prevent the degradation of function availability as a consequence of the first scenario), it is necessary to provide TaCMU with hardware and software safety mechanisms that guarantee coverage (greater or equal than 90% for ASIL D safety goals) with respect to those latent faults, for example:

- A parity bit in all the stack memory entries and for each internal register.
- A software test or hardware testing unit to check the correctness of the comparators.

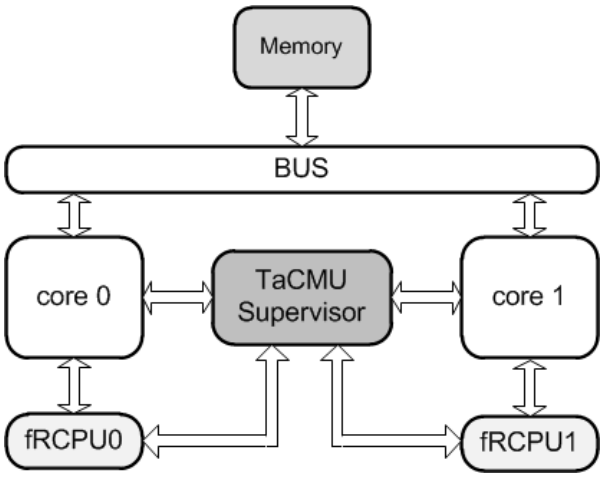


Figure 7.3: Multi-core Architecture with TaCMU Supervisor

- It is moreover required to use a different clock tree with respect to the one of the processor core to avoid common-cause failures on the clock tree.

Our TaCMU is able to detect the timing faults that are originated by random hardware faults, while the software tools used in the time dependable flow help the designers to avoid timing faults caused by systematic faults. Once the TaCMU detects a timing fault, due to a WCET violation of a block of instructions it notifies an error to the OS specifying the identifier of the block of instructions that violated the timing constraint.

7.3.5 TaCMU in a Multi-core Processor

The idea is still valid for multi-core processors: each core will have its own TaCMU. Moreover it is going to be even more useful than in single core processors because of inter-core interferences. Timing faults, in a multi-core scenario, are in fact more likely to happen due to inter-core interferences accessing shared resources.

Such inter-thread interferences [70] [71] have the consequence of delaying the other tasks that are executed on the other cores. The amount of such delay strictly depends on the hardware architecture and the tasks that are running simultaneously. We propose, in a multi-core environment, to have each core equipped with its own

fRCPU (that includes the TaCMU) and a global *TaCMU Supervisor* that communicates with all the fRCPUs. The TaCMU supervisor has an overall view of what it is running and what are the timing issues in each core. The TaCMUs Supervisor can implement a *failures control* strategy, and it proceeds to one of the following operations in case timing failures occur: Instruction Block retry, Task retry, Function retry, Function priority re-ordering, Function replacing, Core reset and MCU reset.

Moreover the TaCMU supervisor can change the scheduling in each core, if too many inter-thread interferences are occurring, with the consequent side effect of originating timing faults. An example of multi-core architecture equipped with the TaCMUs Supervisor is shown in Figure 7.3.

7.4 Results

To run our experiments, we implemented a Verilog synthesizable version of our TaCMU unit, and we integrated it into the internal YOGITECH platform composed of an fRCPU_armcm3 [40] connected with an ARM Cortex-M3 processor [2] described in Section 2.2.2.

The software platform, we used, includes:

- A simple emulation of a WCET measurement-based analysis tool that provides the WCET estimations analyzing the trace of the RTL simulation.
- An additional instrumentation tool, we developed, to specify which blocks of instructions/functions need to have the timing monitored.
- A prototype of a fault injector to inject timing fault by enlarging the execution time of a task by adding a random number of instructions.

To perform the experiments, we executed the complete flow (described in Figure 7.3) on some of the EEMBC automotive benchmarks (see Chapter 2), in particular on *aifirf01*, *a2time01*, *idctrn01*. To check the correctness of our timing flow, we estimated the WCET using the emulation of a WCET measurement based tool and instrumenting the source code at basic block level with the TaCMU timing monitoring

feature at function level. We injected timing faults by modifying the source code of the benchmarks to introduce timing faults by executing an additional random number of instructions to enlarge the execution time in the function under analysis. Such number of extra-instructions is parameterizable and it can be set by the designer.

For each benchmark we estimated the WCET; such WCET values are annotated into the source code in order to be read by TaCMU.

We executed each task 300 times introducing a random number of timing faults. In 100 of those simulations the faults did not lead to the violation of the safety goal, i.e. they were safe faults; this test is necessary to check that TaCMU does not detect false positive (faults that do not lead to the safety goal violation). The other 200 cases were simulations with faults that lead to an execution time that violates the WCET; hence the TaCMU must detect those cases. The results shown in the following table indicate the number of faults that were undetected and the ones that were detected in case of dangerous faults injected, and the false positive and undetected faults in case of unobservable/safe faults. The last column of the table shows the Diagnostic Coverage (DC).

Table 7.1: Results of the simulations with a selection of EEMBC benchmarks

Benchmark	Dangerous Faults		Safe Faults		DC
	Detected	Undetected	False Positive	Undetected	
a2time01	191	9	0	100	94.5%
aifirf01	200	0	0	100	100%
idctrn01	194	6	0	100	98%

The results (see Table 7.1) show that TaCMU does not detect any false positive fault and that the DC is always higher than 94.5%. The DC is not 100% in all cases, because the fault injector does not have the accuracy to increase the execution time of the task by one cycle. For that reason there could be faults that are safe (for few cycles) but they are considered dangerous. As part of the future work we plan to improve the accuracy of the fault injector to solve such problem.

Table 7.2: Results of the instrumentation of EEMBC benchmarks

Benchmark	Overhead at Basic Block level (%)	Overhead at Function level (%)
a2time01	6.1%	< 1 %
aifir01	3.1%	< 1 %
idctrn01	3.6%	< 1 %

In addition to DC, we also analyzed our idea in terms of code size overhead. The overhead introduced by the timing and functional-safety aware methodology in terms of code size is strictly dependent on the granularity chosen by the designer and the characteristics of the source code itself. For the EEMBC benchmarks that we used to test our prototype we instrumented the source code at basic block level and at function level, the results of the source code overhead are shown in Table 7.2.

7.5 Related Work

One of the approaches used in industry nowadays involves the use of watchdog timers [3]. A watchdog timer is a device that helps to assure that the microcontroller is operating properly. A watchdog timer may be internal or external to the system. It is a mechanism that begins to count down once it has been initiated. The device needs to be toggled / refreshed by software within a certain period of time to prevent a microcontroller from resetting. Watchdog timers are useful for detecting failures such as timing delays, infinite loops, and hung interrupts. The watchdogs need to set by the programmer and they need to be manually used. An alternative to the use of watchdogs, it is the unit included into the RENESAS Electronics V850E2R-V3 CPU [65]. Such CPU is equipped with a timing supervision unit to realize timing protection that can prevent the user application that operates under management of the OS from inappropriate possession of the CPU time by performing strict time management of the user application. If a violation is detected as a result of supervising the CPU

status by each counter in accordance with the setting, an exception occurs. The main advantages of TaCMU with respect to the other solutions presented are the flexibility and the fact that is part of an automatic flow. TaCMU is more flexible than using a watchdog because it can check more than one block of instructions at the time, i.e. it supports functions nesting. Moreover the programmer does not have to set the counters manually because the flow is partially automatic. With respect to solution proposed by RENESAS Electronics the advantages of TaCMU are still flexibility and the automatic characteristic.

7.6 Summary

This chapter has shown that is recommendable to extend functional safety and software architecture standards to cover timing dependability, and to guarantee it with proper software and hardware solutions. In this chapter, we show an idea towards the support of timing correctness describing what could be achieved by extending fRCPU by means of a TaCMU. It is also shown the importance of a timing and functional-safety aware methodology to enable designers to avoid systematic failures that derive from a wrong timing analysis.

CHAPTER 8

Conclusions

8.1 Thesis Conclusions

In current and future hard real-time systems it is fundamental for the hardware and the software to provide high performance while maintaining the time composability property. Higher performance is necessary to satisfy the increasing complexity of the functions that are implemented into the system. More complex functions allow increasing safety, comfort, number and quality of services, and lower emissions as well as fuel demands for automotive, avionics and automation applications. Moreover higher performance is mandatory when moving towards Integrated Architectures because multiple functions from different subsystems are integrated into each hardware unit. Time composability property is fundamental to maintain the cost of timing verification low by ensuring Incremental Qualification.

Multi-core processors provide better performance-per-watt ratio compared to single-core architectures, maximizing the utilization of the resources while guaranteeing low cost and low power consumption. All these features make multi-core architectures a very attractive solution for hard real-time systems. Moreover, the core design is

maintained simple avoiding undesirable timing anomalies. Unfortunately, it is hard, or even impossible to perform trustworthy WCET analysis for HRTs running on such processors due to inter-task interferences accessing hardware shared resources, that make the execution time, and so the WCET of a task dependent on the other co-running tasks. This makes not possible to achieve time composability.

In this thesis we have accomplished major results towards the use of multi-core processors in hard real-time systems by proposing a new multi-core design that enables time composability while providing high performance. In our proposed multi-core architecture the WCET estimation of a HRT is independent of the workloads in which it runs into. To that end, the maximum delay a request from a HRT accessing a hardware shared resource can suffer due to inter-task interferences is bounded. That is, our multi-core processor guarantees that a request to a shared resource cannot be delayed longer than a given *Upper Bound Delay* (UBD), ensuring the time composability property.

We have focused on the most common shared resources in a multi-core processor: the shared bus, the shared cache and the shared memory controller that interfaces the cores to the off-chip memory system:

1. Regarding the on-chip shared bus we have computed an UBD based on bus properties such as the latency and the different arbitration policies, identifying the ones that provide a bounded and tight worst-case access delay. To that end, we have proposed an interference-aware bus arbiter composed of an Inter-Core Bus Arbiter (XCBA) that schedules among requests from different cores, and several Intra-Core Bus Arbiters (ICBAs), one per core, which schedules among requests from the same core.

2. Concerning the shared cache, we have studied the effect of storage and bank interferences on the UBD of a HRT. We have evaluated two different cache partitioning techniques - namely *columnization* and *bankization* - which assigns private cache partitions to the different tasks in order to avoid storage and bank interferences. In case of bankization this is achieved by a Bank Remapping Unit

(BRU), that we designed; in case of columnization the replacement algorithm is modified to limit replacement to the columns assigned to a given task.

3. Finally, regarding the off-chip memory system we have computed an UBD to evaluate the impact that different JEDEC-compliant DDRx SDRAM memory systems have on the WCET estimation. First, we have proposed an analytical model based on generic timing constraints to compare different SDRAM memory devices from a WCET point of view. Second, we have presented the effect of the memory controller arbitration on the UBD, enabling the computation of WCET estimations for any HRT. In addition to that, we have designed a Real-Time Capable Memory Controller (RTCMC) that includes two new features to deal with refresh operations, that are one of the main contributors to the variability in the low predictability of memory systems, and to reduce the impact of NHRTs over HRTs.

Moreover, we extend the bus and the memory controller with a hardware feature called *WCET Computation Mode* that allows computing a safe WCET estimation, which is independent of the workload and so accomplishing the time composability property. When a HRT is being time analyzed the processor is set in the WCET Computation mode and the task is run in isolation. In this execution mode, the processor artificially delays each HRT request by UBD cycles. As a result, the computed WCET estimation is a safe upper bound of the execution of the HRT as it considers the worst possible interferences with other tasks that run inside a workload. Hence, with our proposal, the WCET analysis of each HRT can be performed in isolation as done in single-core processors.

In this thesis we also have investigated an allocation algorithm to allow multi-cores to execute mixed-criticality level application workloads. To that end, we presented IA^3, an new off-line interference-aware allocation algorithm for multi-core processors. The IA^3 is based on two novel concepts: the WCET-matrix and the WCET-sensitivity. These two concepts allow IA^3 identifying those tasks with higher resource requirements, allowing to minimize the resources assigned to HRTs and hence enables the rest of the resources to be assigned to NHRTs, thus maximizing

the hardware utilization and taking full advantage of multi-core processors. To do so, IA^3 considers not just a single WCET estimation but a set of WCET-estimations generating a more efficient partitioning.

Moreover, in order to improve the performance of the HRTs, we have made a first analysis towards the possibility of executing parallel applications guaranteeing hard real-time requirements. To that end, this thesis proposes a software/hardware cache partitioning approach that exploits the benefits of the software pipelined parallel programming model to effectively reduce inter-thread interferences when accessing the main memory. The WCET is hence reduced with respect to the single-threaded version of the HRT. The programming model has been defined considering the effects of the software parallelization on the WCET of an application. In fact, if an application is parallelized without considering the effects on its WCET high average performances can be achieved while the WCET may increase a lot, removing all the benefits introduced by the parallelization.

Finally, this thesis also investigates a solution to verify the timing correctness of HRTs without requiring any modification in the core design. To that end, we have designed a hardware unit which is interfaced with the processor and integrated into a functional-safety aware methodology. This unit monitors the execution time of a block of instructions and it detects if it exceeds the WCET. Concretely, we show how to handle timing faults on an industrial automotive platform.

8.2 Future Work

The work done in this thesis opens several research lines targeting new challenges in hard real-time systems some of which are already pursued by other PhD students in the Universitat Politecnica de Catalunya.

In particular, this thesis is one of the first attempts showing that it is possible to design a time composable multi-core processor. Starting from the proposals described in this thesis it is possible to research on a many-core processor composed by 32, 64 or more cores. The design of such time composable many-core processor would al-

low to integrate all the functions of a car into few chips being extremely advantageous in terms of cost, weight and power consumption. Of course, the design of such processor involves several challenges that need to be addressed: more than one memory controller needs to be implemented into the processor, the interconnection network between the cores and the main memory should be designed to be time composable while providing high performance.

In such a many-core architecture it would be possible to fully exploit the thread level parallelism, it is then mandatory the definition of a parallel programming model that allows software developers to easily parallelize the applications achieving high performance while satisfying predictability and time composability properties. Some of these issues are going to be addressed by other PhD students of Universitat Politecnica de Catalunya who will be part of an European FP7 Project called parMERASA that has been recently accepted in the FP7-ICT-2011-1 call. The focus of such project is a many-core processor with hardware and software support for parallel hard real-time applications. In particular future work will be focused on applying the same approach presented in this thesis in a many-core processor (up to 64 cores) addressing complex network on chip, the memory hierarchy and predictable parallel programming models.

Other research lines, include the investigation of applying the proposals of this thesis to existing hardware architectures, like the Next Generation Multipurpose Microprocessor (NGMP) by Gaisler that the European Space Agency (ESA) plans to use for future space missions. In particular it is required to investigate the possibility of performing the analysis of the UBD and implementing the Worst Case Computation Mode to the AMBA bus developed by ARM Ltd. It is also going to be required the investigation of composable I/O devices, like the possibility of designing a real-time capable DMA controller, and a mechanism to handle interrupts in a safe and composable way. Interrupts may delay the execution of a HRT, resulting in a deadline miss. Two collaboration projects have been established with ESA to investigate some of the previously mentioned aspects. In particular those two projects will focus on the following shared resources: AMBA bus and DDR3 SDRAM memory devices.

Bibliography

[1] *ARC Study: Process Safety System Worldwide Outlook - Market Analysis and Forecast Through 2012*.

[2] *ARM Cortex-M3 Technical Reference Manual http://infocenter.arm.com/help/index.jsp*.

[3] *AUTOSAR Glossary V2.2.0 R4.0 Rev 1*.

[4] *IEC 61508-1 Ed. 2.0: Functional safety of electrical/electronic/programmable electronic safety-related systems*.

[5] *ISO/FDIS 26262-1:2010(E) Road vehicles – Functional safety*.

[6] *RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification*.

[7] *MERASA EU-FP7 Project: www.merasa.org*, 2007.

[8] *RapiTime: Worst-case execution time analysis. User Guide. Rapita Systems. Ltd.*, 2007.

[9] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. The MPsim Simulation Tool. Technical Report UPC-DAC-RR-CAP-2009-15, in UPC, 2009.

[10] B. Akesson. *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, feb 2010.

[11] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, USA, 2007. ACM.

[12] B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *Proc. DSD*, Aug. 2009.

[13] J. H. Anderson, J. M. Calndrino, and D. C. UmaMaheswari. Real-time scheduling on multicore platforms. In *RTAS*, pages 179–190, 2006.

[14] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSID*, USA, 2008.

[15] ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.

[16] J. M. Banús, A. Arenas, and J. Labarta. An efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2*, PDPTA '02, pages 809–815. CSREA Press, 2002.

[17] J. M. Banús, A. Arenas, and J. Labarta. Dual priority algorithm to schedule real-time tasks in a shared memory multiprocessor. In *IPDPS*, page 112, 2003.

[18] S. Becz, A. Pinto, L. E. Zeidner, R. Khire, A. Banaszuk, and H. M. Reeve. Design system for managing complexity in aerospace systems. In *2010 AIAA ATIO/ISSMO Conference*, September 2010.

[19] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.

[20] B. Bhat and F. Mueller. Making dram refresh predictable. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 145–154, Washington, DC, USA, 2010. IEEE Computer Society.

[21] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. *RTCSA*, Aug. 2008.

[22] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *ISCA*, US, 1996.

[23] H. Cassé and P. Sainrat. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software*, Toulouse, France, January 2006.

[24] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramírez, and M. Valero. Feasibility of qos for smt. In *Euro-Par*, pages 535–540, 2004.

[25] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramírez, and M. Valero. Predictable performance in smt processors: Synergy between the os and smts. *IEEE Trans. Computers*, 55(7):785–799, 2006.

[26] F. J. Cazorla, A. Ramírez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. Qos for high-performance smt processors in embedded systems. *IEEE Micro*, 24(4):24–31, 2004.

[27] R. N. Charette. This car runs on code. *IEEE Spectrum*, February, 2009.

[28] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *DAC*, Los Angeles, CA, USA, 2000.

[29] P. Clarke. Automotive chip content growing fast, says gartner. *EETimes*, 2010.

[30] T. H. Cormen, C. E. Leiserson, R. L. R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[31] S. Davari and S. Dhall. An on line algorithm for real-time allocation. In *ICSS*, pages 133–141, 1986.

[32] R. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *RTSS*, Washington, USA, 2009.

[33] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 2010.

[34] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (to appear) available from http://www-users.cs.york.ac.uk/ robdavis/*, 2010.

[35] S. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operation Research*, pages 127–140, 1978.

[36] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *CASES*, USA, 2005. ACM.

[37] A. El-Haj-Mahmoud and E. Rotenberg. Safely exploiting multithreaded processors to tolerate memory latency in real-time systems. In *CASES*, USA, 2004. ACM.

[38] P. Emberson and I. Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *RTSS*, pages 270–279, 2008.

[39] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[40] T. Halfill. Fault tolerance for Cortex-M3. *Microprocessor Report*, 2008.

[41] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 2009.

[42] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. *AbsInt White paper*, 2009.

[43] P. Holman and J. H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, 2005.

[44] I. Hur and C. Lin. Adaptive history-based memory schedulers for modern processors. *IEEE Micro*, 2006.

[45] Infineon. *Tricore 1. 32-bit Unified Processor Core v1.3*, October 2005.

[46] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[47] R. Jain, C. J. Hughes, and S. V. Adve. Soft real- time scheduling on simultaneous multithreaded processors. In *RTSS*, page 134, 2002.

[48] K. jay Lin, S. Natarajan, and J. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *RTSS*, 1987.

[49] JEDEC Solid State Techn. Assoc. *JEDEC DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008.

[50] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *RTSS*, pages 129–139, 1991.

[51] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] S. Kato and N. Yamasaki. Extended u-link scheduling to increase the execution efficiency for smt real-time systems. *Real-Time Computing Systems and Applications, International Workshop on*, 0:373–377, 2006.

[53] I. A. Khatib, F. Poletti, D. Bertozzi, L. Benini, M. Bechara, H. Khalifeh, A. Jantsch, and R. Nabiev. A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: architectural design space exploration. In *DAC*, pages 125–130, USA, 2006. ACM.

[54] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *CASES*, 2008.

[55] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442, 1995.

[56] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.

[57] R. Mariani and G. Boschi. A systematic approach for failure modes and effects analysis of system-on-chips. In *Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 187–188, Washington, DC, USA, 2007. IEEE Computer Society.

[58] R. Mariani, P. Fuhrmann, and B. Vittorelli. Fault-robust microcontrollers for automotive applications. In *Proc. 12th IEEE Int. On-Line Testing Symp. IOLTS 2006*, 2006.

[59] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04)*, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[60] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Proceedings of the 24th international conference on Architecture of computing systems*, ARCS'11, pages 122–134, Berlin, Heidelberg, 2011. Springer-Verlag.

[61] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th MEDEA workshop*, MEDEA '08, pages 38–45, New York, NY, USA, 2008. ACM.

[62] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer. How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT. In *23rd International Conference on Architecture of Computing Systems (ARCS 2010), Hannover, Germany*, Feb. 2010.

[63] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads. In *IEEE International Conference on Computer Design 2008 (ICCD 08)*, pages 371–376, Lake Tahoe, CA, USA, Oct. 2008.

[64] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM.

[65] NEC. *V850E2R-V3 32-bit Microprocessor Core Architecture, Preliminary UserŠs Manual*, 2009.

[66] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO*, USA, 2006.

[67] R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[68] Y. Oh and S. H. Son. Tight performance bounds of heuristics for a real-time scheduling problem. Technical report, Charlottesville, VA, USA, 1993.

[69] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Syst.*, 9(3):207–239, 1995.

[70] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[71] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letter*, 2009.

[72] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time COTS based systems. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.

[73] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.

[74] C. Pitter. Time predictable cpu and dma shared memory access. In *In International Conference on Field-Programmable Logic and its Applications (FPL 2007*, 2007.

[75] C. Pitter and M. Schoeberl. Towards a java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 144–151, New York, NY, USA, 2007. ACM.

[76] C. Pitter and M. Schoeberl. A real-time java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10:9:1–9:34, August 2010.

[77] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[78] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison. Technical report, IRISA, Paris, France, October 2006.

[79] S. Rixner. Memory controller optimizations for web servers. In *MICRO*, 2004.

[80] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA*, 2000.

[81] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu. WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 90–100, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[82] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu. Wcet analysis of a parallel 3d multigrid solver executed on the merasa multi-core. In *Workshop on Worst-Case Execution-Time Analysis in conjunction with ECRTS*, 2010.

[83] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[84] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, pages 41–48, 2005.

[85] Tasking. *Tricore v2.2 C Compiler, Assembler, Linker Reference Manual*, 2005.

[86] L. Thiele and R. Wilhelm. Design for time-predictability. In *Design of Systems with Predictable Behaviour*, 2004.

[87] S. Uhrig, S. Maier, and T. Ungerer. Toward a processor core for real-time capable autonomic systems. In *Proc. Fifth IEEE International Symposium on Signal Processing and Information Technology*, pages 19–22, 2005.

[88] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS*, New York, NY, USA, 1998.

[89] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.

[90] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[91] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. In *Proceedings of the 13th IEEE ISORC 2010*, pages 193–201. IEEE Computer Society, May 2010.

[92] W. Wolf. *High-Performance Embedded Computing*. Morgan Kaufmann, 2007.

APPENDIX A

Publications

A.1 Conferences

- Marco Paolieri, and Riccardo Mariani. *Towards Functional-Safe Timing-Dependable Real-Time Architectures*. In Proceedings of the 17th IEEE International On-Line Testing Symposium (IOLTS'11). Athens, Greece, July 13-15, 2011.

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. *IA^3: An Interference Aware Allocation Algorithm for Multicore Hard Real-Time Systems* . In Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11). Chicago, IL, USA, April 12-14, 2011.

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Julian Wolf, Theo Ungerer, Sascha Uhrig, and Zlatko Petrov. *A Software-Pipelined Approach to Multicore Execution of Timing Predictable Multi-Threaded Hard Real-Time Tasks* . In Proceedings of the 14th IEEE International Symposium on Object/

Component/ Service-oriented Real-time Distributed Computing (ISORC '11). Newport Beach, CA, USA, March 28-31, 2011

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. *Hardware support for WCET analysis of hard real-time multicore systems*. In Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA '09). Austin, TX, USA, June 20-24, 2009

A.2 Journals

- Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Hugues Casse, Christine Rochange, Eduardo Quiñones, Sascha Uhrig, Mike Gerdes, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, Jorg Mische, Marco Paolieri and Julian Wolf. *MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability*. In the IEEE Micro 2010, Special Issue on European Multicore Processing Projects, Vol. 30, No. 5, October 2010

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs*. In the IEEE Embedded Systems Letter (ESL), Vol. 1, No. 4. December 2009.

A.3 Workshops

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. *Effcient Execution of Mixed Application Workloads in a Hard Real-Time Multicore System*. In Reconciling Predictability with Performance (RePP) Workshop, within the Embedded System Week (ESWeek '09), October 11-16, 2009.

A.4 Posters

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. *A Multi-core Architecture for Safety Critical Real-time Embedded Systems*. In PhD Forum at Design, Automation & Test in Europe (DATE), Grenoble, France. March, 2011.

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. *Multicore Architecture for Critical Real-Time Embedded Systems*. In Hipeac Innovation Event, Edinburgh, UK. May, 2010. [Best Poster Award]

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. *Multicore Architecture for Hard Real-Time Systems*. In Hipeac ACACES Summer School, Terrasa, Spain. July, 2009.

A.5 Submitted Papers

- Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. *Timing Effects of the Memory System in Real-Time Multicore Architectures: Issues and Solutions*. Submitted to ACM Transactions on Embedded Computing Systems (TECS).

- Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Francisco J. Cazorla, Mateo Valero, and Theo Ungerer. *A Hard Real-Time Capable Multi-Core SMT Processor*. Submitted to ACM Transactions on Embedded Computing Systems (TECS).

APPENDIX B

Glossary

AC Address Computation Stage

ALU Arithmetic Logic Unit

AUTOSAR Automotive Open System Architecture

AXI Advanced eXtensible Interface

BI Block of Instructions

BRU Bank Remapping Unit

CA Collision Avoidance

DE Decode Stage

DDR Double-Data-Rate

DRAM Dynamic RAM

EDF Earliest Deadline First

ESA European Space Agency

FE Fetch Stage

FPU Floating Point Unit

HRT Hard Real-time Task

IA3 Interference-Aware Allocation Algorithm

ICBA Intra-Core Bus Arbiter

ICU Interference Control Unit

IMA Integrated Modular Avionics

IP Intellectual Property

LID Longest Issue Delay

MCU Microcontroller Unit

MEM Memory Stage

MOET Maximum Observed Execution Time

NGMP Next Generation Multipurpose Microprocessor

NHRT Non Hard Real-time Task

OS Operating System

PD Pre-decode Stage

RAM Random Access Memory

RET Request Execution Time

RR-D Data Register Read Stage

RR-A Address Register Read Stage

RTCMC Real-Time Capable Memory Controller

RTL Register Transfer Level

SDRAM Synchronous Dynamic RAM

SMT Simultaneous Multithreading

SoC System-on-Chip

SRAM Static RAM

TaCMU Timing-aware Coverage Monitor Unit

TDM Time-Division Multiplexing

TLP Thread-Level Parallelism

UBD Upper Bound Delay

UAV Unmanned Aerial Vehicle

WB Write Back Stage

WCET Worst-Case Execution Time

XCBA Inter-Core Bus Arbiter