UNIVERSITAT POLITECNICA DE CATALUNYA

Enhancing Timing Analysis for COTS Multicores for Safety-Related Industry: a Software Approach

by

Gabriel Alejandro Fernandez Diaz

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Informatics Faculty
Computer Architecture Department

September 2018

UNIVERSITAT POLITECNICA DE CATALUNYA

Enhancing Timing Analysis for COTS Multicores for Safety-Related Industry: a Software Approach

by

Gabriel Alejandro Fernandez Diaz

September 2018

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Informatics Faculty

Computer Architecture Department

Advisor: Francisco J. Cazorla
Barcelona Supercomputing Center and IIIA-CSIC

Co-Advisor: Jaume Abella
Barcelona Supercomputing Center

Declaration of Authorship

I, Gabriel Fernandez, declare that this thesis titled, 'Enhancing Timing Analysis for COTS Multicores for Safety-Related Industry: a Software Approach' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"I only know that I know nothing"

Socrates

Abstract

Artificial system interaction with the real environment is in general based on the deployment of properly coordinated sensors and actuators, establishing a "dynamic control-loop" between them. The time to close this control-loop characterizes the functionality and applicability to critical systems in response time. In the case of digital control, the performance of the processor is directly related to response time. In this line computational demands in many Critical Embedded System industries such as avionics, space, automotive and railway have experienced an unprecedented growth as a consequence of the need to cope with more sophisticated software functionalities. The use of high-performance hardware features in critical embedded systems, such as multicore architectures, to respond to those performance requirements, challenges the computation of tight worst case execution time (WCET) estimates. The source of this complexity comes from the interferences (contention) when accessing hardware resources shared across the different tasks running simultaneously. Several proposals advocate for hardware support to either eliminate or control inter-task conflicts on access to shared hardware resources (e.g. Time Division Multiple Access (TDMA) in buses, partitioning for caches), to simplify timing analysis via removing or controlling effect of contention. However, to the best of our knowledge, no current Commercial Off-The-Shelf (COTS) multicore processor provides complete isolation or full control of inter-task interference. As a consequence, the execution time of a software program may be inordinately affected by the load that its co-runners place on the hardware shared resources. This Thesis provides software methodologies to characterize and control the contention on COTS multicore processors so that they can be factored in measurement-based timing analysis. To that end, we make the following contributions. First, we perform a study of the vast state of the art on the topic and we propose a taxonomy to classify existing approaches with emphasis on their goals and assumptions. This helps better understanding the symbiosis and overlapping elements of the state-of-the-art works. Second, we propose a measurement-based methodology to derive the longest delay requests from a task can take accessing FIFO and round-robin arbitrated resources, which is fundamental to derive tasks' worst-case contention effects. Third, with the goal of deriving time composable WCET estimates, we introduce signatures and templates to abstract contention caused and incurred by tasks in a multicore. Fourth, we present a methodology to derive WCET estimates during early design stages, before tasks (software units) are integrated. And fifth, we report our experience with timing analysis on two COTS ARM-based multicores.

Acknowledgements

The research leading to this thesis has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080(SAFURE), the European Space Agency under Contract 789.2013 and NPI Contract 40001102880; and COST Action IC1202, Timing Analysis On Code-Level (TACLe)., and the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P.

I would like to express my gratitude to my advisors Francisco J. Cazorla and Jaume Abella for their support, guidance and help. I would also like to acknowledge to Eduardo Quinones, Javier Jalle, Luca Fossati, Guillem Bernat, Marco Zulianello, Christine Rochange, Tullio Vardanega and Sylvain Girbal for their contribution to this thesis. Also I want to thanks to all my colleagues in the Barcelona Supercomputing Center, especially in the CAOS group for all this past years shared together.

Contents

List of Figures

List of Tables

Abbreviations

AMBA	Advanced Microcontroller Bus Architecture
APKI	Access per Kilo Instruction
BSK	Bus Stressing Kernel
CABA	Cycle Accurate /Bit Accurate
CLS	Cache Lines
COTS	Commercial Off-The-Shelf
DL1	First Level Data Cache
EEMBC	Embedded Microprocessor Benchmark Consortium
FCFS	First-Come First-Served
FIFO	First-In First-Out
FR-FCFS	First-Ready First-Come First-Served
IL1	First Level Instruction Cache
IMA	Integrated Modular Avionics
IMA-SP	IMA for Space
IP	Intelectual Property
L2	Second Level Cache
LRU	Least Recently Used
MBTA	Measurement Base Time Analysis
MRU	Most Recently Used
MSK	Memory Stressing Kernel
NDA	Non Disclosure Agreement
NoC	Network-on-Chip
PMC	Program Monitoring Counter
RoRo	Round Robin
RSeK	Resource Sensitive Kernel

RSK	Resource Stressing Kernel
RUl	Resource Usage Template
RUs	Resource Usage Signature
scua	Software Component Under Analysis
sdki	Stack Distance per Kilo Instruction
SDV	Stack Distance Vector
SoC	System on Chip
STA	Static Time Analysis
SurApp	Surrogative Application
TDMA	Time Division Multiple Access
UBD	Upper Bound Delay
WCET	Worst-Case Execution Time

This Thesis is dedicated to Enrique Fernandez Garcia, my father who nurtured my thirst for knowledge answering every scientific question that a child can come with.

In memory of Aranzazu Diaz Urrestarazu, my mother who gave me the wits to follow this path and I miss so much.

Chapter 1

Introduction

During the last decades our society has witnessed a steady growth of all kind of computers. With this increase in the availability of computational power, which has reached the point where we can hold a gigantic amount of computation on the palm of our hands, the demand for these products has followed suit. While most of this growth may pass unnoticed by most of the common users, whom may picture only general purpose computers as the only commercial use of this technology, in fact, computing devices have been taking over several applications that traditionally were in the realm of mechanical or simpler electronics. Such applications have come to be labeled as Embedded Computation, being distinguishable by how focused to a very specific functionality they are. In other words, they correspond to computers embedded in vehicles, industrial equipment, etc. As an illustrative example, let us take the computers that control any modern car, where originally things regulated by mechanical or electrical appliances (e.g. fuel injection, ABS control, etc) now are done by a embedded computer with more efficiency. Far from reaching a stalling point, the development of better embedded computing systems has continued in line with the increasing demand of these kinds of devices, e.g. in the smartphone market. The continued increase of the performance available, far from deter or satiate the demand for these products, has kept growing reanimating even other fields now in high demand like Machine Learning. So now we can see High Performance processors with embedded applications becoming the new norm like using multicores CPUs and GPUs in cellphones.

Even thought the most widely known application of embedded systems may be entertainment and communication, they are used in domains where failures can have devastating consequences. As such, embedded systems in those domains are referred to as critical embedded systems, and can be broadly classified into several non exclusive categories depending on how critical they are. A embedded system may take care of a critical

mission whose failure may generate economical or social harm for the company. These systems may be classified as "mission critical". Aside, a different or the same system may take care of security functions, like cryptography, whose failure may involve security issues, so this system may be classified as "security critical". And last but not least, critical embedded system may be in charge of functionalities related to the safety of human beings, so that a failure may cause fatalities, material loss or environmental damage. Those systems can be classified as "safety critical".

The market for critical embedded systems covers a significant share of the overall embedded market [1] and it is in fact on the rise with further expectations of growth [2]. Evidence of the growth of critical embedded systems can be found in the increasing attention that chip manufacturers have drawn to this market, with specific products targeting them [3, 4].

1.1 Requirements of Critical Embedded Systems

Evidence of correct timing behavior. As for mainstream computing systems, the correctness of critical embedded systems depends on their ability to provide correct functional results. However, unlike regular embedded systems, critical embedded systems involve non-functional results that are as important as the functional ones. Non-functional metrics include, among others, timing (or guaranteed performance), security, and energy. In this Thesis we focus on timing behavior. The timing *Validation and Verification* of *Critical Embedded Systems* focuses on providing evidence that system functions will be performed timely. Timing Validation and Verification builds on two elements: First, timing analysis methods that estimate bounds to the WCET of tasks [5]; and second, task scheduling techniques that assess whether all application software implementing system functions are actually executed timely [6]. Further, in many cases the behavior of critical embedded systems is subject to legal directives. Critical embedded systems providers must show adherence to those directives, before they are allowed to deploy a critical embedded system device. Legal directives are "implemented" showing adherence to safety standards, e.g. ISO26262 [7] in automotive. Those standards aim at reducing the risk of malfunction so that it is as low as reasonably practicable, either in the functional or non-functional behavior of the critical embedded systems, so the system is considered safe enough. The level of evidence to be provided depends on the criticality of the system/subsystem under consideration.

Unprecedented performance requirements. Most new "smart" services in current and future critical embedded systems, e.g. autonomous driving in cars, are software-based. This makes software one of the central elements to increase critical embedded

systems' competitive edge [8]. Smart software services translate into software managing huge amounts of data (e.g. coming from camera and LIDAR in cars). Further, software is instrumental for decision making implementing complex artificial intelligence algorithms. As a result, software complexity and performance requirements raise to unprecedented levels. For instance, in the automotive domain, performance is expected to increase by 100x by 2024 with respect to applications performance requirements in 2016 [9].

Increased integration and use of Commercial Off-The-Shelf multicores. Several sources claim that, by the end of 2016, luxury cars embedded up to 150 Electronic Control Units, in response to the need for high computer performance. This trend towards high Electronic Control Units count faces the problem of limitations in the physical space available in a car and also the cost, weight, and low-reliability of physical components (the electronic control units and the wiring to connect them). To address this problem, critical embedded systems industry has begun the process to adopt multicore processors (multicores in the following) as their baseline computing solution. This situation extends across a variety of application domains, including automotive, avionics and space. In this line, despite the interest of chip providers in the growing critical embedded systems' market, they are driven by the mainstream market (e.g. mobile market). As a result, industrial developers of critical embedded systems have to turn to Commercial Off-The-Shelf (COTS) processors to abate the procurement costs and obtain the performance needed. However, mainstream COTS processors are equipped with several features that complicate timing validation and verification.

Integrated Architectures, Mixed criticality and multi-provider critical embedded systems. In the past, the design of critical embedded systems built on the concept of federated architectures [10], under which critical embedded system consisted of independent interconnected subsystems, each implementing one or few functionalities. This physical separation allows incremental certification by construction. However, as the number of functions to implement by software grows, it is infeasible to devote a dedicated hardware (e.g. Line Replaceable Unit) to run it. Industry has addressed this challenge by adopting integrated architectures, in which several applications, usually subject to different criticality characteristics, run concurrently on a single hardware platform. Examples of integrated architectures are Integrated Modular Avionics (IMA) [11] in avionics, Integrated Modular Avionics for space (IMA-SP) [12] for the space domain, and AUTOMotive Open System ARchitecture (AUTOSAR) [13] for the automotive domain.

In terms of software design, integrated approaches allow system integrators to subcontract the development of different software elements to various software providers. Software Providers are given a specification of the required functionality and time budgets

in which their application(s) must fit. Time budgets are defined according to a global schedule designed by the integrator to run all software functionalities timely. In general, a Software Provider carries out the implementation of its applications in increments, checking in every release applications' functional and non-functional behavior against their specifications.

1.2 Challenges in Timing Analysis

The trends shown in the previous section on the design and use of critical embedded systems have caused disruptive changes in current practice timing analysis.

Multicore Timing Analysis. In spite of the potential to improve performance, embracing multicores for real-time systems industry is challenging as they bring their own difficulties, especially for timing analysis. At a conceptual level, the intent of timing analysis is to provide, at low-enough cost, a WCET bound for programs running on a given processor, so that high guaranteed utilization of the computation resources can be assured. In this formulation, "high" utilization implies tight bounds (hence with as little pessimistic over-provisioning as possible) and "guaranteed" means truly upper bounding (hence with no risk of incautious under-provisioning). In particular, research on timing analysis for multicore processors is still in its infancy, particularly for COTS hardware. Ideally, the transition to multicore processors should allow industrial users to achieve higher levels of guaranteed utilization, together with attractive reduction in the performance-per-watt ratio, design complexity, and procurement costs. Unfortunately, however, the architecture of multicore processors poses hard challenges on (worst-case) timing analysis. The interference effects arising from contention on access to processor-level shared resources need far greater attention than in the single-core case, as much greater is the arbitration delay and state perturbation that resource sharing may cause. In consequence, the "padding" factor that needs to be captured in the computed bounds to compensate for the relevant effects is much greater. The difficulty with the timing analysis of software programs running on multicore processors is, thus, a serious impediment to their adoption in real-time systems industry. Static or Measurement-Based Timing Analysis (STA and MBTA respectively) [14] are affected by the disruption of complex hardware and in particular multicores.

- **Static Timing Analysis** relies on an accurate timing model of the hardware under test. Static timing analysis further creates a mathematical representation of the application, which is combined with the timing model to derive bounds to the applications' timing behavior in that hardware. Static timing analysis focuses on

the soundness and safeness of its application, which allows theoretically, meeting all safety standard requirements. However, the validity of the bounds depends on the correctness of the hardware timing models, which are difficult to develop and validate especially for complex hardware. This is compounded with the lack of timing information of the processor implementation [15]. Even when hardware manufacturers provide timing information (e.g. in the reference manuals), it can be inaccurate or outdated with respect to the deployed chip implementation. As illustrative example, the FreeScale e500mc core documentation comprises several revisions already with non-negligible changes across them [16]. In the case of multicores, this lack of information affects the impact of contention that tasks suffer in the access to shared hardware resources. All these difficulties have made that real-time industry and static timing analysis tool providers resort to measurement-based approaches [17] to derive contention bounds, as done for the Freescale P4080 processor [18].

- **Measurement Based Timing Analysis** executes the program on the real-platform under stressing conditions and collects measurements, This approach also requires certain level of understanding of its behavior, in order to measure the longest and more sensitive path in the code of the program. Those measures are latter operated to derive a bound to the timing behavior of the application. For instance, the longest-observed execution time, or high water-mark time, is recorded and inflated with a safety margin (e.g. 20%). For multicores, the reliability of measurement based timing analysis provided results depends on, among others, ensuring that in the experiments performed the application suffers the maximum contention in its access to the hardware shared resources, but also in the experience of engineers [15]. Designing a proper set of experiments and analyze the measurements taken requires knowledge and proper documentation about the hardware shared resources and the available hardware tools to monitor the execution. Even thought the amount of documentation available and its level of detail required for this approach compared with a static timing analysis approach is lesser, it has a direct impact on how tight the estimates can be.

Timing Analysis of Commercial Off-The-Shelf hardware. Current COTS multicores are designed to improve average performance rather than time predictability, which is an essential ingredient to compute tight and sound WCET bounds for real-time software programs. This kind of processors usually implements several hardware resources and optimizations that are very disruptive for the timing analysis required for critical embedded systems, some of them as common and widespread as shared caches and out-of-order execution. Also, COTS hardware is aimed for a wide market that until

now does not demand the amount of documentation that would be helpful for timing analysis. Sadly, at the present state of the art, analysis solutions capable of delivering tight and sound worst case-execution bounds for COTS multicores are not fully mature.

Time Composable Worst-Case Execution Time estimates. Every new critical embedded system generation sees an increase in the number and complexity of their components. The timing analysis of each component, and also of the system as a whole, is not an easy task, but it can be eased if the analysis of the components is time composable. We use the term *compositional* to mean that some properties of an individual part of the system can only be determined on (assumed) knowledge of the constituents of the system. This is in contrast with the term *composable*, which regards those properties of an individual part that can be determined considering that part in isolation and hold true on composition into the system [19]. Applied to WCET and multicores, a WCET estimate would be composable if it is not dependent on the load co-runner tasks put on hardware shared resources.

Early Timing Estimates. While time composable WCET estimates are desirable, they may account for over-pessimistic interference and lead to pessimistic WCET estimates w.r.t. real operation conditions. In this scenario, one may want to account for some information on contender applications to tighten WCET estimates. However, multicores challenge validating that the application fits its assigned timing budget since the timing behavior of one Software Provider's application depends on how other (contender) applications – likely developed by other Software Providers – use multicore shared resources. To make things worse, contender applications may not be shared among Software Providers or with the integrator for Intellectual Property reasons. This poses new challenges in deriving time estimates during early design phases, relegating time budget testing to late design phases. Clearly, the cost of managing any violation of the time budgets significantly increases during late design phase, potentially jeopardizing the whole design and product's time to market.

1.3 Contributions

While COTS multicores offer a number of advantages for critical embedded system industry, their complexity challenges the overall timing validation and verification process. This Thesis proposes several methodologies to i) increase the confidence on derived WCET bounds; ii) increase the time composability of derived WCET bounds; and iii) help deriving WCET estimates as early as possible during early development phase and system integration. The main contributions of this work can be summarized as follows:

1. **Taxonomy of existing approaches**. Over the years, the critical embedded system community has devoted considerable attention to the impact on execution time that arises from contention on access to hardware shared resources. The relevance of this problem has been accentuated with the arrival of multicore processors. From the state of the art on the subject, there appears to be considerable diversity in the understanding of the problem and in the approach to solve it. This sparseness makes it difficult for any reader to form a coherent picture of the problem and solution space. As first contribution of this Thesis we provide a taxonomy to categorize each known approach to the problem based on its specific goals and assumptions. This piece of work aims at becoming a reference publication for future works on this area.

2. **Increasing the confidence on MBTA WCET estimates**. For measurement based timing analysis, the most used timing analysis technique in industry, one of the main challenges when it comes to time analyze COTS multicores, is deriving the worst-case impact that contention can cause on the access to hardware shared resources. This longest-possible delay, usually referred to as *ubd* (upper-bound delay), is central to derive the worst-case contention that tasks can suffer. State-of-the-art techniques used to compute *ubd* employ resource stressing kernels (RSK) [20][21][22] that put high load on the shared resources. However, we show that those techniques do not achieve the goal of exposing the highest contention. With focus on two of the most used fair arbitration policies, round robin and first-in first-out, we show that under heavy contention scenarios, a "*synchrony effect*" arises that causes each request issued to suffer a contention delay that can be systematically inferior to *ubd*. This challenges the use of measurement based timing analysis together with resources stressing kernels. We propose a measurement-based methodology to accurately derive *ubd* without needing latency information from the hardware provider. Experimental results, obtained on multiple processor configurations, demonstrate the robustness of the proposed methodology.

3. **Signatures and Templates to increase Time Composability**. Contention in the access to hardware shared resources causes that task's timing behavior depends on its co-runners, and in particular, on the the load they put on shared resources. This dependence, negatively affects (reduces) time composability and constrains incremental verification. To attack this problem, we introduce the concepts of resource-usage signatures and templates, to abstract the potential contention caused and incurred by tasks running on a multicore. Building on them, we propose an approach that enables the analysis of individual tasks largely in isolation, with low integration costs, producing execution time estimates per task that are easily composable throughout the whole system integration process. Templates

and signatures make the WCET estimate derived for task τ, time composable with respect to a particular usage u of the hardware shared resources made by the interfering co-runner tasks. The WCET derived under u upper bounds τ's execution time under any workload as long as the co-runners of τ can be proven to make a resource usage smaller than u. As a result, the system integrator only needs to characterize the the tasks' access to hardware shared resources (a low-cost abstraction of the task execution time), ignoring any finer-grain detail of that access behavior.

4. **Surrogate Applications for Early Design Phase WCET estimation**. Properly allocating time budgets to applications during system's early design phase prevents costly-to-handle time over-runs in late design phase. Applications running on a multicore affect each others' behavior, which complicates reaching this goal. Further, in multi-provider software developments, software providers are reluctant to share their applications for intellectual property reasons. Both factors prevent deriving tight bounds until late design phase when applications are actually integrated. We propose a modelling approach that simplifies time budgeting in early design phase by developing surrogate applications (SurApps) and an automatic framework to generate them. A SurApp copies the non-functional behavior of a given target application automatically. Each software provider generates, for an application App_A, a surrogate application $SurApp_A$ and gives it to other providers without the risk of revealing any intellectual property. By running their applications against $SurApp_A$, other providers obtain a tight estimate of the slowdown their applications will suffer when run against App_A. This process is repeated for all providers facilitating early time budgeting for multicores.

5. **Assessment of a COTS architectures (ARM big.LITTLE) for critical embedded systems**. COTS processors pose a number of challenges for their use in critical embedded systems. Amongst COTS architectures, we regard high-performance ARM architectures as very popular in consumer electronics and increasingly tested in critical embedded systems. However, time budgeting on those architectures is an open problem. Thus, in this thesis we assess the suitability of the ARM big.LITTLE architecture for real-time critical embedded systems by attempting to measure the maximum contention that can be experienced in the access to shared resources. We perform this task in two different incarnations of this architecture: the Qualcomm SnapDragon 810 processor and the ARM Juno System-on-Chip. Our qualitative and quantitative assessment of these boards provides indications of how they can be used for critical embedded systems.

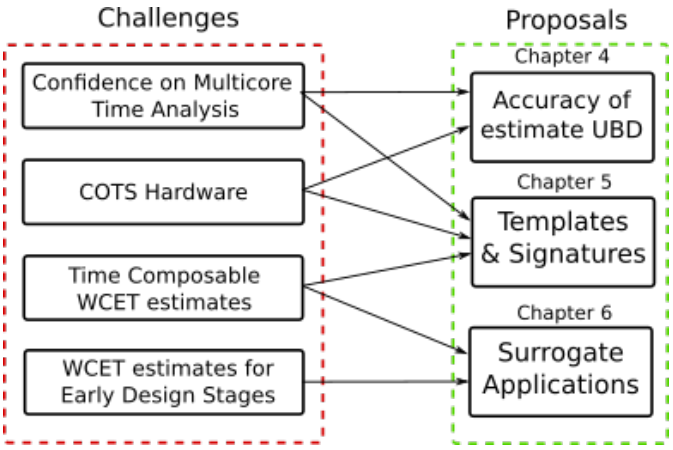


FIGURE 1.1: Challenges identified for future critical embedded systems and main proposals of this Thesis

Overall, the ultimate goal of this Thesis is to provide software support, in the form of concepts and specific methodologies, to improve the quality (i.e. tightness and reliability) on WCET estimates that can be derived with measurement based timing analysis, including early design stages of the software.

1.4 Structure of this Thesis

Each of the major technical contributions of this Thesis covers some of the main challenges identified for critical embedded systems, see Figure 1.1. Further, each such contribution is mapped to a different chapter: Increasing accuracy deriving *ubd* is presented in Chapter 4; Signatures and Templates in Chapter 5; and Surrogate Applications in Chapter 6. In addition to those:

- Chapter 2 introduces our experimental methodology in terms of simulation platforms and applications.
- Chapter 3 proposes a taxonomy of the works in the state of the art.
- Chapter 7 presents the results of our first-hand experience with timing analysis on two different ARM big.LITTLE-based boards.
- Chapter 8 summarizes the main conclusions of this Thesis and presenting the main future directions.

1.5 List of Publications

This Thesis has resulted in the following publications:

- Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega and Francisco J. Cazorla.
  "Contention in multicore hardware shared resources: Understanding of the state of the art."
  In 14th International Workshop on Worst-Case Execution Time Analysis (WCET). Madrid, Spain. July, 2014.
  DOI: 10.4230/OASIcs.WCET.2014.31

- Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quinones, Tullio Vardanega, Francisco J. Cazorla
  "Increasing Confidence on Measurement-Based Contention Bounds for Real-Time Round-Robin Buses"
  Design Automation Conference (DAC) San Francisco, CA. June, 2015.
  https://doi.org/10.1145/2744769.2744858

- Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quinones, Tullio Vardanega, Francisco J. Cazorla
  "Resource Usage Templates and Signatures for COTS Multicore Processors"
  In Design Automation Conference (DAC) San Francisco, CA. June, 2015.
  https://doi.org/10.1145/2744769.2744901

- Gabriel Fernandez, Jaume Abella, Eduardo Quinones, Luca Fossati, Marco Zulianello, Tullio Vardanega, Francisco J. Cazorla
  "Computing Safe Contention Bounds for Multicore Resources with Round-Robin and FIFO Arbitration"
  IEEE Transactions on Computers (Volume: 66, Issue: 4)
  http://dx.doi.org/10.1109/TC.2016.2616307

- Gabriel Fernandez, Francisco J. Cazorla, Jaume Abella.
  "Consumer Electronics Processors for Critical Real-Time Systems: a (Failed) Practical Experience"
  The ERTS2: Embedded Real Time Software and Systems, Toulouse, France. January, 2018.
  https://hal.archives-ouvertes.fr/hal-01708723

- Gabriel Fernandez, Jaume Abella, Guillem Bernat, Francisco J. Cazorla
  "Surrogate Applications for Early Design Stage Multicore Contention Modeling"
  IEEE Transactions on Emerging Topics in Computing 2018.
  http://dx.doi.org/10.1109/TETC.2018.2852760

- Gabriel Fernandez, Francisco J Cazorla, Jaume Abella and Sylvain Girbal
  "Assessing Time Predictability Features of ARM big.LITTLE Multicores"©2018 IEEE
  In International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Lyon, France, September 2018.

1.5.1 Related publications

The following publications were done as preamble to this thesis during my master studies.

- Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Luca Fossati, Marco Zulianello, Tullio Vardanega, Francisco J. Cazorla
  "Introduction to Partial Time Composability for COTS Multicores".
  In 23th ACM/SIGAPP Symposium On Applied Computing (SAC). Salamanca (Spain), April 13-17 2015.

- Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Luca Fossati, Marco Zulianello, Tullio Vardanega, Francisco J. Cazorla
  "Seeking Time-Composable Partitions of Tasks for COTS Multicore Processors"
  In IEEE International Symposium On Real-Time Computing (ISORC) Auckland, New Zealand. April, 2015.

Chapter 2

Experimental Framework

This Chapter introduces the experimental infrastructure used to quantitatively assess the proposals made in this Thesis. It also provides a description of the benchmarks and reference applications used to that end.

In terms of experimental infrastructure, we use the following:

1. COTS hardware multicore boards. These are specially used for the software-only proposals.
2. Architectural simulators. The use of simulators is widespread in industry and academia for low-level performance analysis and hardware design. In this line we use simulators for our hardware proposals.

In terms of benchmarks we use:

1. Representative benchmark suites commonly used in research in the area of critical embedded systems (MediaBench and EEMBC Autobench).
2. Space representative applications provided by the European Space Agency.
3. Synthetic applications aimed at capturing extreme (corner) behaviours.

2.1 COTS Hardware Multicore Platforms

In order to fulfil the experimental needs of this Thesis, we have used several hardware platforms. In this section we describe the architecture of the platforms and the boards where these architectures are implemented. We use two very different architectures: LEON and ARM.

2.1.1 The LEON processor

LEON processors are provided by Cobham Gaisler AB as synthesizable VHDL models. We use two versions of these processors: LEON3 and LEON4. Both of them are 32-bit cores compliant with the SPARC V8 architecture, but they are highly configurable, making them suitable for systems-on-chip (SoC) designs.

- **LEON3** implements a 7-stage pipeline with hardware multiply, divide and MAC units. The cache follows the Harvard architecture, separating instruction and data cache, see Figure 2.1. Caches comprise ways whose size may range from 1 to 256 kilobytes. They implement least-recently used (LRU) replacement policy. LEON3 uses an AMBA-2.0 AHB bus interface that is used to connect with external components (e.g. memory controller). The core has also Advanced on-chip debug support with instruction and data trace buffers. The source code is available under GNU GPL license, free for research and educational purposes, but it is also available under a low-cost license for commercial purposes.

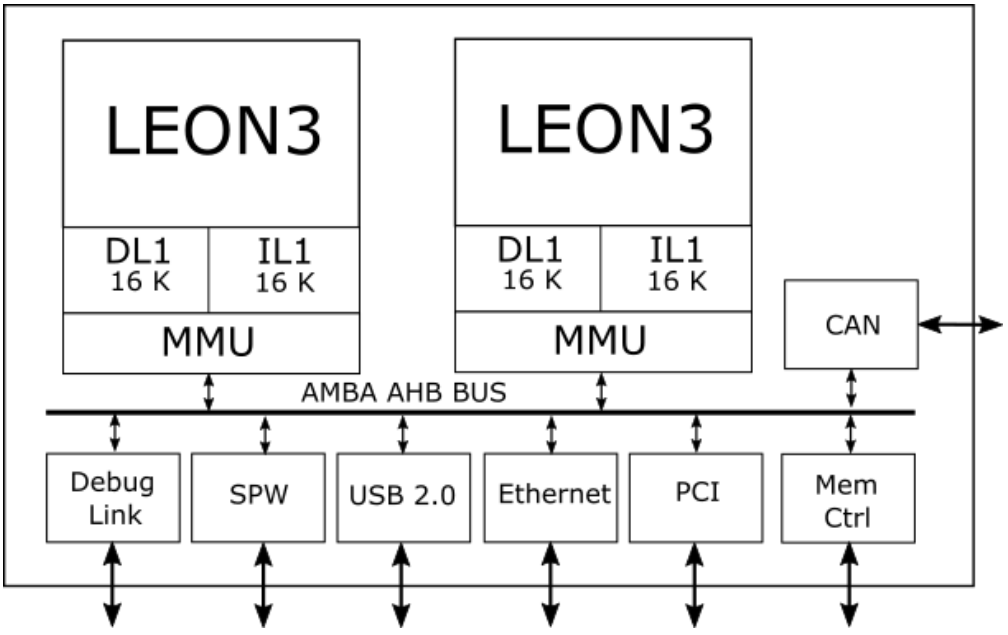


FIGURE 2.1: Block diagram of the GR712RC implementation of the LEON3 architecture used in this study.

- **LEON4** implements a 7-stage pipeline with a branch predictor. Even though the ISA is 32-bits, the core implements a 64-bit internal data path for loads and stores. The AMBA-2.0 AHB interface can be configured to use either 64 or 128 bits. The cache follows the Harvard architecture, separating instruction and data cache. First level caches also comprise 1 to 256 kilobytes per way, and implement LRU replacement policy. The core has also Advanced on-chip debug support with

instruction and data trace buffers. The source code is only available under a low-cost commercial license.

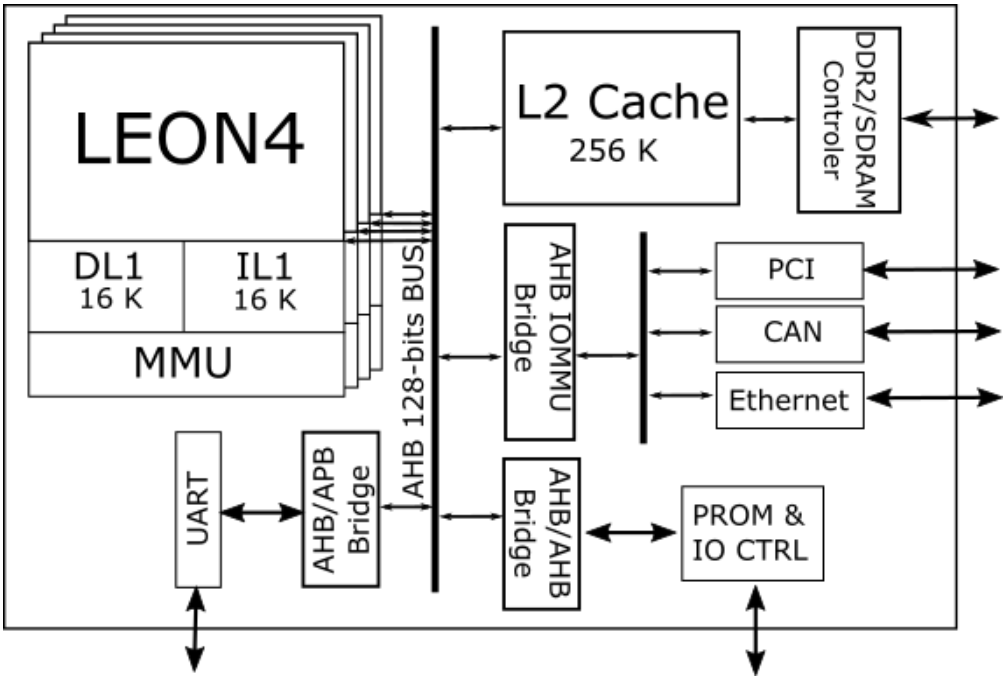


FIGURE 2.2: Block diagram of the ML510 implementation of the LEON4 architecture used in this study.

2.1.2 The LEON platforms

We have used two platforms that implement the LEON3 and LEON4 multicore processors respectively: the GR712RC and the ML510 respectively.

- The **GR712RC** platform implements an ASIC with two LEON3 cores (see Figure 2.1). Each one comprises private first-level 16KB data and 16KB instruction caches, both being 4-way and 32 bytes/line. The ASIC is connected to the on-chip SRAM and the memory controller through an AMBA AHB bus [23]. The memory controller connects both cores to the off-chip SDRAM and SRAM devices. In the GR712RC there is no shared L2 cache, so the effect of the slowdown that a task suffers is mainly due to inter-task interferences in accessing the on-chip bus to reach main memory.

- The **ML510** platform contains a Virtex 5 FPGA on which we use a preliminary implementation of the Next Generation Multipurpose Processor (NGMP) [24, 25] owing to FPGA space limitations, this platform does not have an on-core floating point unit. The NGMP was developed by Cobham Gaisler and the European Space Agency. This implementation comprises 4 LEON4 cores with always-taken

branch predictors and private data and instruction caches of 16KB each. Both the instruction and the data caches have 32-byte lines and are 4-way associative. The data cache employs a write-through with no-allocate miss policy. Each LEON4 core connects to a shared 256KB L2 cache through an AMBA AHB processor bus with 128-bit data width and round-robin arbitration policy. The L2 cache uses LRU replacement policy and implements a write-back, write-allocate policy. The L2 cache connects to the memory controller through a single memory channel shared by all cores (see Figure 2.2), but its space can be partitioned assigning ways to specific cores. In the NGMP, the effect of the slowdown that a task (benchmark) suffers is due to inter-task interference in accessing the on-chip bus, the on-chip shared L2 cache and the memory bandwidth [26].

2.1.3 The ARM big.LITTLE architecture

The ARM big.LITTLE architecture is mostly intended for mobile applications, although it is also used in other domains. It implements two clusters with different types of cores: a high-performance cluster with Cortex-A72 or Cortex-A57 cores, and a low power cluster with Cortex-A53 cores. The purpose of this design is enabling both, high-performance operation despite relatively high energy consumption, and low-power operation despite relatively low performance. This IP is highly configurable by the chip manufacturer. Next, we provide some details of this architecture. However, a large fraction of details are not provided in the documentation, and some others are specified as "implementation dependent" in ARM documentation.

- **Cortex-A72 and Cortex-A57**: this IP provided by ARM uses the architecture Armv8-A. It can work alone or in a cluster with up to 4 cores. The cluster can be interconnected using AMBA 5 CHI or 4 ACE buses. These cores are 64-bit supporting the AArch64 ISA and the AArch32 ISA for full backward compatibility with Armv7 code. These cores are designed for high performance so they implement a triple-issue out-of-order pipeline and a branch predictor. Aside, all the cores in the cluster share a L2 cache that can go from 512 KB to 4MB.

- **Cortex-A53**: as the A72 and A57 cores, the A53 may be deployed in clusters of 1 to 4 cores. They also uses the Armv8-A architecture, able to use the AArch32 and AArch64 ISAs. These cores are designed for power efficiency so the pipeline is in-order and dual-issue. Still, it has a branch predictor. It includes several power saving features like hierarchical clock gating. The cluster also shares a L2 cache, but its size can only be in the range 128KB up to 2MB.

2.1.4 The ARM platforms

During the experiments we did for this Thesis, we used originally the platform SnapDragon 810, but later on we switched to another platform the Juno Board. The design of the processor is the roughly the same, but implemented by different manufacturers. The reason behind using both boards is a technical issues in the SnapDragon 810, in Chapter 7 we describe in detail this issues.

- The **SnapDragon 810 processor** is a Qualcomm implementation of the ARM big.LITTLE architecture used by Sony in some of their Xperia devices during 2015. It comprises abundant hardware events that can be tracked with Performance Monitoring Counters (PMCs), so conclusions obtained on this specific processor apply to several others in the consumer electronics market, especially those building upon ARM big.LITTLE architecture and those implemented by Qualcomm.

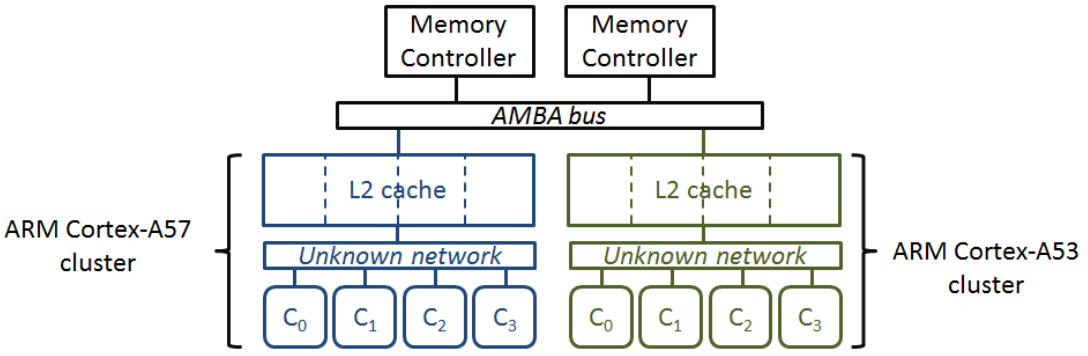


FIGURE 2.3: Schematic view of the elements of the SnapDragon 810 processor

The architecture of the processor, shown in Figure 2.3, comprises 2 clusters (also referred to as processors according to ARM's nomenclature): an ARM Cortex-A57 cluster with 4 cores and an ARM Cortex-A53 cluster with 4 cores. Each core is equipped with local first level instruction (IL1) and data (DL1) caches. Caches of the cores in one cluster are connected to a shared L2 cache, local to the cluster. An AMBA bus interface connects both clusters to two shared memory controllers to access DRAM. Peripherals and accelerators, also present in this platform but not shown in the figure, are connected to the AMBA bus too. In this Thesis we discount their effect by keeping them either disabled or idle.

Both clusters (A57 and A53) and the AMBA bus have been developed by ARM, the IP provider. Qualcomm, the chip manufacturer, integrates those components along with some others, which may or may not be provided by ARM. Moreover, in the integration process, Qualcomm may have introduced modifications in some IP

components and/or their interfaces w.r.t. what is described in the documentation (not only those parameters regarded as implementation dependent).

- The **Juno SoC** is a Development Platform provided by ARM intended for software development. As the previous platform, it implements a big.LITTLE configuration with a Dual-core Cortex-A72 processor and a Quad-core Cortex-A53. The former has 48KB IL1, a 32KB DL1 and a shared 2MB L2 cache. In the latter cluster, it IL1 and DL1 caches of 32KB and a shared L2 of 1MB.

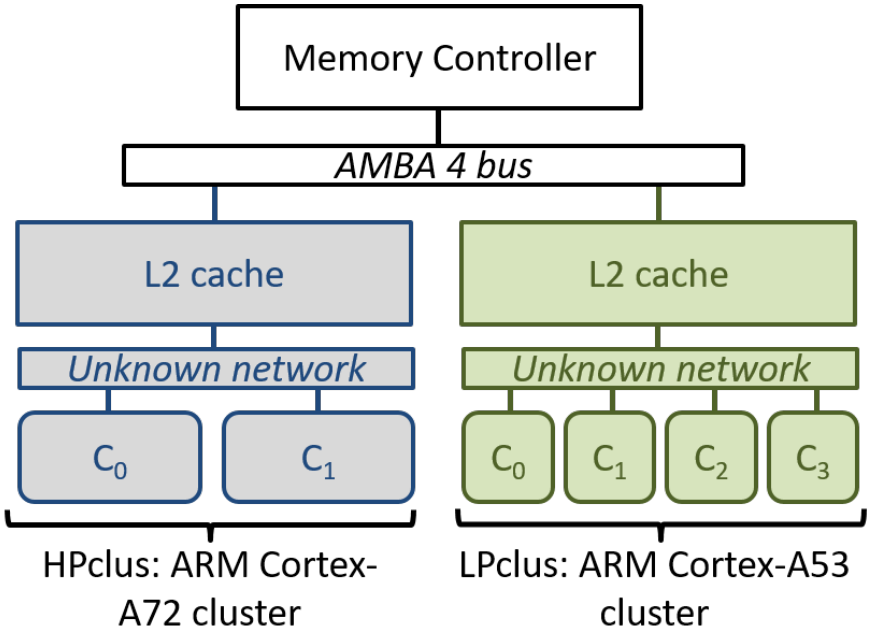


FIGURE 2.4: Schematic view of the elements of the Juno SoC processor

2.2 Simulation Insfrastructure

We use a modified version of the SystemC component library SoCLib [27] as baseline platform in order to create a cycle accurate execution-driven simulator. SoCLib is an open platform for virtual prototyping especially for multi-core systems. It is designed to be programmed using Object-Oriented Coding, easing the creation of modular elements and providing tools to connect and synchronize them. For this purpose it is build upon the SystemC framework. Its modular organization helps the development and maintenance of the system. In this way each component (e.g. cache, core, memory, etc.) is coded individually and then easily connected to the rest of the system. Aside we can divide the system in two entities: the emulator and the timing simulator. The former handles the functional execution of the guest code and the latter uses the information provided by the first in order to provide cycle accurate timing. SoCLib has two types

of models: Cycle Accurate/Bit Accurate (CABA) and Transaction Level Modeling with Distributed Time. The implementation we use follows the former model.

We use an emulator that mimics a LEON4 processor core with the same configuration as the ML510 platform, previously described. But this configuration may be modified in some Chapters for experimental purposes. The main target of this modifications is the memory hierarchy configuration. Each Chapter describes the specific cache configuration used. The last level of the memory hierarchy is the memory controller which has been simplified to always hit using the same timing parameters that the ML510 platform. Moreover, the arbitration policy has been changed in Chapter 4. In a study with the European Space Agency [27], the performance accuracy provided by the simulator was assessed against a real NGMP implementation, the N2X [28] evaluation board. To that end, a low-overhead operating-system (kernel) that allowed cycle-level validation was deployed. Results for EEMBC benchmarks showed a deviation in terms of accuracy of less than 3% on average. For the HAWAII benchmark [29], an algorithm used to process raw frames coming from the sate-of-the-art near infrared (NIR) HAWAII-2RG detector, the inaccuracy reduced to less than 1%. Following the modular philosophy of SoCLib, we added new instrumentation inside the timing simulator in order to assess our solutions in an agile manner.

2.3 Benchmark Suites

Interestingly so far multicores are mainly exploited in real-time systems by consolidating independent applications on the same chip. Those applications either share no data or the sharing is performed off-chip (e.g. on memory). This is made more evident as current timing analyses rely on splitting the last-level cache, hence, preventing data sharing. Hence, while it is clear that in the long-term parallel execution will be adopted, this is still a recent (and very active) research area of research not yet ready for industrialization. Overall, we build on the fact that applications do not share data and hence the coherence mechanism does not disturb the execution time measurements.

2.3.1 EEMBC Autobench

The Embedded Microprocessor Benchmark Consortium (EEMBC) [30] has produced several benchmark suites. Among them, we have chosen the Autobench set that is designed for the automotive industry. All the benchmarks in the suite share a common loop that executes several times an algorithm that changes across benchmarks. These algorithms include bit manipulation, matrix mapping, a specific floating-point tester, a

cache buster, pointer chasing, pulse-width modulation, multiplication, and shift operations.

Table 2.1 lists all benchmarks providing their acronym and their description.

TABLE 2.1: EEMBC names

Name	**Description**
a2time	Angle to Time Conversion
basefp	Basic Integer and Floating Point
bitmnp	Bit Manipulation
cacheb	Cache 'Buster'
canrdr	CAN Remote Data Request
aifft	Fast Fourier Transform (FFT)
aifirf	Finite Impulse Response (FIR) Filter
aiifft	Inverse Fast Fourier Transform (iFFT)
aiirflt	Infinite Impulse Response (IIR) Filter
matrix	Matrix Arithmetic
pntrch	Pointer Chasing
puwmod	Pulse Width Modulation (PWM)
rspeed	Road Speed Calculation
tblook	Table Lookup and Interpolation
ttsprk	Tooth to Spark

2.3.2 Mediabench

The Mediabench benchmark suite is composed of multimedia applications. Real-time systems are used more and more for executing this kind of applications, which makes this suite highly representative. Aside, the applications in this suite have a considerable bigger memory footprint than EEMBC, which makes them more attractive for certain parts of our research. Table 2.2 lists all the benchmarks and a simple description of each one. Several of these benchmarks are multimedia compressors, so they provide a encoder and decoder binary.

TABLE 2.2: Mediabench names and descriptions

Name	Description
adpcm	Simple and well known audio coder
epic	Image compression application
g.721	Voice compression application
ghostscript	PostScript interpreter
gsm	Full-rate speech transcoding
jpeg	JPEG standard compressor
Mesa	3D graphic library
mpeg	MPEG2 standard compressor
pegwit	Public key encryption
pgp	"Message digests signature" generator
rasta	Speech recognition application

2.4 Space Applications

In addition to the previous benchmark suites, we use some applications used by the space industry.

- OBDP benchmark [29], is an algorithm used to process raw frames coming from the state-of-the-art near infrared (NIR) HAWAII-2RG detector, already deployed in space projects like the Hubble Space Telescope.
- DEBIE software controls an instrument, which was carried on PROBA-1 satellite, to observe micrometeoroids and small space debris by detecting impacts on its sensors, both mechanically and electrically.

2.5 Resource Stressing Kernels

During this Thesis we make use of *micro-kernels* [17, 26, 31], a set of single-phase user-level programs with a single execution behavior designed so that all their operations access a given shared resource, e.g. the bus. Due to their purpose these micro-kernels are referred as *Resource Stressing Kernels* (RSK) These micro-kernels consist of a main loop whose body includes a substantial number (e.g. 256) of instructions designed to generate a steady stress load on target resources. The fact that the loop body executes repeatedly the same instruction causes the target resource to be continuously accessed. Moreover, placing a high number of identical instructions in the loop body drastically reduces the impact of control instructions (down to 2-4%) [26]. The shape of a RSK

depends on the platform where is going to be executed and of its purpose. For this reason, more details on the RSK are provided n those Chapters where they are used.

Chapter 3

A Taxonomy of the Works in the State of the Art

This Chapter presents the first contribution of this Thesis, namely, a taxonomy of the related works on the topic of multicore contention analysis. It also introduces some basic concepts that are further used in the rest of this document. It is noted that, the most specific related works to each proposal are presented in the corresponding Chapters describing those proposals. From that point of view, this Chapter does not provide an exhaustive list of all related works but a categorization of those.

Different angles have been considered to address the timing effects of contention for shared resources, either on-chip or off-chip, leading to a number of families of techniques. However, while many solutions are claimed to exist, there is evident lack of common understanding of the problem space, in terms of processor features, and of the assumptions made to solve it. We attempt to cure this problem with the following taxonomy.

3.1 A possible taxonomy of state-of-the-art techniques to analyse the timing impact of resource contention

Under the umbrella term of resource contention, we capture the various forms of timing interference that software programs suffer owing to access to shared hardware resources. Notably, our analysis does not cover the contention on access to software resources. Furthermore, contentions arising from parallel execution of a software program fall outside of our analysis and are recognized in Section 3.2 as an important emerging ramification of the problem.

The challenge of contention in multicore processors has been addressed with various approaches. In this chapter we classify them in four broad categories, dependent on where they seem to direct their focus:

1. on system considerations, which address the contention problem top down, from the software perspective;
2. on WCET considerations, which take the opposite view, studying how contention phenomena affect the timing behaviour of the software;
3. on architecture considerations, which devise processor features and arbitration policies that help achieve composable timing behavior; and
4. on Commercial Off-The-Shelf(COTS) considerations, which propose processor-specific ways to deal with processor-specific contention and arbitration features.

We discuss the approaches in each category in isolation and we break them down into subgroups where appropriate.

3.1.1 System-centric techniques

System-centric techniques take a top-down approach to the problem. The techniques in this category take an off-chip, hence coarse-grained, perspective. Off-chip resources have longer latency than on-chip resources, and also a higher degree of visibility from the software standpoint than on-chip resources. For instance, at software level one can easily tell where a set of addresses is mapped to memory, but it is (much) harder to determine which data item is in cache at a given time and which is not. Similarly, software 'sees' (hence can program or directly effect) higher-level resources such as the Direct-Memory Access, but it is unaware of (and unable to directly control) low-level resources like a bus for inter-core communication and, other than in very special cases, the on-chip shared cache. In fact, the cache impact has characteristics that can be captured, from different angles and with different precision, with techniques that we classify in different categories of our taxonomy (system-centric and WCET-centric). Besides this cross-boundary overlap, the techniques in this category predominantly focus on off-chip resources.

We single out three angles worth of specific discussion: timing analysis frameworks; access scheduling and allocation; and works on COTS architectures.

3.1.1.1 Timing analysis frameworks

Two main characteristics are predominant in the techniques in this sub-class that differentiate them from similar techniques captured in our taxonomy. In general, these techniques assume that on-chip shared resources (e.g. core-to-cache bus, caches, ...) are replicated or partitioned across cores, so that software programs allocated to a core suffer no contention on access to on-chip resources. Their analysis frameworks model off-chip shared resources in isolation and provide worst-case access timing bounds for them. The impact of contention is only considered for those resources and it is captured compositionally, when the WCET of the software program, determined assuming no contention, is increased by delay factors that consider the sources of off-chip contention in the presence of co-runners.

The shared resources considered in this class of approaches are assumed to process one request at a time. It is also assumed that the corresponding services cannot be preempted (or split or segmented). It is further assumed that the requests are synchronous so that the requesting task is stalled while the access request is served. The analysis focuses on individual tasks, whose program units are logically divided into blocks for which maximum and minimum access bounds and execution time bounds can be (more easily) derived.

For approaches in this class, the access to the shared resource is assumed to be arbitrated by either a TDMA bus [32] or a dynamic arbitration bus [33] or else an adaptive bus arbiter [34]. For TDMA buses, focus is on determining the worst-case alignment of the requests in the TDMA schedule. As the bus schedule is static, co-running tasks do not affect one another execution time, which makes their execution time composable with respect to the bus. The fact that service is assumed run-to-completion and that requests do not overlap simplifies the problem.

For dynamic arbiters, the workload that a task places on the shared resource affects the access time of the co-running tasks, which breaks time composability. This type of arbiters has generated a research line of their own. Several authors [33, 35] have proposed various approaches to derive bounds for the number of accesses per task in a given period of time. The timing analysis for an individual task therefore depends on the request workload generated by the co-runners in that time duration. Interestingly, while the number of accesses that a task generates to the resource can be considered intrinsic to the task (i.e independent of the co-runners) as long as caches are partitioned, the task's frequency of access depends on how often the co-runners delay the task's requests. The cited models capture that dependence.

Adaptive arbiters (as in, e.g., FlexRay) exhibit a window with per-requester slot scheduling combined with a window in which requests are dynamically arbitrated; this trait makes them show characteristics that we have seen above as distinctive for static and dynamic arbiters.

The authors of [36] provide a useful survey of how time-deterministic approaches to bus arbitration and scheduling for multicore processors can be captured, compositionally, by timing analysis techniques. The cited work also presents benchmark-based empirical evidence of the degradation that TDMA arbitration causes to average-case performance in comparison to other techniques with acceptable characteristics in terms of time determinism.

3.1.1.2 Task scheduling and allocation

The state-of-the-art approaches to multicore scheduling and schedulability analysis that match the techniques which fall in this category can be grouped in two sets: those that ignore contention issues at their level and leave it to WCET analysis; and those that consider it as a factor of influence to task allocation, which is adjusted to attain increased schedulable utilization.

There essentially exist three classes of scheduling algorithms, which differ in the way they assign tasks to cores [6]. Partitioned and global scheduling place at the respective extremes of the spectrum: the former statically maps tasks to core, so that a task can only be scheduled on the core it has been assigned to; the latter allows tasks to migrate jobs from one core to another. The latter are work conserving, at the expense of possibly costly task (job) migrations. The former risks considerable under-utilization of the processing resources, especially for tasks with medium to high loads. The middle of the spectrum is occupied by clustered or semi-partitioned algorithms, which – with various techniques and for different goals – only allow or cause statically-determined groups of tasks to migrate within statically or dynamically determined subsets of cores.

Contention oblivious. The principal works on scheduling algorithms and associated schedulability analysis for multicore processors assume that the WCET of all tasks is given in input. They therefore assume that the WCET bounds may be determined before decisions are made on task mapping to cores and on scheduling at run time. This is tantamount to postulating that the WCET bounds are composable, that is to say, free from variations determined by the presence of contenders in the system. Ironically, the only plausible way in which WCET bounds can actually be made to be composable for use in schedulability analysis for multicore processors is by increasing them compositionally, by a factor determined by given patterns of conflicts that are

asserted to bound from above the actual contention delays suffered at run time. In essence, the approaches of this kind escape the intrinsic (and painful) circularity between the dependence on WCET analysis on knowledge of the contenders and the dependence of schedulability on knowledge of the WCET of the tasks in the system, by inflating the WCET budgets so that they can always be trusted to upper bound the actual costs.

Contention aware. Techniques such as [37, 38] focus on the shared last-level cache as one of the main resources in which contention occurs. The cited works benefit from hardware proposals that split the cache into different ways or allocate program data into different pages (colours) so that each task is limited to use a subset of the sets in cache, thereby reducing conflicts.

These works often assume partitioned scheduling for software programs, so that conflicts can be determined in a less pessimistic way, and focus their attention on devising cache-aware allocation algorithms that consider the mapping of tasks to cores determined by partitioning. Some of the works focus on how to assign colours (i.e. set partitions) to the tasks. It is also the case that works in this class do not address the contention occurring in other shared resources like the memory.

Other works [39] build on hardware proposals that control the interaction in several hardware resources (e.g. on-chip bus, cache and memory controller) in addition to the cache. These proposals also consider task allocation and scheduling.

3.1.2 WCET-centric techniques

WCET-centric techniques determine the impact of contention in the access to shared resources as part of WCET analysis. For multicore architectures, shared resources include cache memories, buses and memory controllers, but some approaches have also been designed to support intra-core resource sharing (e.g., pipeline and functional units in multi-thread cores [40]). The objective of WCET-centric techniques is to derive safe stall times that can be accounted for at instruction-level timing analysis. We distinguish between the approaches that consider all the competing threads/tasks together to exhibit the possible interleaving of their respective accesses to the shared resource, from those that exploit a static allocation of slots among cores. In the latter category, some contributions include WCET-based strategies to optimize the mapping/scheduling of threads/tasks to cores to optimize the global WCET and/or to enhance schedulability.

3.1.2.1 Joint analysis of concurrent tasks/threads

One way to identify how contention may impact the WCET of a task is to combine the analysis of concurrent tasks to identify where they can interfere. Two kinds of interferences are considered here: spatial (tasks share storage, e.g. a cache) and temporal (tasks share bandwidth, e.g. a bus). Both incur additional delays.

Techniques that address spatial contention, first perform individual tasks analysis, then determine how contention affects their results. More precisely, they determine which cache lines used by one task might be replaced by another task in a shared L2 instruction [41][42] or data [43] cache. The analysis of contention does not account for the exact respective timings of tasks (then could be valid for any schedule, provided all possible concurrent tasks are known at analysis time). However, [42] improves the accuracy of the analysis by considering constraints on task scheduling (non-preemptive, priority-based, with task inter-dependencies), which allows bounding tasks lifetimes and limits the opportunities for contention.

To account for temporal conflicts and derive instruction timings, possible interleavings of (statically-scheduled) threads must be explored. Several approaches use timed automata to represent both the tasks and the state-based behavior of hardware components. All these automata are combined and model checking techniques are used to determine the WCET though a binary search process. [44] focuses on the shared L2 cache with fixed cache miss latencies. A shared bus with First-Come-First-Served (FCFS) or TDMA arbitration is analysed in [45]. The weakness of these approaches is the vast number of states to be handled.

3.1.2.2 Independent analysis of tasks/threads

Some techniques leverage deterministic guarantees offered by the underlying hardware on access to a shared resource. Thanks to such guarantees, they can analyse the WCET of one task/thread independently of the concurrent workload.

The impact of arbitration delays on a TDMA bus with uniform slot size is explored in [46]. The cited work presents an approach to evaluate the misalignment of accesses with TDMA slots (TDMA offsets). A TDMA-composable system is assumed: arbitration delays neither impact instructions that do not access the bus nor the bus access time (except for the arbitration delay).

Some solutions that enforce access guarantees in hardware do not offer equal opportunities to all threads. Cache partitioning may use partitions with different sizes [39]. Bus

arbitration may grant cores different numbers of slots [47] or [48]. Those techniques use mechanisms to increase the performance achievable by combined task-to-core allocation and scheduling decisions, especially in the case of unbalanced workloads (with variable demand levels to the shared resource). Performance benefits are obtained as a result of reducing the WCET bound predictions for the affected tasks. As noted in Section 3.1.1, our taxonomy is not clear-cut enough to place some of these techniques uniquely in one class, as they might arguably also belong to the system-centric group.

3.1.3 Architecture-centric techniques

Several hardware design paradigms have been proposed to deal with the inter-task interference caused by contention for shared hardware resources. Four topical approaches can be singled out in this group: the time-triggered architecture [49]; PRET [50]; CompSOC [51]; and MERASA [52].

One of the differentiating elements for these approaches is whether they achieve composability at the level of the WCET bounds that they allow computing or at higher levels of abstraction. The objective of the former solution is to support determining WCET bounds for individual tasks in isolation, independently of the activity of their co-runners. When that is guaranteed, the execution time of a task may well suffer variations caused by contention effects caused by some of its co-runners, but its WCET estimate stays valid. With the latter type of solutions, composability is achieved by regulatory mechanisms operating at run time, and thus with effect on the task execution time. Those regulatory mechanisms ensure that the activity of the co-runners cannot affect the response time of the hardware shared resources. This form of composability may place more requirements on the processor hardware than the former approach. In general it requires that the access time to a hardware shared resource stays always the same irrespective of the actual load of the system. To that end, a resource that might respond ahead of time is stalled until the agreed latency for the request is reached.

From another angle, it is worth noting that a trade off arises as a consequence of the observation that the pursuit of time composability always comes at the cost of some (over-provisioning) pessimism. The effect of this (static) over-provisioning allows tightening the WCET bounds, because they eradicate sources of variations, but at the cost of renouncing the true meaning of time composability (as independence from the presence of contenders), which is central to the incremental verification needs of integrated architectures such as Integrated Modular Avionics(IMA).

Somewhat orthogonal to the discussion above, the focus of several proposals is to upper bound the access time to hardware shared resources, either indirectly, by guaranteeing

pre-determined bandwidth on access to the resource, or directly by ensuring bounds on the access time (comprised of the wait time preceding access upon request, and the actual service time).

The techniques of interest from this angle vary for *stateless* and *stateful* resources. Stateless resources have an access time that is not or only very modestly dependent on execution history. A single-cycle latency bus is a typifying example of resources of this kind. If the bus had a two-cycle latency, then the service time of a request might depend on whether the preceding request was sent the cycle before the current one gets ready. In that case, the current request waits one cycle to be granted access and two extra cycle to effectively access the resource. Caches are a difficult example of stateful resources. This is because the state-dependent effect builds up with history of execution, which causes analysis to have to keep track of the full history of access. Truncated information requires conservative assumptions to be made. This difficulty explains why the typical solution proposed for caches consists in splitting its space in small areas assigned to individual tasks, so that history becomes much smaller (and free of conflicts with co-runners) and thus easier to trace. This can be done dividing the cache into different banks or different ways [53].

The most prominent stateless resources on which the real-time community has focused are network-on-chip (NoC) and memory controllers. For the interconnection network, proposals exist which range from from simple buses [54] or rings [55] to more complex solutions such as those described in [56]. All share the goal to provide some type of bound to the longest time a request has to wait to get access to the resource. For the memory controller, proposals with the same goal exist [57, 58], though the actual solutions are more complex since the state retention is higher.

3.1.4 COTS-based techniques

The goal of several works focusing on real hardware is to analyse how amenable a given multicore design is for real-time analysis. To that end authors analyse different shared resources as well as their impact on execution time. It is the case that for those resources the manuals of the processor under analysis do not provide all the required information to analytically derive those bounds. As a result, the way in which the authors derive bounds is by experimentation on the specific architectures under analysis [17, 26, 31]. These works include analysis of the FreeScale P4080 and some FPGA versions of the Aeroflex Gaisler LEON4.

Another set of works is carried out at an analysis level providing understanding of the timing behavior of hardware shared resources and the challenges they bring to timing

analysability [59–61]. Finally, some of the works on software-cache partitioning (page colouring) have been done for processors like the ARM Cortex A9 [38].

3.2 Other aspects of interest

In this section we briefly touch upon two other aspects that, for different reasons, are tangent to questions addressed in this thesis. One aspect, parallel programming, intrinsically enabled and called for by multicore processors, presents a novel, emerging challenge to bounding contention effects. The other aspect, with interesting potential and important ramifications, stems from shifting the angle of attack to the timing analysis problem, from finding a single value, the smallest possible computable upper bound, for all possible executions of a software program, to determining a probability function whose tail can be cut at the exceedance threshold of interest to the system.

3.2.1 Parallel applications

Parallel programming introduces software shared resources. Communication of data in message-passing and synchronisations in shared-memory programming induce delays that must be accounted for in execution times. The focus is on deriving the WCET of the whole set of threads together (i.e. the WCET of the longest thread) instead of individual WCETs.

Two kinds of synchronization exist. Mutual exclusion is very similar to accessing a hardware shared resource that can serve a single thread at a time. Computing the worst-case stall time of a thread at a critical section is analogous (when threads are served in a FIFO order) to computing the worst-case delay to a round-robin bus [62]. Stall times can then be integrated to instruction-level timing analysis. Another approach is to use timed automata and a model checker, as in [44]. In [63], a shared-memory parallel programming language is introduced and a fix-point analysis is able to identify all the possible thread interleaving at critical sections.

Progress synchronisation includes barriers as well as condition signalling and blocking message passing. Collective synchronisations (barriers), where all threads meet, are easier to consider since the goal is to compute the WCET of the longest thread, i.e. the last one to reach the barrier [62]. For point-to-point synchronisations (condition signalling or message passing) however, stall times depend on the respective progress of the threads. In [64], parallel applications where threads communicate through message passing are considered. A joint analysis is proposed, where the analysis of worst-case

communication times is integrated into the analysis of the global WCET. The approach consists in merging the control flow graphs of parallel threads, then adding edges to model the synchronisations (dependencies) related to sending/receiving messages.

Some system-centric approaches have been extended to parallel fork-join applications and decide altogether the allocation of threads' memory in caches, the scheduling of threads' accesses to the shared bus and the scheduling of the threads themselves to the cores [65].

3.2.2 The probabilistic approach

Timing analysis techniques can be broken down into *deterministic*, which produce a single WCET estimate, and *probabilistic* that produce multiple WCET estimates with associated exceedance probabilities. It is noted that our discussion above has focused on standard (deterministic) timing analysis techniques. While both deterministic and probabilistic approaches try to reach time predictability, the former do so by advocating for hardware and software designs that are deterministic in their execution time, while the latter advocates for hardware and software designs that have a randomized timing behaviour, to produce WCET estimates that can be exceeded with a given *probability*.

The probabilistic approach offers ways to deal with contention that differ from those deployed in the deterministic approach. On the one hand, in [66] several time-randomised bus arbitration policies are proposed as an alternative to deterministic policies such as round robin. In [67] it is proposed a time-randomised shared cache for which impact of contention among co-running tasks can be determined. The main feature of this cache is that it does not split the cache, either into ways or sets, to prevent the interaction among co-running tasks. Instead, it controls how often tasks evict data from cache as a way to bound the impact of contention on tasks' WCET estimates.

3.3 Critique

This section reviews the techniques captured in the taxonomy presented in Section 3.1 against multiple criteria including: (1) the presence of overlaps between them; (2) the presence of gaps among them; (3) the realism of the assumptions on which they base; (4) the challenges in taking that technique to industrial use; and (5) the relation between the confidence on the bounds determined by timing analysis and the assurance guarantees proper of the application domain.

Much like the proposed taxonomy, the review discussed here is not meant to be exhaustive. It therefore does not cover all criteria for all techniques. Instead, it only aims at singling out specific issues that we consider to need particular attention by the prospective user and further study by the research community. Regarding criterion (3) for example, it is interesting to observe that when time delays on access to a shared resource are computed separately from the execution time (which is the case for the system-centric approaches and also for some WCET-centric approaches), the important assumption is made that the relevant factors (and the behaviours that originate them) can be analysed compositionally in the time dimension [19].

System-centric techniques. The principal limitation with this class of techniques stems from their resting on two strong assumptions: that programs can be statically subdivided in blocks that can be studied in isolation; and that only one shared resource needs attention, which also does not support split transactions. The former assumption increases pessimism – hence decreases its attractiveness – because every single code section captured in the static breakdown of the problem is attached a single worst-case cost value, which may be considerably higher than the actual cost in the worst-case traversal of that block as taken by the program. The latter assumption instead reduces the applicability of the solution against increasingly common hardware.

For dynamic arbiters, the critical factor is in the dependence of their timing analysis on the request workload generated by the co-runners of the program of interest in a given time duration. On the one hand this trait reduces pessimism since the duration in which conflicts on access may occur can be better determined. On the other hand, it breaks time composability and resorts to compositionality. The latter defect may be a serious impediment to incremental verification, which is a prerequisite to high-criticality domains (e.g., avionics). Budgeting in advance for the co-runners is obviously one countermeasure to that, but at the direct cost of over-provisioning.

WCET-centric techniques. The main challenge for this class of techniques arises from having to find tractable ways to analyse increasingly complex hardware. The abstract interpretation approach on which those techniques base is inherently exposed to the state explosion problem, which is dramatically worsened by the way in which the architecture of modern processors cause the timing behaviour of several resources to exhibit possibly large jitter, extremely sensitive to the history of execution [61]. This dependence obviously accumulates bottom-up and manifests in very complex ways at software level.

As an example we consider a TDMA bus, whose timing behaviour is easy to model with three main parameters: window size, number of contenders, and slot size per contender. Interestingly, the state space for even such a simple model is already not negligible:

when the exact time of an access request cannot be determined in fact, a conservative assumption must be made on when access will be granted (which inflates pessimism) or multiple candidate access times are considered, which causes multiple states to be contemplated upward in the analysis. As more complex NoC architectures are adopted by modern multiprocessors, more parameters will be needed to model the sources of contention, with inordinate increase in the complexity and cost of the analysis tools.

Architecture-centric techniques. A recurrent question on the viability of the techniques in this class is whether the hardware design that they propose in the intent to favour time analysability, will ever hit the market. This is a question of economics that equally applies to all research domains that propose for hardware architectures. However, it is especially important to the real-time systems domain, which holds a tiny niche of the market size, in comparison to consumer products, without insufficient critical mass to swing the prevailing design criteria from optimized for the average case to well-behaved in the worst case.

This is a long-known challenge for the real-time systems community. Fortunately, perseverance and authority have shown able to win some battles, so that some of the proposed designs (e.g., cache partitioning) are indeed retained in real processors. Our view here is that the changes proposed for the bus and the memory controller are simple enough so that they can be implemented in production with moderate effort and cost, for tangible benefits on timing analysability. Whether or not that will actually happen remains to be seen.

In general all hardware approaches assume processor designs without timing anomalies. It is interesting wondering, whether processor cores can be made simple enough to assure freedom from timing anomalies, without this causing detriment to the attainable performance. Architectural solutions will have to be devised that combine those two objectives harmoniously, which is not the case yet with the dominant approaches to multi- and manycore processor architectures.

COTS-based techniques. The techniques that belong in this class face the challenge that the architectural properties needed to provide full time isolation or time predictable interaction among processor cores cannot be had owing to the lack or inaccuracy of specification information or IP restrictions. Various approaches have been proposed to live the consequent uncertainty, which all require building confidence arguments that accord with the requirements and practices of the application domain. The work in [68] makes an interesting review of how safety assurance guarantees relate to stipulating bounds on execution time.

3.4 Conclusions

A wealth of relevant literature addresses the problem of finding a bound to the timing effect of contention on access to hardware shared resources in modern multicore processors. The industrial practitioner, and the researcher alike, who approach that body of knowledge without a preconceived solution in mind, may have serious difficulties in seeing the "big picture" of what options are possible and at what consequences. This chapter sketches an initial taxonomy of the principal approaches that appear in the state of the art, and discusses gaps and overlaps among them.

Chapter 4

Increasing reliability on Measured UpperBound Delays(UBD)

One of the challenges of timing analysis for COTS multicores stems from the difficulty of determining the worst-case impact of contention on the access to hardware shared resources. In this Chapter, the term *ubd*, for *upper bound delay*, denotes that impact factor. Studies exist that investigate the *ubd* arising on access to the on-chip bus [54] and the memory controller [58, 69]. Those works however lead to a tight and sound *ubd* estimation only when enough information about the timing behavior of the target processor is available. Both *Static Timing Analysis* (STA) and *Measurement-Based Timing Analysis* (MBTA) methods [14] need trustworthy *ubd* to compute reliable WCET bounds. STA uses *ubd* to cost every request to a shared hardware resource issued by a software program. MBTA, which still is the most used practice in industry, needs to know *ubd* to gage the contention delay that may be suffered by application programs.

Unfortunately, as the complexity of multicores continues to grow and the information about their internal functioning is increasingly restricted by intellectual property, the static derivation of *ubd* becomes increasingly harder. As a testimony to that, the contention behavior of the P4080 processor has been analysed by an avionics end-user and a STA tool provider [18] using measurements, thereby obtaining a measured approximation of *ubd* [17], here denoted ubd_m, instead of proper *ubd*. The net consequence of that difficulty is that the confidence the user that can be placed on WCET bound rests on the confidence that can be attached to ubd_m, in particular, on how well it approximates the actual *ubd*.

To the best of our knowledge, the techniques used to compute ubd_m most frequently employ specialized programs executing in the application space, often called *resource stressing kernels*(see Chapter 2). The *rsk* approach computes the ubd_m by running

the *software component under analysis* (*scua*) against a battery of *rsk*. In particular, ubd_m is derived by dividing the execution-time increment suffered by the *scua*, (Δ_{ET}), owing to the contention generated by the *rsk* by the number n_r of access requests made by the *scua*: $ubd_m = \Delta_{ET}/n_r$. Interestingly, whereas *rsk* are expressly designed to produce high contention on a given shared hardware resource (e.g., the bus) so that the designated victim suffers high slowdown, insufficient attention has been devoted to determining whether *ubd* is best approximated by using the *scua* or a *rsk* as victim.

We show in this chapter that the state-of-the-art *rsk* methodology may fail at producing reliable ubd_m values. In particular, we analyse the impact that round-robin (*RoRo*) and fist-in-first-out (*FIFO*) arbitration policies, widely used in real-time systems due to their time-predictable traits [54] [53] , have on the computation of the ubd_m.

Overall, in the context of increasingly complex multicores where measurements are increasingly used to derive the impact of contention bounds, our approach becomes a fundamental step to attain trustworthy ubd_m to the contention in the bus and the memory controller, thereby increasing the trustworthiness of the WCET bound computed using that information.

4.1 Contention analysis for RoRo and FIFO

4.1.1 Studying the Bus and the Memory Controller

The interconnection network and the memory controller are two of the hardware resources whose sharing in multicores causes most bottlenecks for contending tasks that run in parallel. The determination of the *ubd* for those resources has already received the attention of researchers, under the hypothesis that public documentation on the internal functioning of the processor exists.

- Bus-based interconnection networks are known to require little energy as well as to ease protocol design and verification, at the cost of modest performance degradation [70, 71]. The Advanced Microcontroller Bus Architecture (AMBA) is a bus exemplar that is widely used in microcontroller devices as well as in a range of ASIC and SoC parts with real-time capabilities. The AMBA bus is in the focus of our work here.
- The memory controller, which policies access to memory and thus is necessarily shared across cores, causes considerable contention and exacts a high toll on the WCET bound. Several memory controller designs have been proposed to contain contention overhead [72][58][73][74], which we consider in this work.

We study how to derive *ubd* for those two hardware shared resources, assuming RoRo and FIFO arbitration for them. While there are other arbitration policies focused on improving average performance, they usually results in more pessimistic – or simply not boundable – *ubd*. This is the case of some types of priority arbitration [53] and policies like first-ready first come first serviced (FR-FCFS) [75].

We start by looking at each such policy in more depth in isolation.

RoRo. Consider a *RoRo*-arbitrated resource, with an access time smaller than or equal to l_{res}^{max} cycles, accessed by N_c cores, where l_{res}^{max} is the maximum delay it can take a request to be serviced by the resource. We will elaborate on this delay for the bus and the memory, respectively called l_{bus}^{max} and l_{mem}^{max}.

When core c_i, with $i \in \{1, .., N_c\}$, has the highest priority in a given round of arbitration, the priority ordering for the subsequent round becomes: $\{c_{i+1}, c_{i+2}, ..., c_{Nc}, c_1, c_2, ..., c_i\}$, where c_{i+1} becomes the core with the highest priority and c_i gets the lowest. *RoRo* is work conserving, or in other words a lower-priority requester can be granted access to the resource when all higher-priority requesters do not require it.

When all cores continuously issue access requests, the theoretical worst case is that any request r_i issued from the *scua* always has the lowest priority. We therefore have:

$$ubd_{RoRo} = (N_c - 1) \times l_{res}^{max} \tag{4.1}$$

Under a contention scheme of this type, both STA and MBTA can be applied to the *scua* in isolation (hence with no parallel contention) and then the worst-case contention overhead can be added compositionally by factoring in the above *ubd* to each access to shared resource.

Obviously however, the particular time alignment between the *scua*'s access and the circulation of the *RoRo* priority token across cores determines the contention delay actually suffered, so that the ubd_m may be significantly lower than the *ubd*. This is further discussed in Section 4.4.

FIFO. Consider now the same resource, this time with *FIFO* arbitration, accessed by N_c cores, where each core can have only up to one pending request in flight. FIFO assigns access priority in order of arrival, so that the requests arriving earlier to the arbiter get higher priority.

The theoretical worst case for the *scua* occurs when all cores have a pending request and, a request r_i from the *scua* becomes ready and it is preceded by $N_c - 1$ older requests from the other cores. This produces the same *ubd* as for *RoRo*:

$$ubd_{FIFO} = (N_c - 1) \times l_{res}^{max} = ubd_{RoRo} \tag{4.2}$$

However, by the time request r_i is issued, the oldest request at the top of the FIFO queue may have progressed to near completion, which – again – causes ubd_m to be substantially lower than *ubd*. We can thus observe that under both *RoRo* and *FIFO*, the worst case occurs contingent on a particular alignment between the *scua*'s request(s) and those of all other contending cores, and distinct for each arbitration policy.

4.1.2 Difficulties in Determining the ubd

When the internal workings of the processor cannot be known, the *ubd* cannot be determined analytically, but only approximated via ubd_m, as was the case in [18].

We noted earlier that designing observation experiments to maximize the impact that the interfered *scua*'s requests suffer from other cores (which is required to "observe" the *ubd*) is impaired by the need to control the alignment in time between the *scua*'s requests and those of the contending cores.

Consider N_c arbitrary software components, $SC = \{sc_1, sc_2, ..., sc_{N_c}\}$, one of which is our *scua*, with each sc_i pinned to a distinct core, and all contending access to a *RoRo*-arbitrated resource. It is evident that if we simply run all those programs together, with no other precaution, it would be highly unlikely that each and every *scua*'s request encountered worst-case contention. This is so because, when a request r_i from the *scua* is issued in the program run, its *RoRo* priority is not necessarily the lowest and therefore its wait time is less than ubd_{RoRo}. The case of *FIFO* arbitration is analogous, because it is equally unlikely that every single *scua* request is issued when all other cores have pending requests enqueued and none of them is already being served.

In principle, given a specific *scua*, one might possibly design *matching contenders* capable of issuing their access requests with the frequency needed to cause the *scua*'s requests to always be last in the queue and encounter *ubd* contention. However, it goes without saying that this effort would be utterly disproportionate, owing to its extreme sensitivity to the particular behavior of (the particular version of) the *scua* and, even worse, to its critical dependence on detailed knowledge of the inner workings of the resources of interest so that the desired timing of request generation can be well understood and fully controlled.

We can therefore maintain that soundly approximating the *ubd* with observation measurements that are sustainable for knowledge need and affordable for design and implementation costs is an open problem. Interestingly however, solving that problem would

be of great value to industrial users, as they would be provided with *scua*-independent test sets capable of causing ubd_m to be a sound approximation of *ubd*, which could thus be used as an additive factor to the WCET bound determined for the *scua* in isolation, with state-of-the-art single-core analysis techniques. This is the challenge we take in this Chapter.

4.2 Elements of the Proposed Solution

In this chapter we use as our reference architecture LEON4-based NGMP, which we described in section 2.1.1.

We also develop several resource stressing kernels, that were described in Chapter 2 but further down this chapter we will detail the specifics of the RSK used:

4.2.1 Resource Stressing Kernels

We first discuss the specialization of *rsk* for the processor resources of interest, and then we show that they fail to safely approximate the respective *ubd*. Subsequently we present a new methodology to do that.

```
1: for i = 0 to bus_Accesses/5 do
2:     ld [0x10000000], $0
3:     ld [0x10000400], $0
4:     ld [0x10000800], $0
5:     ld [0x10000C00], $0
6:     ld [0x10001000], $0
7: end for
```

(a) *bsk*

```
1: for i = 0 to bus_Accesses/5 do
2:     ld [0x10000000], $0
3:     ld [0x10010000], $0
4:     ld [0x10020000], $0
5:     ld [0x10030000], $0
6:     ld [0x10040000], $0
7: end for
```

(b) *msk*

FIGURE 4.1: Pseudo-code of *rsk* for the bus made with load operations(©2016 IEEE)

Bus. We call bus stressing kernel (*bsk*), the *rsk* dedicated to the bus. The *bsk* is designed to cause every instruction to miss in DL1 and hit in L2. This structure ensures a short turn-around time for memory requests, which keeps the bus busy as much as possible.

Given that DL1 uses LRU replacement, the *bsk* comprises a loop with $W + 1$ load instructions, where W is the number of DL1 cache ways (see Figure 4.1(a)). Those loads have a predefined stride among them so that they access the same DL1 set, thereby exceeding its capacity and systematically missing in DL1. Furthermore, the memory

addresses referenced by the *bsk* are designed to exactly fit in L2. In this way, all accesses miss in DL1 and hit in L2.

To hit in L2 we use load operations, which produce the highest bus contention. In the NGMP in fact, L2 hits hold the bus until the L2 serves the request, while L2 load misses are split transactions, which release the bus until memory sends the missed data, and store requests are immediately served, thus keeping the bus for a shorter duration.

Figure 4.1(a) presents the *bsk* for the NGMP: as the DL1 has 4 ways, the loop body of the *bsk* includes a stride of five instructions that all map to the same set.

Had the DL1 replacement policy been unknown, we would have designed the loop body to perform $N \gg W + 1$ distinct accesses to the same set, for an N that does not exceed the L2 capacity in the corresponding L2 set, to make it highly unlikely for memory operations to hit in DL1.

Memory controller. Analogously, we call *msk* the *rsk* dedicated to stress the memory controller. The *msk* design follows the same principles as for the *bsk*, except that the memory accesses in the *msk* have to yield L2 misses. The factors of influence to this end are the size of the way for DL1 and L2 (to cause L2 misses and therefore access the memory controller), and the size of the cache line (that determines the unit of transfer).

For the NGMP, we use a load stride of 64 KB, which is an integer multiple of the DL1 way size (4 KB). Hence, all memory accesses map to the same DL1 set. This is also the way size of the L2, hence memory accesses also map onto one and the same set. As L2 uses LRU replacement, every memory access made by the *msk* results in a miss. Figure 4.1(b) presents the pseudo-code of the *msk*.

4.3 The Synchrony Effect

Intuitively, one would expect that assigning specialized *rsk* to all cores contending with the *scua* should capture the worst-case contention scenario, and thus allow obtaining a trustworthy approximation of the relevant *ubd*.

As we show next however, this intuition is wrong in practice, because – when subject to heavy load conditions – both *FIFO* and *RoRo* incur a particular phenomenon that we term *synchrony effect*. The essence of this phenomenon is that, when all cores issue requests at a given constant rate to the resource of interest, their requests interleave in a particular way *systematically*, so that their interleaving becomes synchronous. In that situation, the resulting contention delay becomes constant and, more important, unlikely to match *ubd*.

TABLE 4.1: Main terms used in this chapter

Δ_{ET}	Exec. Time increment suffered by the SCUA
n_r	Number of accesses made by an SCUA
l_{res}^{max}	Max response time of one resource
N_c	Number of cores
c_i	Core i
R^x	All request done by a task X
r_i	Request i
δ_i	Injection time between r_i and r_{i-1}
γ_i	Contention delay by request i
et^{rsk}	Exec. time versus a rsk
et^{isol}	Exec. time in isolation
d_{bus}	Exec. time increment due to interferences on the bus

We now discuss the *synchrony effect* for the bus, which we obtain by using $N_c - 1$ *bsk* as the contenders to the *scua*, under both *FIFO* and *RoRo*. We use the main terms presented in Table 4.1.

4.3.1 Synchrony Effect under FIFO

The *synchrony-effect* causes the shared resource to behave as if it was multiplexed across all cores, with each core being assigned a time slot of duration equal to the service time of an individual request. Interestingly, this applies to both *FIFO* and *RoRo*. Let us now see that for *FIFO*.

The contention delay suffered by the *scua* for its request r_{i+1} depends on the time elapsed since its preceding request r_i and how r_{i+1} positions in the request queue.

Let us assume that the *scua* may issue multiple requests to the bus, which we denote $R^{scua} = \{r_0, r_1, ..., r_m\}$. Assume that those requests may be issued at arbitrary times, so that some time span intervenes between any two subsequent requests from the *scua*. Let us call injection time, denoted δ_i, the time span between the issue of requests r_{i-1} and r_i for any R. Accordingly, for R^{scua}, we have $\{\delta_1^{scua}, \delta_2^{scua}, ..., \delta_m^{scua}\}$.

In our reference architecture, δ_i corresponds to the time elapsed since the data loaded by r_{i-1} is sent back to DL1, until r_i is ready to access the bus. A minimum injection time δ_{min} separates any two subsequent requests from R, equal to the time it takes for DL1 and the core to process r_{i-1}, once it is served, and execute the instruction that generates r_i until it gets ready to access the bus.

When a program runs in parallel with other contenders, each of its request r_i may suffer a contention delay γ_i. Accordingly, for R^{scua}, we have $\{\gamma_1^{scua}, \gamma_2^{scua}, ..., \gamma_m^{scua}\}$.

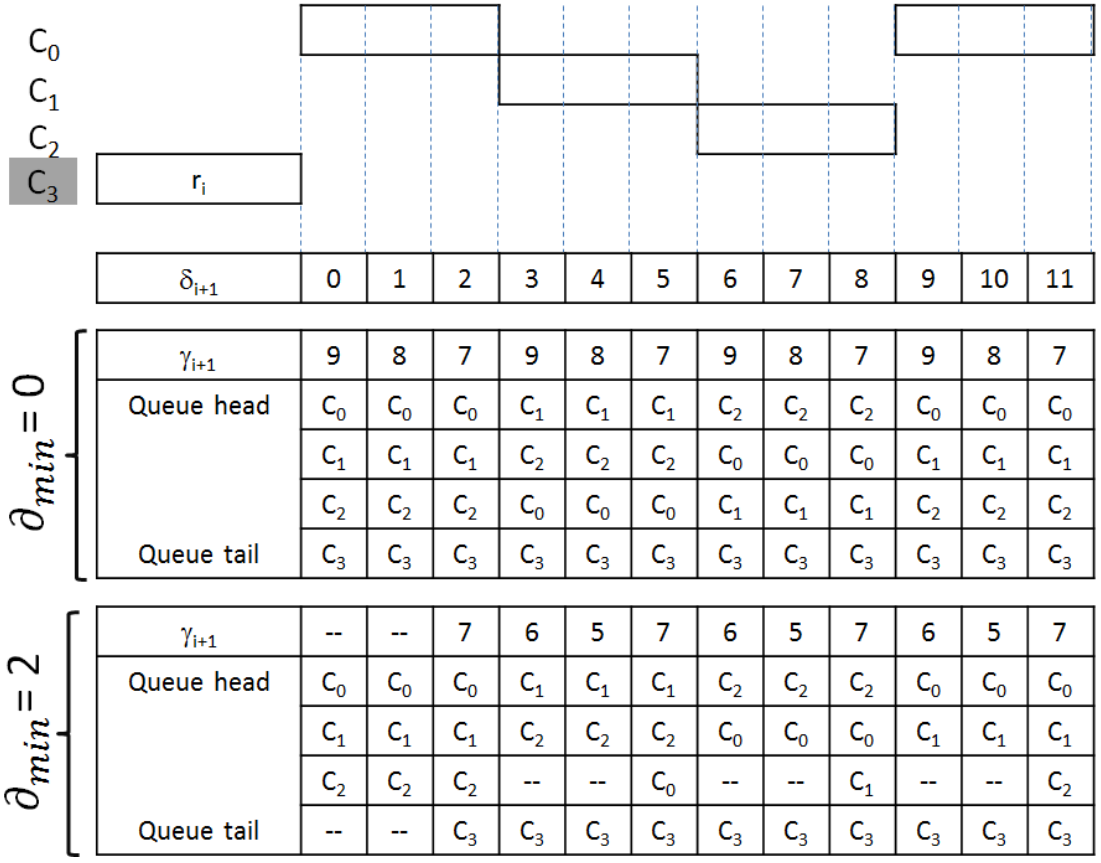


FIGURE 4.2: Contention delay γ as a function of δ (FIFO) for $\delta_{min} = 0$ and $\delta_{min} = 2$, respectively. In each cycle priorities are those at the start of the cycle, prior to arbitration(©2016 IEEE).

Since the *bsk* are designed to access the bus with high frequency, their requests have low injection time. In concept, the maximum contention scenario should occur for $\delta_{min} = 0$.

We now illustrate the synchrony effect under *FIFO* with an example where contenders are *bsk* and the *scua* can be either another *bsk* or any other software component. We explore two scenarios, with $\delta_{min} = 0$ and $\delta_{min} > 0$ respectively. The former, while infeasible in reality, serves for illustration.

Scenario $\delta_{min} = 0$: Let us assume that request r_i of the *scua* is just serviced and all other cores have pending requests enqueued. Figure 4.2 (rows $\delta_{min} = 0$) illustrates how γ_{i+1} varies as a function of δ_{i+1} (shown in the first row). For instance, if $\delta_{i+1} = 1$, then $\gamma_{i+1} = 8$ since r_{i+1} cannot be granted access to the bus until the ongoing request from c_0 is completed (which takes 2 more cycles) and requests from cores c_1 and c_2 are also serviced (which takes another $3+3 = 6$ cycles) since they are both already in the queue.

Assuming that each core can only have one pending request, the worst contention (*ubd*) occurs when r_{i+1} is delayed by the full service of $N_c - 1$ requests coming from the other $N_c - 1$ cores. In this example in Figure 4.2, this means $\gamma_{i+1} = 9$. When $\delta_{min} = 0$ and l_{bus} denotes the bus service time for an individual request, the synchrony effect manifests in

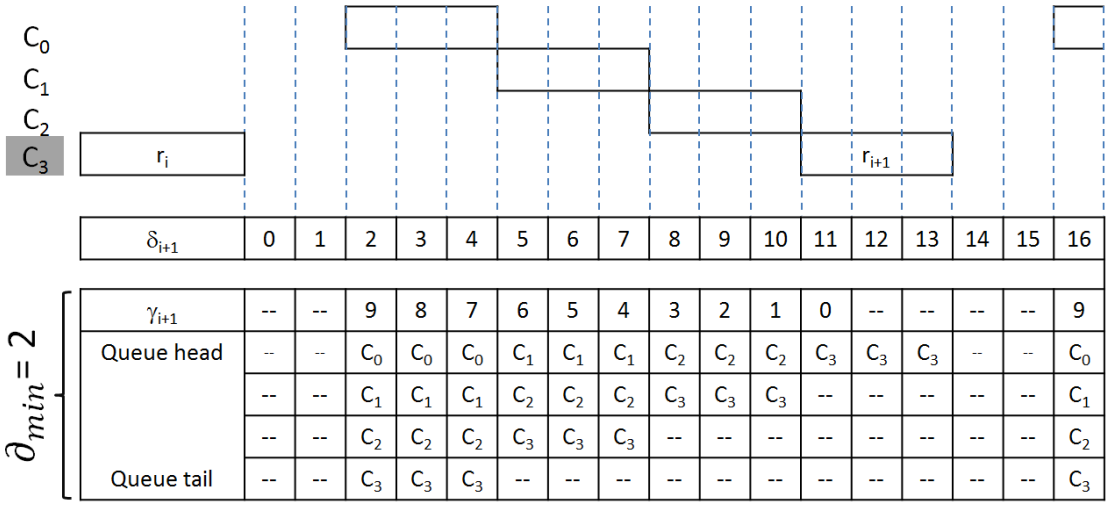


FIGURE 4.3: Example where contention delay γ is maximized for FIFO(©2016 IEEE).

the fact that γ_{i+1} has a periodic behavior that ranges from $(N_c - 2) \times l_{bus} + 1$ (when one contending request is near completion) to $(N_c - 1) \times l_{bus}$ (when all other contending requests are pending and none is being serviced). Thus, the particular value of δ_{i+1} determines the value of γ_{i+1}.

If δ_i^{scua} is arbitrary, it stands to reason that it is very unlikely that all requests $r_i^{scua} \in R^{scua}$ experience $\gamma_i^{scua} = ubd$. If for *scua* we use another *bsk*, which has $\delta_i = 0$ for all $r_i \in R$, as shown in Figure 4.2, $\gamma = ubd$ systematically.

Scenario $\delta_{min} > 0$: Owing to cache latency, the common case for the bus is $\delta_{min} > 0$. (For other farther-away off-core resources, such as the memory controller, $\delta_{min} \gg 0$.)

The bottom rows in Figure 4.2 show the impact on γ_{r+1} when $\delta_{min} = 2$. Right after r_i is serviced, γ_{r+1} would be equal to *ubd*. However, for 2 cycles r_{i+1} cannot reach the bus and thus $\delta_{r+1} \geq 2$. In particular, if $\delta_{r+1} = 2$, then c_0's request is already being processed at the time r_{i+1} is issued, hence $\gamma_{r+1} < ubd$. If $\delta = 3$, then c_0's request has been processed and its subsequent request will take at least 2 cycles to be issued and reach the queue. Thus, if $\delta = 3$, then $\gamma_{r+1} = 6$. Analogously, if $\delta_{r+1} = 4$, then $\gamma_{r+1} = 5$. If $\delta_{r+1} = 5$, then r_{i+1} finds the same scenario as for $\delta_{r+1} = 2$, with the only difference that the particular requests in the queue have different core owners, but for the same contention effect on r_{i+1}. Thus, if the *scua* executes against *bsk*, it cannot experience *ubd* contention regardless of whether the *scua* is a *bsk* or not.

In general, if the contending cores execute *bsk*, γ_i^{scua} for request $r_i \in R^{scua}$ can be described with the following equation, where $\delta \geq \delta_{min}$ holds:

$$\gamma_{FIFO}(\delta) = max\left(ubd - ((\delta - \delta_{min}) \mod l_{bus}) - \delta_{min}, 0\right) \tag{4.3}$$

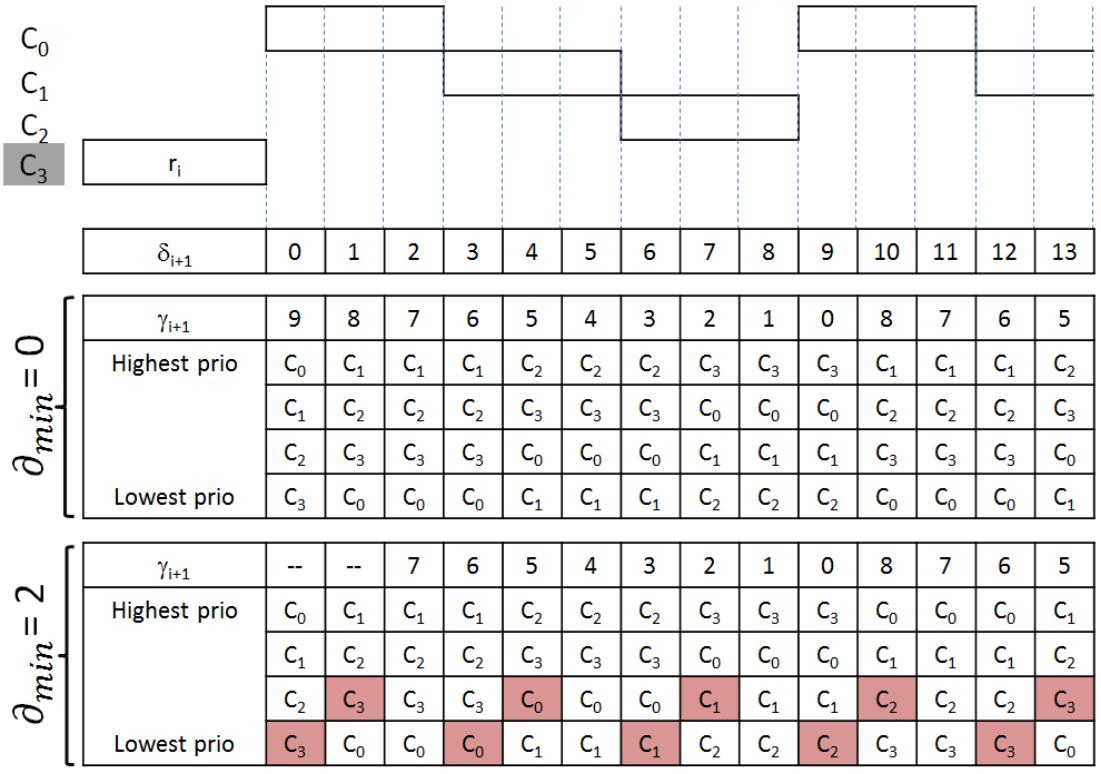


FIGURE 4.4: Contention delay γ as a function of δ (RoRo). In each cycle priorities are those at the start of the cycle, prior to arbitration. Shaded cells in the priority rows correspond to requests not in the queue(©2016 IEEE).

Note, however, that this does not mean that *ubd* cannot be experienced systematically. For instance, assume that the *scua* is a *bsk* and the contending cores execute programs that experience $\delta = 11$ in c_0, $\delta = 8$ in $c1$ and $\delta = 5$ in c_2, as shown in Figure 4.3. In this scenario, after r_i is serviced, the queue is empty for 2 cycles, and when $\delta = \delta_{min} = 2$, then r_{i+1} is issued and contends with requests from all other cores, which arrive simultaneously and are enqueued before it. All requests are processed in order and r_{i+1} experiences $\gamma = ubd$. Then, the queue is empty again for δ_{min} cycles until the same scenario for $\delta = 2$ repeats for $\delta = 16$. However, while this scenario could be hypothetically produced, it is very difficult – if at all possible – for a user to create programs with given δ values, which align in time properly, while ensuring that when requests arrive to the bus simultaneously, they are systematically enqueued in the desired way.

4.3.2 Synchrony Effect under RoRo

Under *RoRo*, the incoming requests are not necessarily served in order of arrival, but in the order determined by the round-robin assignment of access slots.

Again, we assume that *bsk* are run as contenders. If $\delta_{min} = 0$, all contenders always have a request pending in the queue. Thus, the only parameter that determines who is granted access to the bus is the current priority order. This is better illustrated in Figure 4.4 (see the $\delta_{min} = 0$ rows). As shown, c_0, c_1 and c_2 always have requests in the queue, either in service or still pending. Noticeably, r_{i+1} from c_3 becomes the highest priority request when $\delta_{r+1} = ubd = 9$. We also observe that $\gamma_{r+1} = ubd$ only when $\delta_{r+1} = 0$. Otherwise, γ_{r+1} traverses all values from $ubd - 1$ down to 0 consecutively in a round-robin fashion as δ_{r+1} increases.

Hence, if $\delta_{min} = 0$, running a *bsk* as *scua* would suffice to observe the highest contention consistently for all of its requests. However, as we noted before, the general case is $\delta_{min} > 0$, owing, for example, to the DL1 cache latency.

Figure 4.4 also shows the case for $\delta_{min} = 2$. In it, vacant positions in request the queue are marked with shaded cells in the priority rows.

In general, assuming $0 < \delta_{min} \leq ubd$ (as it is often the case in reality) so that 100% bus utilization can be reached, then γ stays *exactly the same* as if $\delta_{min} = 0$. This is so because δ_{min} only effects the contents of the request queue. Hence, r_{i+1} can only incur $\gamma_{r+1} < ubd$. Moreover, if δ is constant for all of the *scua*'s requests, then γ is also constant. This observation is of prominent importance in our methodology, as we discuss in the next section.

In the scenario where all contenders are *bsk*, γ can be described with the following equation:

$$\gamma_{RoRo}(\delta) = \begin{cases} ubd & \text{if } \delta = 0 \\ (ubd - (\delta \bmod ubd)) \bmod ubd & \text{otherwise} \end{cases} \tag{4.4}$$

In general, δ depends on δ_{min} and the particular *scua*. An arbitrary *scua* may observe different values of δ and so little can be concluded about the actual contention experienced. Alternatively, running a *bsk* as *scua*, we observe exactly $\gamma = ubd - \delta_{min}$ for all requests. In fact, it is hard to determine the actual value of δ_{min} even when cache latencies are known, since some pipeline stages may delay the access of DL1 misses to the bus. Thus, nothing can be concluded for certain about whether the highest contention has been observed or how far the observation is from the highest extreme. To tackle this issue, in next sections we propose a systematic methodology to determine *ubd* based on measurements. For that purpose we devise specific *rsk* that allow determining accurately *ubd* for round-robin and FIFO arbitrated shared resources including a shared bus and a shared memory controller.

4.4 Deriving the UBD for the bus

Taking stock of the *synchrony effect* discussed earlier, we now present a measurement-based method which computes a ubd_m that is guaranteed to be a safe approximation of *ubd* for hardware shared resources in COTS multicores.

In this section, we first describe the strategy we follow. Then we show how it can be implemented and applied in practice for the bus in our reference architecture, considering both *FIFO* and *RoRo* arbitration. Finally, we summarize some architectural issues of relevance.

4.4.1 Nop-based Methodology

As captured with Equations 4.3 and 4.4, when using *bsk* as contenders, the synchrony effect causes the amount of contention suffered by any request to be a function of δ.

We use that notion to construct a new *bsk*, which we call *bsk-nop* and illustrate in Figure 4.5 (a). In the *bsk-nop* we intersperse low-latency (*nop*) operations between the (load) instructions that access the bus. The effect of those *nop*s is to delay the injection time of each request to the bus, which modifies the δ value accordingly. Hence, whereas in the *bsk*, constituted of consecutive contending requests, we have $\delta = \delta_{min}$, if we add just one (for the sake of example) *nop* in between loads, we obtain $\delta = \delta_{min} + \delta^{nop}$, where δ^{nop} is the delay added by one *nop*.

```
1:  for i = 0 to bus_Accesses/5 do
2:      ld [0x10000000], $0
3:      nop
4:      ld [0x10000400], $0
5:      nop
6:      ld [0x10000800], $0
7:      nop
8:      ld [0x10000C00], $0
9:      nop
10:     ld [0x10001000], $0
11:     nop
12: end for
```

(a) *bsk-nop*

```
1:  for i = 0 to bus_Accesses/5 do
2:      ld [0x10000000], $0
3:      nop
4:      ld [0x10010000], $0
5:      nop
6:      ld [0x10020000], $0
7:      nop
8:      ld [0x10030000], $0
9:      nop
10:     ld [0x10040000], $0
11:     nop
12: end for
```

(b) *msk-nop*

FIGURE 4.5: Code of rsk_{nop} *implementations:* bsk-nop *and* msk-nop(©2016 IEEE)

By varying the number k of *nop* instructions inserted between load operations, each resulting bus request experiences a different δ_k. Figure 4.6 shows this effect for *FIFO*

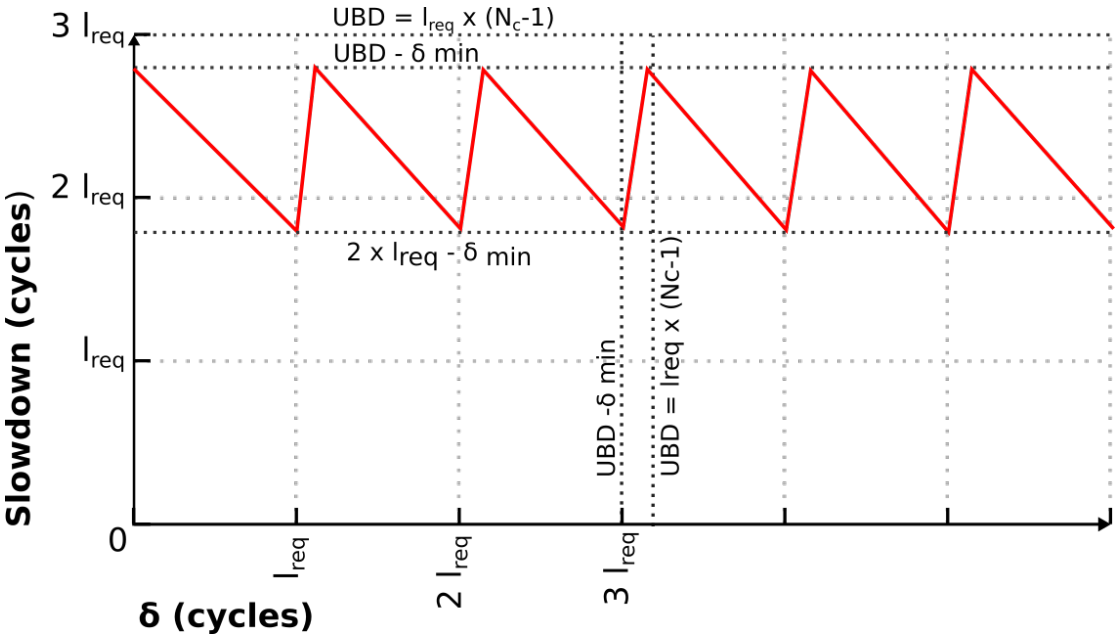


FIGURE 4.6: Saw-tooth behavior for *FIFO* with $\delta_{min} = 1$(©2016 IEEE).

with $\delta_{min} = 1$, which manifests as a saw-tooth profile. An analogous phenomenon occurs for *RoRo*, see Figure 4.8.

4.4.2 *bsk-nop* for *FIFO*

Figure 4.6 uses Equation 4.3 to plot γ as a function of $\delta_{min} = 1$. We see there that the values taken by $\gamma = ubd - \delta_{min}$ periodically repeat every l_{bus} cycles. This repetitive behavior reflects the fact that the requests issued by *bsk-nop* over l_{bus} cycles find decreasing contention load in the queue until a contending request issued by one *bsk* running in parallel on another core is queued again. The maximum contention delay experienced is $\gamma = ubd - \delta_{min}$, hence systematically inferior to *ubd*, since once a contending request is serviced, it takes δ_{min} cycles for a new request to be enqueued. At that time, contention is highest when the contenders are *bsk*, and amounts the theoretical worst case (*ubd*) minus the progress performed during δ_{min} cycles. We observe the saw-tooth shape in Figure 4.6, its period is equal to l_{bus} and the maximum of the function is $(N_c - 1) * l_{bus} - \delta_{min}$. In this case, *ubd* corresponds to $N_c - 1$ periods of the function. For instance, if we consider the example in Section 4.3.1 where $N_c = 4$, $\delta_{min} = 2$ and $l_{bus} = 3$, the saw-tooth will range between 7 and 5 cycles, and it will repeat every $l_{bus} = 3$ cycles. Thus, $ubd = l_{bus} \times (N_c - 1) = 9$. As shown, although we cannot observe the actual *ubd*, we can accurately infer it based on measurements with our methodology.

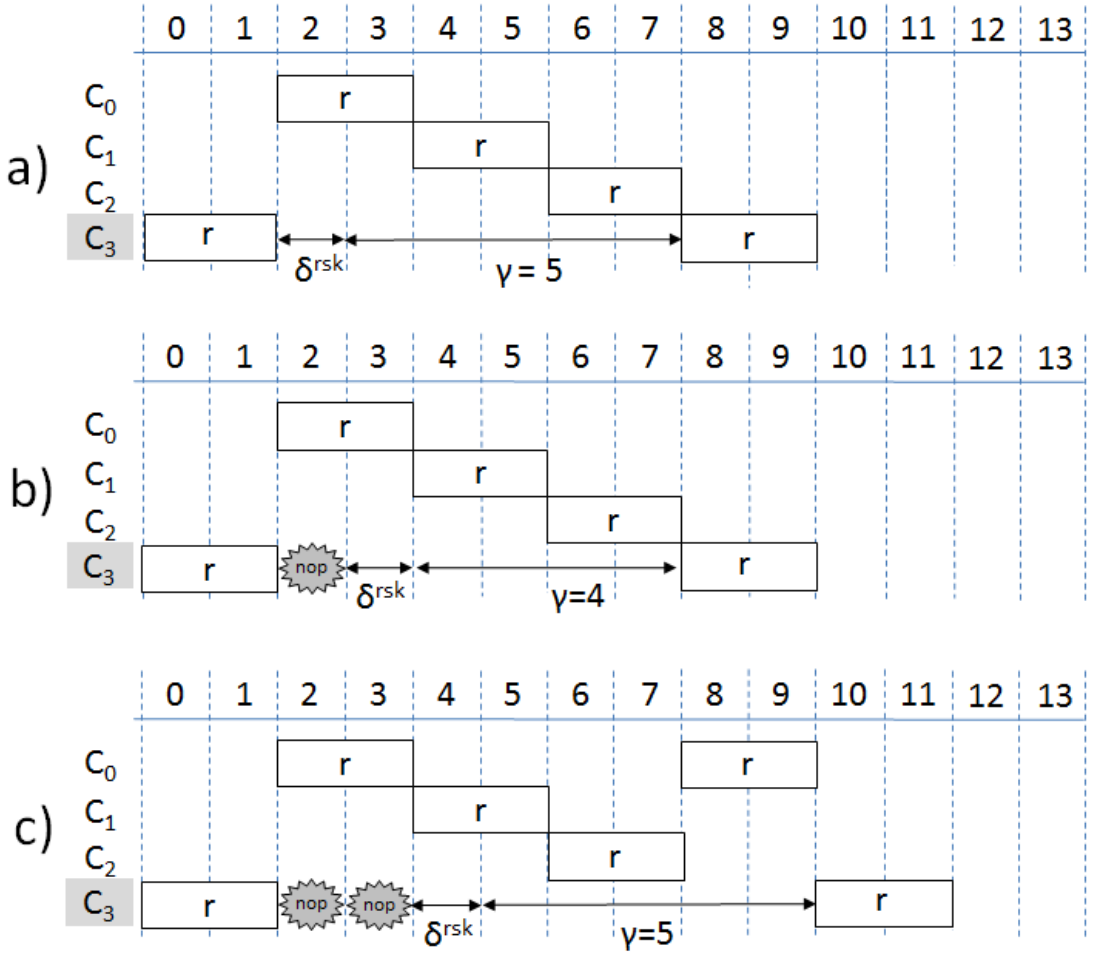


FIGURE 4.7: Timeline of the *FIFO* scenario for different k *nop* instructions: a) $k = 0$, b) $k = 1$, c) $k = 2$(©2016 IEEE).

Figure 4.7 illustrates this phenomenon, for $l_{bus} = 2$, $\delta^{rsk} = \delta_{min} = 1$, and an increasing number of inserted *nop*s, with $\delta^{nop} = 1$. We start from scenario a), where we assume $\delta^{rsk} = \delta_{min} = 1$ and we see that the request issued from core c_3, where the *scua* runs, suffers a contention of $\gamma(\delta^{rsk}) = 5$ cycles. In scenarios b) and c), we show the effect of increasing the number of *nop* instructions inserted between load operations in all contenders. In scenario b), we see that $\gamma(\delta^{rsk} + \delta^{nop}) = 4$, whereas in scenario c), core c_3 loses its turn for access to the bus, which increases its γ to 5 cycles again and shows the periodicity of γ as a function of l_{bus}, where $l_{bus} = 2 \times \delta^{nop}$ in this case. For higher *nop* counts, scenarios a), b) and c) repeat.

4.4.3 *bsk-nop* for *RoRo*

Figure 4.8 shows the variation in the contention delay incurred for *RoRo* as captured with Equation 4.4. The contention value reaches $ubd - 1$ at most, which – for $\delta_{min} > 0$ – occurs periodically at every ubd cycles.

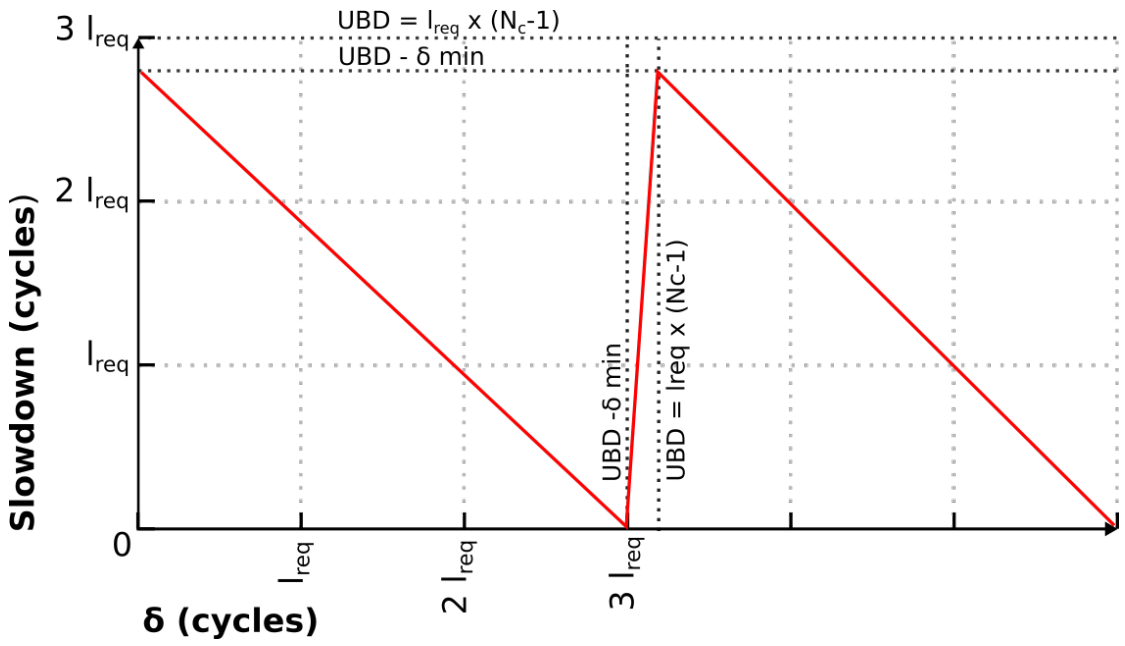


FIGURE 4.8: Saw-tooth behavior for *RoRo* with $\delta_{min} = 1$(©2016 IEEE).

This phenomenon is better illustrated in Figure 4.9, again for $l_{bus} = 2$, $\delta^{rsk} = \delta_{min} = 1$, and an increasing number of inserted *nop*s, with $\delta^{nop} = 1$. We start from scenario a), where the request issued from core c_3, where the *scua* runs, suffers a contention delay of $\gamma(\delta^{rsk}) = 5$ cycles. In scenarios b)-f), we show the effect of increasing the number of *nop* instructions inserted between load operations in all contenders. In scenario b), $\gamma(\delta^{rsk} + \delta^{nop})$ decreases down to 4. Through the scenarios c)-f), $\gamma(\delta)$ keeps decreasing as the number of *nop* instructions inserted, k, increases from 1 to 5. In scenario g), when $k = 6$, the situation becomes the same as in scenario a).

The following observations are now in order: (i) for $ubd \geq \delta_{min} > 0$, we have $\gamma \leq ubd - 1$, as per Equation 4.4; (ii) the variation of γ is periodic, with a period of ubd, *independent* of δ_{min}; and, more importantly, (iii) the exact value of ubd can be inferred from the period of $\gamma(\delta)$, as seen to vary with k: this holds true for any δ_{min} as long as $\delta_{min} \leq ubd$.

4.4.4 Applying the rsk-nop method

Our method to determine ubd requires carrying out several experiments using *rsk-nop* as *scua* and the normal *rsk* as contenders. *rsk-nop*(k) is parametrized by varying, incrementally, the number k of *nop* instructions inserted between the operations that access the bus.

We run *rsk-nop*(k) against $N_c - 1$ instances of *rsk*, recording its observed execution time, $et^{sc}_{scua}(k)$, and computing the increment from its execution in isolation, $et^{isol}_{rsk-nop}$.

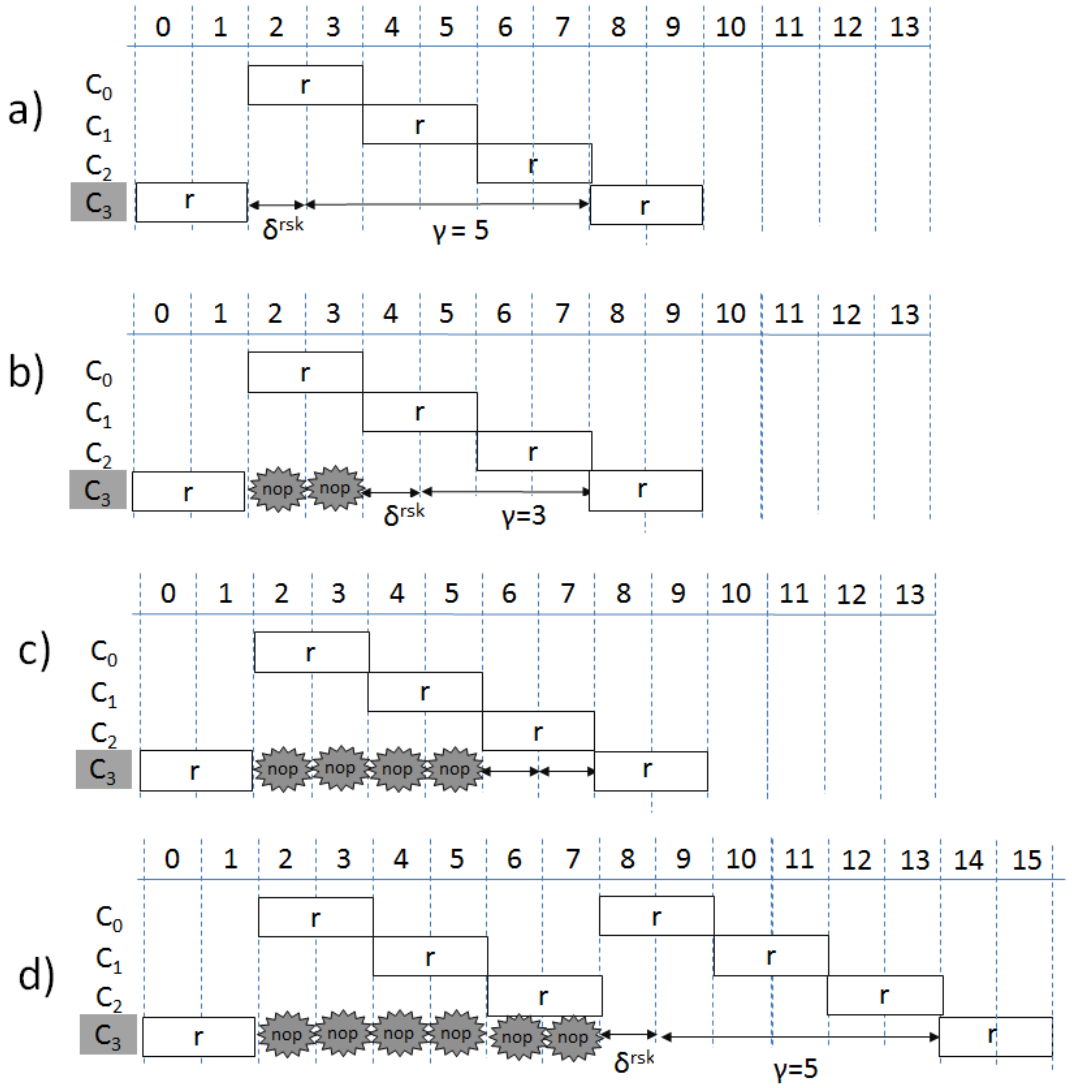


FIGURE 4.9: Timeline of the *RORO* scenario for different k *nop* instructions: a) $k = 0$, b) $k = 2$, c) $k = 4$, d) $k = 6$(©2016 IEEE).

$$d_{bus}(k) = et^{rsk}_{rsk-nop}(k) - et^{isol}_{rsk-nop} \tag{4.5}$$

Plotting the values of $d_{bus}(k)$ for a range of k, we see a saw-tooth behavior, with period *ubd*. Assume now that the extremes of that period correspond to k_i and k_j respectively. With that in mind, for *FIFO* we have:

$$\begin{aligned} ubd_{FIFO} =& (N_c - 1) \times l_{reqFIFO} \quad \text{where} \\ l_{reqFIFO} =& |k_i - k_j| : (k_i \neq k_j) \text{ and } (d_{bus}(k_i) = d_{bus}(k_j)) \end{aligned} \tag{4.6}$$

For RoRo, we have that *ubd* can be computed as the period of the resulting saw-tooth shape of d_{bus}.

$$ubd_{RoRo} = |k_i - k_j| : (k_i \neq k_j) \text{ and } (d_{bus}(k_i) = d_{bus}(k_j)) \tag{4.7}$$

4.4.5 Deriving l_{bus}^{max}

UBD depends on the number of rounds to wait to get access to the shared resource (i.e. Nc-1); and he maximum duration that a request to the resource can take (l_{bus}^{max}). In the measurement based approach proposed in our work, RSKs have to be designed to trigger l_{bus}^{max}. In our reference architecture, whether accesses to the bus are read/writes and hit/miss in L2 cache determines their latencies. In [76] it is empirically assessed that load hits to the L2 use the bus 9 cycles, load misses 7, while writes (whether or not they miss in L2) take 1 cycle. Hence, we factor in l_{bus}^{max} in our methodology by making that for the bus stressing kernel all memory operations are load hits to the L2.

4.4.6 Multicycle nop operation

So far we have assumed that $\delta_{nop} = 1$. This is indeed the case in most architectures since nop instructions do not have input/output dependencies and use the fast integer pipeline, if present. In the unlikely case that $\delta_{nop} > 1$, varying the number of *nop* instructions in the *scua* will be equivalent to sampling the saw-tooth behavior shown in Figures 4.6 and 4.8. If the value of δ_{nop} can be determined, then we can obtain the saw-tooth period easily. Otherwise, we infer δ_{nop} as follows: we use a *rsk* whose loop body solely includes k *nops* instructions, as many as possible without causing misses in the instruction cache; at that point, by dividing the observed execution time of that *rsk* by k, we derive δ_{nop} very accurately.

4.4.7 Summary

The method we have illustrated in this section empirically derives ubd_m, requiring little knowledge about the underlying architecture, which is often available in the corresponding public documentation. First, *Requirement*: We tested our approach for the bus, under *FIFO* and *RoRo* and we have shown it to work. Second, *Inputs*: Our approach requires knowing the type of instructions that may generate requests to the bus, which is typically documented in the processor's manuals. And third, *Confidence*: Two elements are central to confidence on the obtained ubd_m. On the one hand, $N_c - 1$ cores running a *rsk* should suffice to increase the bus utilization to 100%, also considering the handshaking overhead. In several processor architectures, performance monitoring counter support exists to measure the bus utilization. For instance, the memory mapped

registers at the addresses $0x17$ and $0x18$ in the Cobham Gaisler NGMP provide per-core and cumulative bus utilization counters [28]. On the other hand, we have provided a method for the user to derive δ_{nop}, which is needed to determine the saw-tooth period.

The derived bound, ubd_m, can be used by STA as *ubd* by adding it compositionally to the access time to the bus without contention [18]. With MBTA, the user must determine an upper bound n_r to the number of bus requests that the *scua* issues to the bus. The WCET bound of the *scua* is then padded with $pad = n_r \times ubd_m$.

4.5 UBD for the memory controller

In this section we show how to empirically derive *ubd* for the memory controller. In our reference architecture, the L2 forwards its misses to a request queue located in front of the memory controller. Each core has one entry in that request queue, which therefore has 4 positions. On an L2 miss, a split command is sent to the bus to stall the core that caused the miss, until the corresponding memory request has been served. In the meanwhile, the other cores can continue working. To determine which pending request accesses memory, the memory controller implements two arbitration policies, *FIFO* and *RoRo*, which we discuss below in isolation.

Before we do that, though, we must clarify an inner detail of consequence. Assume that, at a given point in time, the request queue is full, so that it contains N_c requests. Once one of those requests, r_i, has been served, two actions occur. First, a new request r_j from another core is granted access to memory. Second, the core that issued r_i (and has now resumed working) may miss again in L2 and therefore cause a request r'_i to be stored in the request queue of the memory controller. As an L2 access is faster than a memory access, it is fair to assume that, in general, r'_i gets stored in the request queue *before* r_j is served.

As a building block to our measurement-based analysis, we use the *msk* concept outlined in Figure 4.1(b). This kernel causes a continuous stream of misses in DL1 and in L2, with each such request going to memory. Following the same methodology as for the bus, we generate a variant of this kernel, called *msk-nop*, which inserts a variable number of *nop* instructions in between cache accesses (cf. Figure 4.5(b)).

4.5.1 *msk-nop* for *FIFO*

Using a *msk* as *scua*, under *FIFO* arbitration, we must consider that the time to serve a memory request is longer than the time it takes for a *msk* to reach memory with another

request r_{i+1} after its previous request has been served. When r_{i+1} reaches memory, it is preceded by exactly $N_c - 1$ pending requests, one for every other core, which all run *msk*. $N_c - 2$ of those requests are still to be enqueue awaiting service, whereas one of them has begun to be serviced for a duration that corresponds to the δ_{min} factor for memory. We can therefore see that this scenario is analogous to the one we have seen for the bus under *FIFO*, shown in Figure 4.2 for $\delta_{min} > 0$. The extent of contention captured in that case is high, but not enough to observe *ubd*.

Using *msk-nop* as *scua* allows us to explore a range of γ whose period extends through l_{mem}. During that duration, the number of pending requests that precedes r_{i+1} is exactly $N_c - 1$ for $l_{mem} - \delta_{min}$ cycles, and $N_c - 2$ for δ_{min} cycles. Plotting the observed γ as a function of the nop instructions inserted in the *msk-nop* used as *scua*, we would see the exact same shape as shown in Figure 4.6, except with a different scale.

4.5.2 *msk-nop* for *RoRo*

Analogously to the case of *FIFO* arbitration, if we use a *msk* as *scua*, whenever a request r_{i+1} reaches memory after its previous request has been served, it is preceded by exactly $N_c - 1$ requests. One of those pending requests has begun to be serviced for δ_{min} cycles: this means $\gamma = ubd - \delta_{min}$. We can therefore see that this scenario is analogous to what we saw for the bus with *RoRo*, as shown in Figure 4.4 $\delta_{min} > 0$. Once again, *ubd* contention is not empirically observed.

Using *msk-nop* as *scua*, we obtain the "sawtooth plot" depicted in Figure 4.8, in which γ ranges between $ubd - 1$ and 0, which allows us to derive *ubd* for the memory controller analogously to what we do for the bus under *RoRo*.

4.5.3 Deriving l_{mem}^{max}

As for the bus, the duration of each request to the main memory can vary, which requires deriving l_{mem}^{max}. In general for DRAMs, the duration of a request depends on [73, 77]: i) the memory mapping scheme – that defines the mapping of physical addresses from the processors to the actual memory blocks in the memory devices; ii) the row-buffer policy; the type of the request; and the type of its predecessor request. That is the latency of a request is a combination of the type, DRAM page, bank and rank it accesses and the same parameter for the previous request. For instance, a given request type (e.g. read) take typically shorter when the previous requests is of the same type, i.e. Read-After-Read (or Write-After-Write) than otherwise, which is also affected to whether accesses

go to the same bank and rank. As another example, access to open pages (i.e. hitting in the row-buffer) take shorter than to close pages that have to be loaded in the row-buffer.

Those effects have been conveniently studied in the literature [73, 77] and are typically well documented in DRAM specifications [78–80] as opposed to the timing information of COTS processors. Based on this information memory-stressing kernels (msk) an be designed to force request to take l_{mem}^{max}. The only change required is to design specific msk that alternate different types of operations. The address of the accesses are to be properly set so those operations are forced to access the desired bank and rank, depending on the memory row-buffer managing policy and the memory mapping scheme.

4.5.4 Memory refresh

An intuitive solution consist in factoring refresh delay in the computation UBD. This solution effectively considers that every single request is affected by a refresh operation. However, this approach is too pessimistic. With measurement-based approaches the execution time observations taken on the real platform already factor in the impact of refreshes. Depending on how measurements are aligned with refresh periods, the number of refreshes that can affect the execution time can be one more than those observed, so it is enough to pad the observed execution time with t_{RFC}.

It is also the case that a task execution time is increased by Δ_{cont} to capture in the impact of contention on the bus and the memory (excluding refreshes). The number of refresh operations occurring during Δ_{cont} can be easily computed by the following recurring equation (fixed-point iteration): $N_{REF}^{(k+1)} = \lceil (\Delta_{cont} + N_{REF}^{k} \times t_{RFC})/t_{REFI} \rceil$, where Δ_{cont} is the extra time the task execute due to contention, without considering the impact of refreshes; t_{REFI} the period at which refresh command (REF) are sent to all banks, and t_{RFC} the number of cycles a refresh command takes to be completed. The recursion terminates for the value of k such that $N_{REF}^{(k)} = N_{REF}^{(k+1)}$.

Overall, the impact of refreshed can be easily factored in by padding the WCET, not being required to capture it in the computation of UBD. The padding value is given by $(1 + N_{REF}^{kf}) \times t_{RFC}$ for the kf for which convergence is achieved.

Side effects of bus contention. When deriving *ubd* for memory, accesses may also compete for the bus, thus creating some interference. In general however, bus contention is generally much lower than memory contention, so that the former cannot mask the latter. Moreover, owing to the synchrony effect, the set of *msk* issue requests with a given (constant) frequency. Hence, if memory requests as served with TDMA with full bandwidth utilization, the corresponding bus requests are served in the bus with

analogous frequency, but with lower occupancy. For instance, if $l_{mem} = 10$ cycles and $l_{bus} = 2$ cycles, then we could have memory requests served in cycles 2-11 (core c_0), 12-21 (c_1), 22-31 (c_2), and so on, and bus requests in cycles 0-1 (c_0), 10-11 (c_1), 20-21 (c_2), and so on.

However, when using *msk-nop* as *scua*, the requests from c_0 will get issued later progressively until they collide in the bus with requests from c_1. Under *RoRo* arbitration, this collision is not an issue since which request from them is granted access first in the bus has no impact on memory contention as long as both of them reach memory before the corresponding core becomes the highest priority contender in memory. Under *FIFO* arbitration instead, if both requests are issued to the bus at exactly the same cycle, whether one or the other gets granted first may invert the order of requests in memory during one cycle with respect to the expected behavior. However, as long as hardware is deterministic and always solves these cases in the same way (e.g., granting access to c_0), the shape of the plots will remain as in Figure 4.6, and our approach to derive *ubd* will continue to work correctly.

4.6 Evaluation

We first present our experimental set-up. Then Section 4.6.2 (also Section 4.6.3 using stores instead of loads) and 4.6.4 show how *rsk-nop* allows deriving *ubd* in the presence of the synchrony effect. In that narration, we assume knowing the bus and memory controller latency as well as the actual value of *ubd*. This information is instead assumed unknown in Section 4.6.5, which demonstrates the applicability of our methodology to a real COTS multicore.

4.6.1 Experimental Setup

We model a 4-core NGMP [81] running at 200 MHz comprising a bus that connects cores to the L2 cache and an on-chip memory controller, see Figure 2.2. As explained in Chapter 2, Each core has its own private instruction (IL1) and data (DL1) caches. IL1 and DL1 are 16 KB, 4-way with 32-byte lines. The shared second level (L2) cache is split among cores, with each core receiving one way of the 256 KB 4-way L2. Hence, contention only happens on the bus and the memory controller. DL1 is write-through and all caches use LRU replacement policy. We model the worst-behavior of a closed-page 2-GB one-rank DDR2-667 [79] with 4 banks, burst of 4 transfers, and a 64-bit bus that provides 32 bytes per access, i.e., a cache line. Its longest latency across requests of any type is 27 cycles in our setup.

As illustrated in the Chapter 2, we use the EEMBC Autobench suite [30], which includes some real-world automotive software functions. We also use the *bsk, msk, bsk-nop* and *msk-nop* concepts presented earlier in this work, which use load operations to access the bus.

4.6.2 Synchrony Effect on the Bus

In order to show the robustness of the proposed methodology we evaluate it in the *reference architecture* as presented above, as well as a in *variant architecture* (labelled as *ref* and *var* respectively in following figures). In the latter, we change DL1 and IL1 access latency to 4 cycles (instead of 1 cycle). This variation increases the minimum injection time (δ_{min}) of all bus-access instructions by 3 cycles.

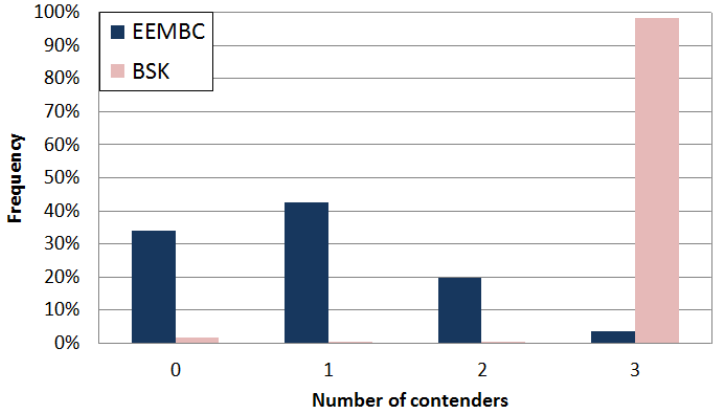
For the purpose of showing how *rsk-nop* allows safely approximating *ubd* from ubd_m, we use the following timing information of both the *ref* and *var* architectures. A given request suffers maximum contention latency of $l_{bus} = 9$ cycles per contender: 6 cycles corresponding to the L2 hit latency, and 3 cycles for bus transfer and arbitration handover. As a result, we have $ubd = 27$ cycles for the bus, following Equation 4.1.

In a first experiment, we run eight randomly generated 4-task workloads with EEMBC benchmarks under the *ref* architecture. The workloads are itemized in Table 4.2.

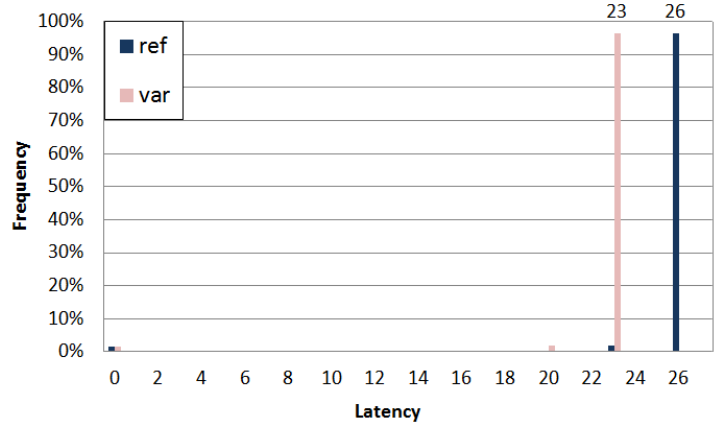
TABLE 4.2: Randomly-generated workloads used for evaluation

number	benchmarks
1	`cacheb`, `puwmod`, `canrdr`, `rspeed`
2	`iirflt`, `cacheb`, `puwmod`, `canrdr`
3	`ttsprk`, `iirflt`, `cacheb`, `puwmod`
4	`aifirf`, `ttsprk`, `iirflt`, `cacheb`
5	`tblook`, `aifirf`, `ttsprk`, `iirflt`
6	`a2time`, `tblook`, `aifirf`, `ttsprk`
7	`basefp`, `a2time`, `tblook`, `aifirf`
8	`pntrch`, `basefp`, `a2time`, `tblook`

Figure 4.10(a) presents the histogram of the number of contenders ready to send a request when the EEMBC benchmark in core c_0 requests the bus to start a transaction under *FIFO* (results for *RoRo* are analogous). Results for different workloads are quite similar. Most of the times, the requests issued by the EEMBC benchmark in c_0 find the bus empty or with just one contender. Only occasionally, the EEMBC in c_0 crosses ways with 2 or 3 contenders. This provides empirical evidence that with real application workloads it is very difficult to incur scenarios in which the number of contending requests is the highest possible value. As workloads or time-alignments results may vary, in fact, no a-priori guarantees can be provided that requests align in the worst possible way.



(a) Histogram of contenders



(b) Histogram of access latency for *bsk*.

FIGURE 4.10: Results for the bus for FIFO(©2016 IEEE)

Incidentally, while for *FIFO* all contenders will be served first at some point, in the case of *RoRo* the particular state of the priority assignment determines whether those contenders will be served before or after c_0.

In a second experiment we run 4 *bsk* that constantly access the bus. In this case, (see the pink (light grey) bars in Figure 4.10(a)), we observe that on almost every arbitration round the number of contenders is $N_c - 1 = 3$. Hence, the *bsk* reach their goal of causing maximum contention load on the bus. *However, owing to the synchrony effect, this ability is not sufficient to ensure that each* scua*'s request incurs a ubd.* As we have seen earlier, in fact, when $\delta_{min} > 0$ for both *FIFO* and *RoRo*, the actual contention is always inferior to *ubd*.

This experiment, in which we run 4 *bsk*, allows us to analyze this phenomenon in more detail, by measuring the actual contention delay γ_i that each individual request issued by c_0 suffers. Figure 4.10(b) shows the histogram of γ under the reference and the variant architecture. Results for *FIFO* (shown in the figure) and *RoRo* (not shown in the figure) are practically identical. We observe that the synchrony effect causes almost all requests

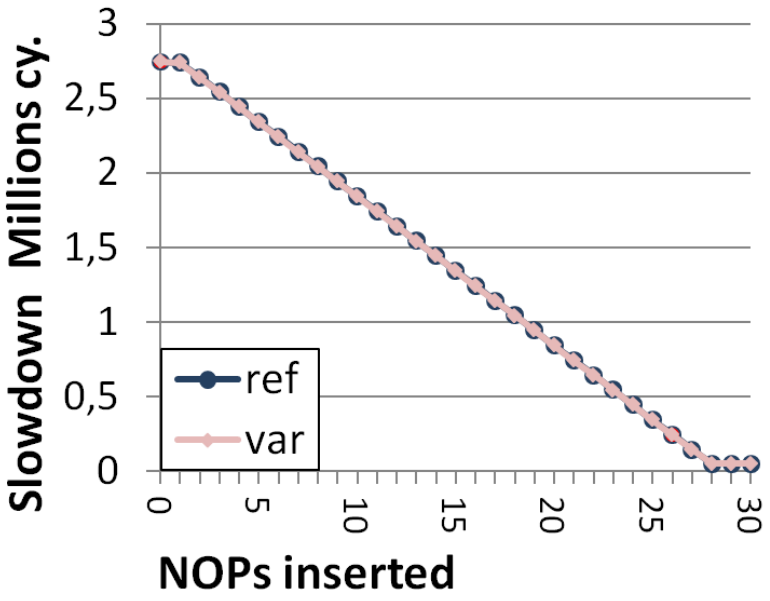


FIGURE 4.11: Slowdown when executed *rsk-nop* as *scua* against 3 *rsk* co-runners. Results shown as a function of nop instructions(©2016 IEEE).

in each case to incur the same latency, since the injection time among requests is the same. Further, we observe that the distance among ubd_m (observed) and the actual *ubd* (27 cycles in this case) varies across the two architectures: ubd_m is 23 for the *var* architecture and 26 for the *ref* one. This shows that the approximation quality of ubd_m varies as a function of the δ_{min} of the underlying architecture, which in turn does not permit to use *bsk* to arrive at a safe approximation of *ubd*. As we saw earlier in fact, $ubd_m = ubd - \delta_{min}$ when $0 < \delta_{min} < l_{bus}$.

The 2% requests with different ubd_m correspond to the requests executed until all *bsk* get synchronized and those requests at the beginning of the loop, due to the effect of loop control instructions.

We may therefore conclude that, in the general case, when the details about the latency of the bus are unknown, the use of *bsk* does not allow estimating *ubd* accurately enough.

4.6.3 Using store operations instead of loads

So far we have used load operations in the *rsk* and *rsk-nop*. We can also use stores, having in mind that our reference architecture has a store buffer that keeps store requests and allows instructions to proceed in the pipeline unless the buffer is full, i.e. a store request is considered completed as soon as it is put in the buffer. The requests in the buffer access the bus with an injection time $\delta = 0$ since once the buffer is filled, requests can be issued in consecutive cycles. In a high occupation scenario of the buffer, store requests suffer *ubd* in our scenario, i.e., one entry of the buffer is freed every *ubd* cycles. As δ

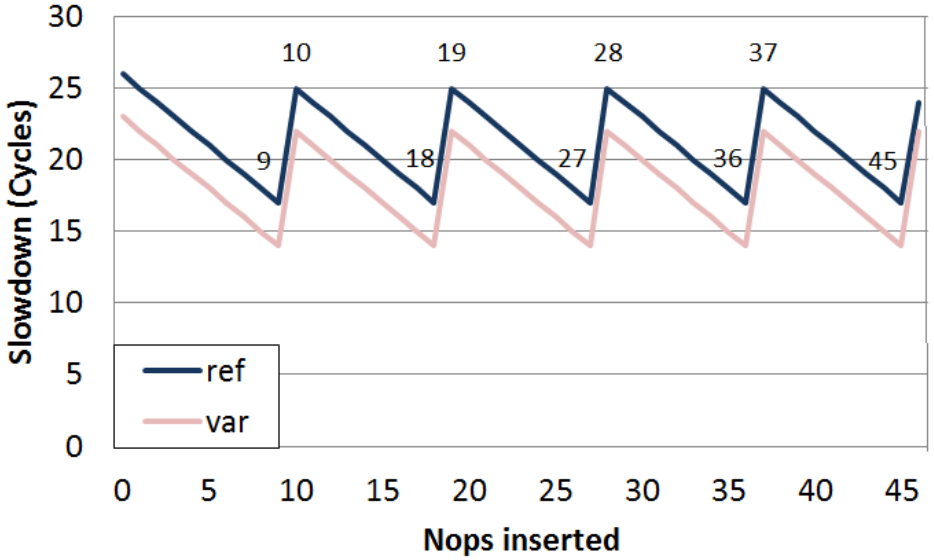


FIGURE 4.12: Slowdown when executing *bsk-nop* as *scua* against 3 *bsk* co-runners with *FIFO*(©2016 IEEE).

increases (by inserting *nop* operations), the slowdown in the rsk-nop corresponds to the difference between the latency of a new empty slot in the buffer, i.e. *ubd*, and δ. When δ is higher than *ubd* the buffer is able to allocate an empty slot before a new request comes, thus the slowdown suffered is always zero because the buffer is effectively hiding the store latency. As it can be seen in Figure 4.11, this causes that for one entire period the slowdown has a saw-tooth shape, while for following periods, the slowdown is zero. We observe that the first period spans from $k \in [1, .., 28]$, whose length matches the ubd. The one cycle shift in k is caused by the number of entries in the store buffer and its processing time.

4.6.4 Synchrony Effect on the Memory

The same conclusions presented in the previous section for the bus, also hold for the memory controller: A request suffers a maximum contention of 23 cycles. Hence, $ubd = (N_c - 1) \times 23 = 69$ cycles.

Our results confirm that: i) using three *msk*, one per core, suffices to cause that more than 98% of the times any request issued by the *scua* finds 3 pending contending requests enqueued at the memory controller. ii) in spite of that, neither in the reference architecture nor in the variant one, ubd_m is smaller than 69 cycles.

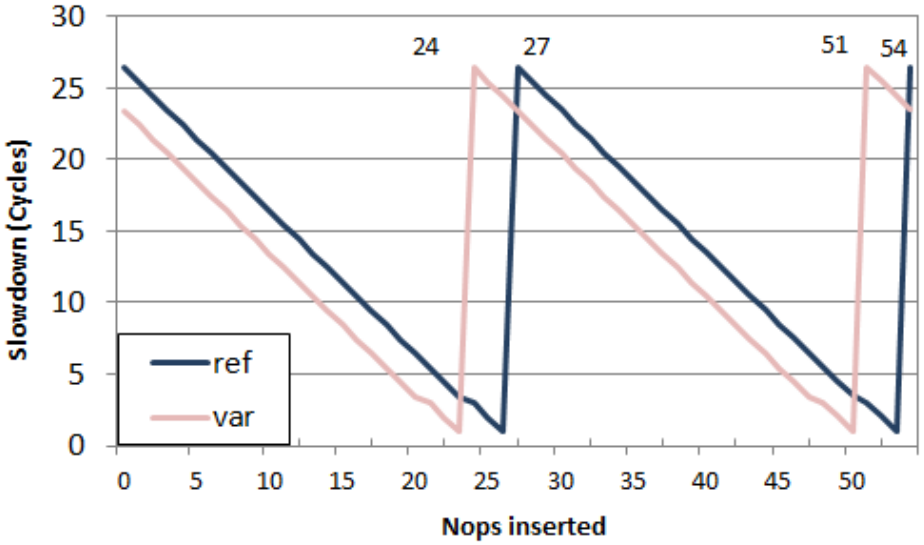


FIGURE 4.13: Slowdown when executing *bsk-nop* as *scua* against 3 *bsk* co-runners with *RoRo*(©2016 IEEE).

4.6.5 Evaluation of bsk-nop methodology for the bus

For the evaluation of the bsk-nop methodology, for FIFO and RoRo, we assume that no latency information is known.

FIFO: As shown in Section 4.4.1, to infer *ubd*, the injection time can be varied by inserting *nop* instructions between consecutive accesses of the *rsk* used as *scua*.

The Y-axis in Figure 4.12 shows the slowdown (in millions of cycles) suffered by *bsk-nop* with respect to its execution in isolation and the horizontal axis represents the variation of γ as a function of the number of nop instructions inserted.

We observe that results match those in Figure 4.6: the period of each sawtooth is 9 cycles, which corresponds to l_{bus}. As discussed in Section 4.4.1, however, we have to take into account $N_c - 1$ periods. For instance, from the first peak (cycle 10) until the forth one (cycle 37), the difference is exactly $ubd = 37 - 10 = 27$ cycles. Notably, the results for the *ref* and *var* architectures are exactly the same, but the absolute contention value decreases as δ_{min} increases.

RoRo: Figure 4.13 shows the result of the same experiment when the bus uses *RoRo*. As predicted in Figure 4.8, the slowdown is sawtooth-shaped, with period $ubd = 51-24 = 27$ cycles for *var*, and $ubd = 54-27 = 27$ cycles for *ref*. Hence, the period of the sawtooth is the same for both architectures, which proves the robustness of our method in inferring the *ubd* under different processor arrangements.

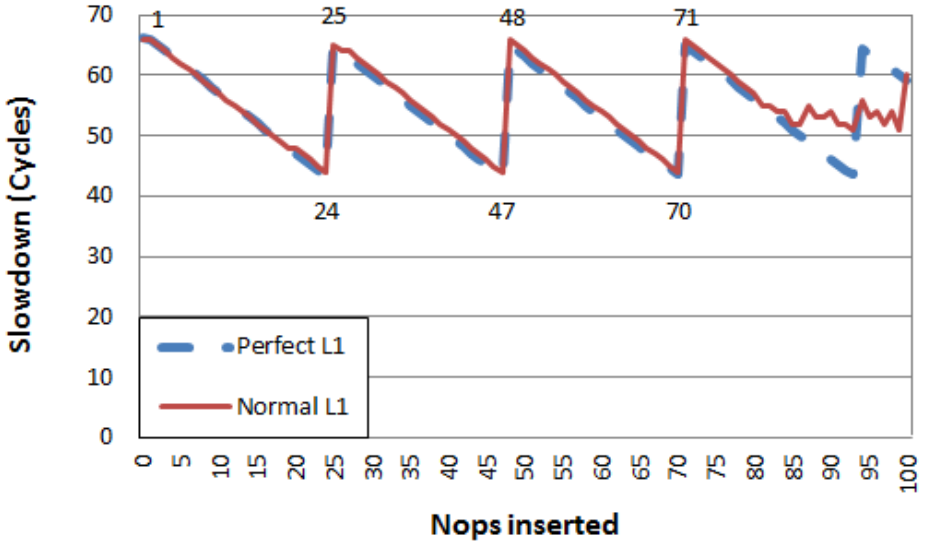


FIGURE 4.14: msk-nop methodology for FIFO(©2016 IEEE).

4.6.6 Evaluation of msk-nop methodology for the memory

We now repeat the same experiment as for the bus, by injecting nop instructions in the *msk-nop* used as *scua*. Since *ref* and *var* yield analogous results again, we only report those we obtained for the *ref* architecture.

FIFO: The vertical axis in Figure 4.14 shows the slowdown in cycles, compared with execution in isolation. The horizontal axis shows the number of *nop* operations inserted between memory accesses as shown in Figure 4.5(b) for the msk. We can observe the same sawtooth shape as in Figure 4.12, but with larger scale. The shape reaches its maximum with a period of 23 cycles when 2, 25, 48 and 71, ... *nop* instructions inserted, which means $l_{mem} = 25 - 2 = 23$, as expected.

After the 71th experiment, the results sop following the sawtooth shape. We studied why that happens and concluded that at that point, the number of nop instructions in the loop is large enough to exceed the IL1 capacity, so that IL1 misses occur at each iteration. In order to confirm this observation, we repeat the experiment with a processor set-up in our simulator that comprises a perfect IL1, i.e. an IL1 in which all accesses are hits. This is shown as "L1 perfect" in Figure 4.14: we observe that execution times follow the sawtooth shape is confirming our hypothesis in the increase in the number of conflicts in IL1. In order to solve this problem we propose the following approach.

Instruction cache-aware msk-nop methodology. The msk-nop methodology starts by adding a given number of memory accessing operations (load operations) in the main loop. This number is usually high to reduce the overhead (in relative terms) of the loop control applications, see Figure 4.1. In the *msk* used for the experiments in previous

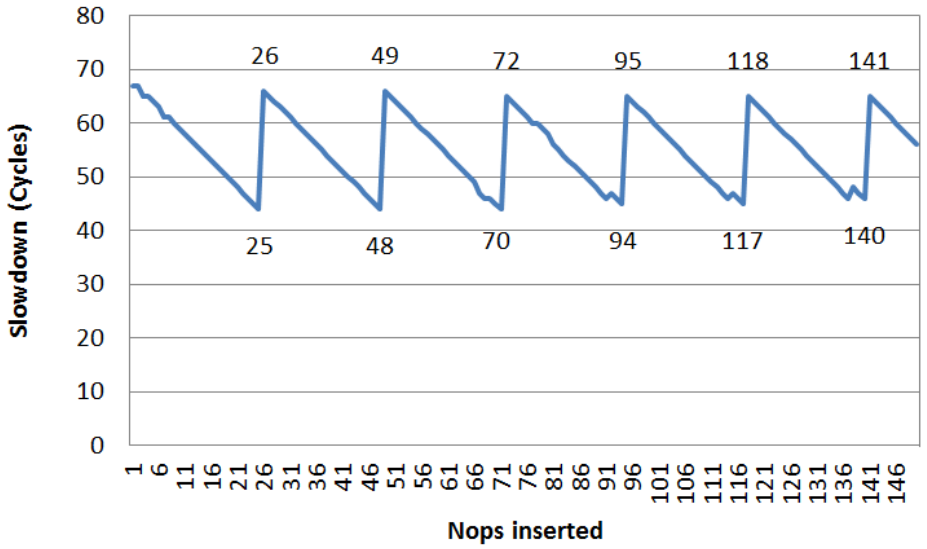


FIGURE 4.15: msk-nop inst.-cache aware methodology for FIFO(©2016 IEEE).

section, 50 load operations were included in the loop body, whose size therefore is around 200 bytes. When we add one nop instruction in between successive loads, the loop body doubles in size. When the number of nop instructions between loads reaches 80, the size grows to $(50 \times 80) \times 4 = 16,000$ bytes, which equals the IL1 size. As shown in Figure 4.12, the results start degrading just past that number of nop instructions.

To test the impact of having more than 80 nop operations between load operations, we simply reduce the number of load operations in the loop body such that its size, taking into account the size of load operations and the nops between them, does not go beyond the instruction cache size (16KB in this case). For instance, we put 50 loads in the loop for all experiments below 80 *nop* instructions. Then, we reduce the load count until 40 for all experiment until 100 *nop* instructions, and so on and so forth not to exceed IL1 capacity.

With the new experiment we can corroborate that $ubd_m = (49 - 26) \times 3 = 69$ cycles, so $ubd_m = ubd$.

RoRo: Figure 4.16 shows the results for *RoRo* with the original *msk-nop* that can exceed IL1 cache size. As for *FIFO* the shape degrades beyond 80 *nop* instructions, with the difference that in this case deriving *ubd* may not be possible if we do not fix our *msk-nop* methodology. Again, when making IL1 perfect the sawtooth shape is obtained as expected, so we apply exactly the same solution as for *FIFO*: keeping loop size below IL1 cache size at all times. We do so with the experiment in Figure 4.17, where we observe that the distance between two teeth of the plot is exactly $udb_m = 136 - 67 = 69$ cycles, so $ubd_m = ubd$.

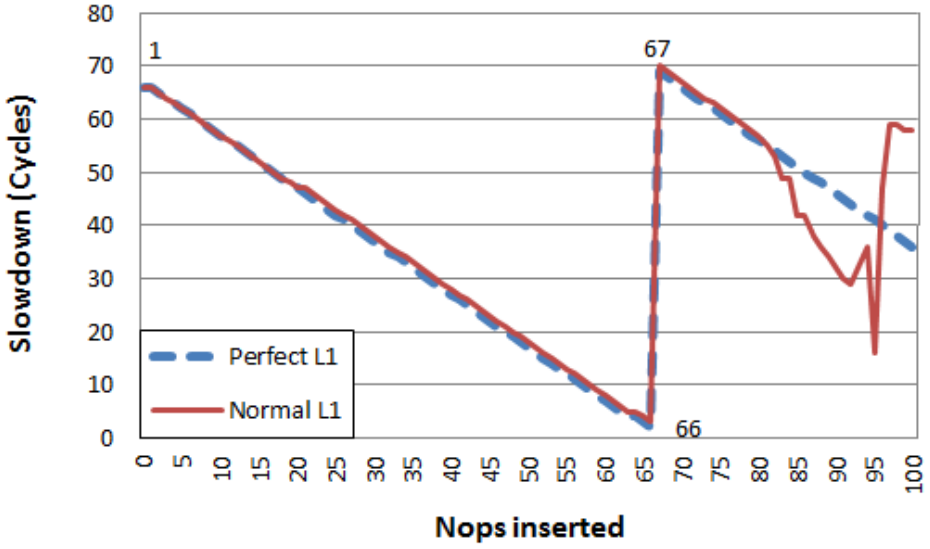


FIGURE 4.16: msk-nop methodology for RoRo(©2016 IEEE)

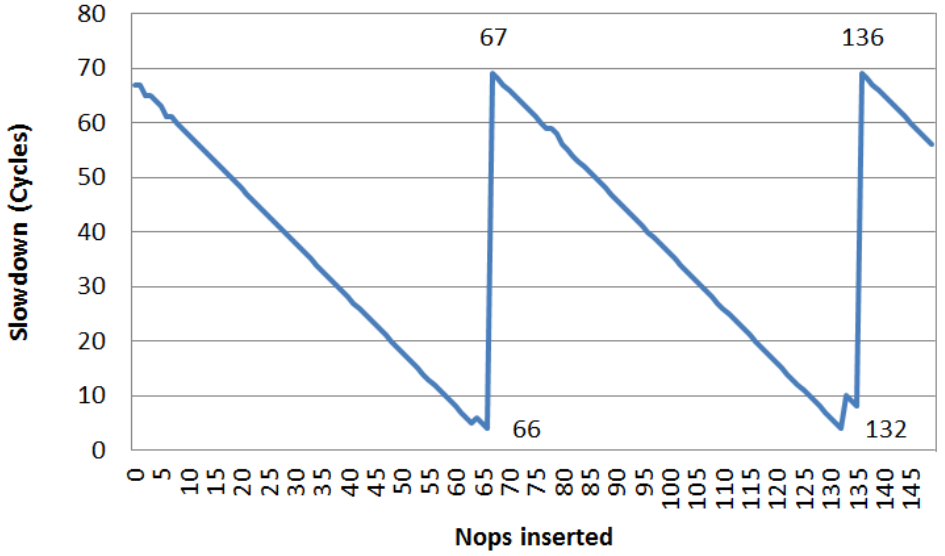


FIGURE 4.17: msk-nop inst.-cache aware methodology for RoRo(©2016 IEEE)

Overall, the proposed *msk-nop* instruction-cache aware methodology manages to derive the actual *ubd* value for both FIFO and RoRo under FIFO and RoRo arbitration policies.

4.6.7 Summary

As shown, our methodology based on injecting *nop* operations in the corresponding *rsk* allows deriving the sawtooth shapes needed to derive *ubd* for both *FIFO* and *RoRo* arbitration policies. Differences in the shape across resources (bus and memory) only affect the scale of the plots, but not their interpretation. Also, we have observed that it is critically important keeping the size of the *rsk* small enough to fit in IL1 since, otherwise, IL1 misses corrupt results and our methodology is no longer usable.

4.7 Related Work

Resource-stressing kernels [31] have been proposed to characterize the contention on certain resources of a multithreaded architecture. They are also used in [26] to characterize the NGMP [81] or in [17] to characterize the Freescale P4080.

Authors in [82] analyze the impact of resource sharing in multicore and criticize the confidence that one can obtain with *rsk*. We acknowledge the need to increase the confidence on the results provided with *rsk*, which in fact is the focus of this chapter by proposing *rsk-nop*-based methodology.

In [83] authors report a counter intuitive behavior with a RoRo based multicore: the execution time of a task running against a given number of cores can be smaller than its execution time when running against fewer number of cores. Our work identifies the reasons behind this counter intuitive behavior, namely the synchrony effect behavior, and takes advantage of it to derive the *ubd*.

Deriving WCET estimates for various arbitration policies has been analysed in the past including RoRo [53], TDMA [84] a similar policy to RoRo with groups [53] called MBBA [85], or even a comparison between arbitration policies [54]. In [86] authors propose a method based on Performance Monitoring Counters (PMC) to enable deriving WCET estimates with Measurement-based timing analysis, when the *ubd* for a RoRo bus is known. All these works assume knowledge about the bus timing: slot sizes or maximum transfer times. Our work assumes no knowledge about the timing of the bus.

While the methodology proposed in this chapter is assessed against the NGMP processor, we expect it to hold for other processors deploying fully non-blocking caches and out-of-order execution like the ARM Cortex A9 and A15.

While in our reference architecture each core can have a single outstanding request to the L2, hence exploiting memory-level parallelism among tasks, some architectures allow several outstanding requests per core in the L2 that are properly stored while served. In the latter case, *rsk* should be designed to ensure that the L2 request buffering capabilities are saturated so that each request actually takes l_{res}^{max}.

Out of order execution, which is a challenge for itself for timing analysis, can be taken into account in the *rsk* design so that it does not change its behavior. The fact that *rsks* use only nop and memory operations should simplify this task.

4.8 Conclusions

The lack of information about internal processor timing behavior advocates for the use of measurements to derive those unknown timing parameters. For the bus and the memory, this parameter is the maximum contention delay a request can suffer when accessing the bus: *ubd*. Trustworthiness on the derived WCET bound for COTS multicore processors depends on both the soundness of the timing-analysis tool/technique and the input parameters given to the timing analysis tools, *ubd* in the case of buses and memory. In this Chapter we propose a measurement-based methodology that needs no information about the bus and memory controller timing parameters to successfully derive *ubd*. Overall our methodology increases the reliability of the WCET bound for COTS multicores deploying FIFO and RoRo buses and memory controllers.

Chapter 5

Abstracting Multicore Contention Interference: Templates and Signatures

As it has been made clear in previous Chapters, timing analysis of COTS multicores is a complex challenge that needs to be solved before multicore adoption in safety-critical real-time systems industry may become viable. Deriving an WCET bound for tasks running on multicores is challenged by the contention, also known as inter-task interference, occurring on access to hardware shared resources. Unless otherwise restrained, contention causes the execution time of any one task, hence its WCET bound, to depend on its co-runners. This has disastrous impact on system design and validation, as it conflicts with the incremental development and verification model that industry pursues to contain qualification costs and development risks. This industrial goal is sought by allowing individual subsystems to be developed in parallel against an agreed master specification, then qualified in isolation and incrementally integrated, with virtually no risk of functional regression at system level. In the time domain, incremental integration and qualification postulate *composability* in the timing behavior of individual parts, whereby the WCET bound derived for a task determined in isolation, should not change on composition with other tasks.

Several approaches have been proposed to deal with contention for multicore on-chip resources. On the one end of the conceptual spectrum in the state of the art, some authors propose computing WCET bounds so that they upper bound the effect of any possible inter-task interference a task may suffer on access to hardware shared resources. WCET bounds computed this way are *fully time composable* [87][88]. They therefore enable incremental integration and qualification, but at the cost of pessimism that may

cause untenable over-provisioning, as the timing behavior actually occurring in operation may fall much below the level determined considering the worst-case interference possible in theory [17, 26, 31]. On the opposite end, other authors [89] propose – currently only for research platforms – to determine WCET bounds simultaneously for multiple tasks in specific configurations. Those WCET bounds are *non-time composable*, as they only hold valid for the tasks being analysed and for their specific configuration. If any such parameter changes, all WCET bounds become invalid and the entire analysis has to be repeated.

In this chapter, we tackle resource contention in multicores by proposing the new concepts of resource usage *signature* (*RUs* or $\mathcal{S}$) and *template* (*RUl* or $\mathcal{L}$). *RUs* and *RUl* aim at making the WCET bound derived for an interfered task τ, time composable with respect to a particular usage u of the hardware shared resources made by the interfering co-runner tasks. The tasks' WCET bounds are determined for a particular set of utilizations $\mathcal{U}$ such that the WCET bound derived for any $u \in \mathcal{U}$ upper bounds τ's execution time under any workload so long as the co-runners of τ can be proven to make a resource usage smaller than u. We explain later what "smaller" means and how this can be determined. This abstraction allows deriving time-composable WCET bounds for individual tasks in isolation for each $u \in \mathcal{U}$, so that the system integrator can safely pull those (interfering) tasks together as long as the resource usage made by their individual set of co-runners is upper-bounded by some u. All that the system integrator has to care in that regard is to characterize the tasks' access to hardware shared resources (a low-cost abstraction of the task execution time), ignoring any finer-grained detail of that access behavior. In this chapter we present an approach to produce WCET bounds in that manner, using measurement-based timing analysis techniques.

RUs and *RUl* are, on purpose, made to be agnostic to the particular timing distribution of the resource access requests to be considered. Hence, two tasks generating the same number of accesses to a resource, though with different patterns, have the same signature. The challenge in the proposed method is in determining an effect on the interfered task that upper bounds the interference caused by contending accesses, regardless of the time distribution of those accesses as made by the interfered and the interfering tasks. In this chapter we present the following main contributions:

1. We develop the novel concepts of *RUs* and *RUl* for the timing analysis of COTS multicores and sketch an algebra of operators over *RUs*/*RUl* to enable their practical use.

2. We provide exemplary *RUs* and *RUl* for the cases when requests accessing shared resources incur either fixed or variable response latency.

3. We present an implementation of *RUs* and *RUl* for a 4-core NGMP-like [25] architecture, focusing on the bus and the memory controller as exemplars of on-chip shared resources. In our experiments we assume that the L2 cache is partitioned, as it is the case of the NGMP.

Our results show that when *RUs* and *RUl* are tailored to upper bound the access load caused by a task's co-runners, the WCET bound of that task is 1.36 times bigger than its execution time in isolation. If templates upper bound the highest number of accesses that any workload could produce, the (fully time composable) WCET bound would instead be 2.57 times bigger. *RUs* and *RUl* thus provide an effective way of abstracting resource usage in the quest for tight and trustworthy WCET bounds.

5.1 Formalization of RUs and RUl

RUs and *RUl* allow analysing, for the most part in isolation, the timing behavior of tasks, by abstracting the perturbation that they may incur from the contention for hardware shared resources occurring on a multicore caused by co-runner tasks.

5.1.1 Resource Usage signature (*RUs*)

A *RUs* abstracts the use of resources of a given interfered task, τ_A. Once computed, it will be used for τ_A's multicore timing analysis instead of τ_A itself.

We describe the use of a hardware shared resource through a set of features, which correspond to quantitative values. A *RUs* for task τ_A, is a vector $\mathcal{S}_A = (a_1, a_2, ..., a_n)$ that contains the aggregate of relevant features that characterize all the hardware shared resources, for the evaluation of contention effects. Since *RUs* are quantitative, the *RUs* of distinct tasks are comparable and can also be combined together to form a joint *RUs*.

Consider the reference multicore architecture shown in Figure 5.1(a), where the bus and the memory are shared. Further consider two types of accesses to those shared resources, for read and write operations respectively. In this case, *RUs* have at most 4 features: bus reads (n_{rd}^{bus}) and writes (n_{wr}^{bus}); memory reads (n_{rd}^{mem}) and writes (n_{wr}^{mem}). *RUs* are thus defined as $\mathcal{S}_A = (n_{rd}^{bus}, n_{wr}^{bus}, n_{rd}^{mem}, n_{wr}^{mem}) = (a_1, a_2, a_3, a_4)$.

If the bus were the only shared resource, the *RUs* of a task τ_A would be abstracted as a *RUs* with two features: n_{rd}^{bus} and n_{wr}^{bus}. If both types of requests hold the bus for the same duration, the *RUs* would consist of a single feature corresponding to the sum of n_{rd}^{bus} and n_{wr}^{bus}, i.e., $\mathcal{S}_A = (n_{rd}^{bus} + n_{wr}^{bus}) = (a_1 + a_2)$. The addition of $\mathcal{S}_B$ to $\mathcal{S}_A$

is given by $\mathcal{S}_A + \mathcal{S}_B = (a_1 + a_2 + b_1 + b_2)$. For comparison, instead, we say that $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \succsim \mathcal{S}_B$, if the interference by the former is greater than that by the latter: $a_1 + a_2 \geq b_1 + b_2$.

This reasoning easily extends to the more realistic scenario in which the bus holding times are asymmetric; for example, with reads holding the bus longer than writes. In that case, the *RUs* for τ_A could be either single-feature, considering all accesses as "long" accesses (counting writes as reads in the example), or multi-feature (two, in the example), i.e., $\mathcal{S}_A = (a_1, a_2) = (n_{rd}^{bus}, n_{wr}^{bus})$. In the latter formulation, addition and comparison change as follows: addition is defined as vector addition, i.e., $\mathcal{S}_A + \mathcal{S}_B = (a_1 + b_1, a_2 + b_2)$; for comparison, $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \succsim \mathcal{S}_B$ if $(a_1 \geq b_1) \wedge (a_2 \geq b_2)$.

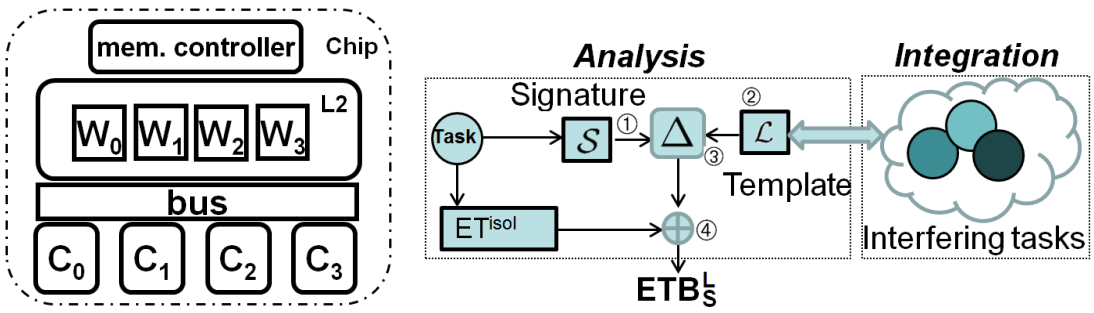
5.1.2 Resource Usage template (*RUl*)

RUl have the same form as *RUs*, namely, a vector of features $\mathcal{L}_K = (k_1, k_2, ...k_n)$, but with a different use. *RUs* abstract tasks according to their use of the shared resources while *RUl* abstracts the use of the shared resources so that $\mathcal{L}_K$ can be used as an upper bound to the interference effects caused by any task τ_i whose *RUs* $\mathcal{S}_i$ is such that $\mathcal{L}_K \succsim \mathcal{S}_i$ (i.e. $\mathcal{S}_i$ is dominated by $\mathcal{L}_K$).

Tasks are made time composable against some *RUl* $\mathcal{L}_K$ so that the WCET bound derived for a given task τ_A and for that *RUl*, denoted $WCETbound_A^K$, upper bounds τ_A's execution time inclusive of the interference that the contenders of τ_A, whose *RUs* do not exceed $\mathcal{L}_K$, may cause.

Returning to the example in which the bus is the sole shared resource with all accesses to it incurring the same contention effect: for a $\mathcal{L}_K$ that captures a given number of accesses to the shared bus, we want to determine the highest impact by $\mathcal{L}_K$ on $WCETbound_A$, so that $WCETbound_A^K$ can be regarded as a time-composable bound for τ_A in any workload in which $\mathcal{L}_K \succsim \sum_i \mathcal{S}_i$ for all co-runner tasks τ_i of interest.

A maximally time-composable template $\mathcal{L}_{TC}$ exists, which is an upper bound for any workload. $\mathcal{L}_{TC}$ corresponds to the case in which all accesses from the signature suffer the highest contention from the $N_c - 1$ contending cores. In that case, every access from $\mathcal{S}_A$ contends with $N_c - 1$ other accesses, i.e., $\mathcal{L}_{TC} = (N_c - 1) \times \mathcal{S}_A$. Any $\mathcal{L}_K \succsim \mathcal{L}_{TC}$ would produce exactly the same result as $\mathcal{L}_{TC}$, since τ_A cannot be interfered more than the accesses in its signature $\mathcal{S}_A$.



(a) (b)

FIGURE 5.1: Reference multicore architecture (a), and main steps in the *RUs* and *RUl* methodology (b).

5.1.3 Illustration of RUs and RUl use through an example

In this section we return to the case in which the bus is the sole shared resource and all accesses to it incur the same contention effect. For now we limit our attention to two cores. The task under analysis, τ_A, runs in one of the two cores. The contending requests from the two cores are arbitrated with the round-robin policy.

Figure 5.1(b) depicts the process we follow when the proposed approach is applied to this case. First, we obtain the *RUs* of τ_A, denoted $\mathcal{S}_A$. In the example architecture, the *RUs* of tasks using the shared resource is the number of accesses they make, a for τ_A, hence $\mathcal{S}_A = (a)$. Our approach treats contention such that the WCET bound of τ_A can be derived by upper bounding τ_A's execution time considering the interfering effect that it incurs when its co-runner task, whatever it is, makes up to k contending accesses to the shared resource. To this end we define a *RUl* $\mathcal{L}_K$, which is the system integration parameter that defines the inter-task interference to be considered in the determination of τ_A's WCET bound. The abstraction captured by $\mathcal{L}_K$ with $\mathcal{L}_K = (k)$ is a *RUl*.

Once the $\mathcal{S}_A$ and $\mathcal{L}_K$ are defined, we determine Δ_A^K, the increment to be applied to the execution time that τ_A may incur, to capture the contention effect from $\mathcal{L}_K$. This corresponds to step 3 in Figure 5.1(b). More precisely, Δ_A^K upper bounds the increment that the execution time of a task τ_A with at most a accesses to a shared resource may suffer from k contending requests. $WCETbound_A^K$ (i.e τ_A's WCET bound determined under the *RUl* $\mathcal{L}_K$) is computed as the summation of ET_A^{isol}, the execution time of τ_A when running in isolation, without contention, and Δ_A^K, the increment that upper bounds the contention effects from any k interfering accesses. This corresponds to step 4 in Figure 5.1(b). Overall, $WCETbound_A^K$ is time composable against any co-runner task τ_B with signature $\mathcal{S}_B = (b)$, as long as the *RUs* of the co-runner is lower than $\mathcal{L}_K$, which means that τ_B makes $b \leq k$ contending accesses. We denote this as $tc(WCETbound_A^K, \tau_B)$, which holds if $b \leq k$.

RUs abstract the distribution of requests over time. Taking into account the exact distribution of requests over time, for instance in the form of requests arrival curves [90], would potentially enable deriving tighter WCET bound. However, deriving such distributions is complex, as programs normally have multiple paths of execution, each with its own access pattern (distribution). And, paradoxically, considering these particular distributions would decrease timing composability. Instead, our approach only requires the tasks' access count for every individual shared resource, as well as ET_i^{isol} (execution time in isolation) for each individual task τ_i. Notably, both are already had with high accuracy by state-of-the-art technology, e.g., [91]. With our approach, the ability to abstract away from the need to know the exact points in time at which requests would be made to shared resources releases the system integrator from the obligation of adopting rigid and inflexible scheduling decisions (which fares poorly with the development unknowns of novel systems) or from the labour-intensive cost of exact analysis.

Our approach requires the user to set the *RUl* to capture the potential co-runner tasks precisely. The spectrum of this capture has two ends. On one extreme we find the time-composable templates, $\mathcal{L}_{TC}$, which represent an upper bound for *RUl*. However, if *RUl* is close to that template, the WCET bound of tasks might be unnecessarily increased. On the opposite extreme, if *RUl* is too small, it constrains the choice of tasks that may be allowed to run in parallel. A simple solution consists in deriving for each task an WCET bound under different *RUl*, such that at integration time, the smallest *RUl* that upper bounds the signature of the actual co-runner tasks is used. With this, the residual part of the timing verification at system integration is small and simple. Selecting the proper number of *RUl* represents a trade-off between effort and accuracy: the higher the number of *RUl* the lower the over-estimation of WCET bound and the greater the analysis time, and vice-versa. Finding appropriate *RUl* is a standard optimization problem that is part of our future work.

In the example considered in this section we have made several simplifications to facilitate understanding: two cores, one single type of access, synchronous accesses (i.e. the core stalls when the access occurs until served) and a single shared resource. In real processors we have different types of accesses to the shared resource (synchronous and asynchronous), each with a distinct access latency. Hence, simply bounding the effect of contention by adding access counts is not enough.

5.2 RUs & RUl for Measurement-Based Timing Analysis

Next we present one concrete realization of *RUs* and *RUl* for use with measurement-based timing analysis (MBTA), specifically for a NGMP-like processor architecture [25].

5.2.1 Methodology

For this approach we make use of RSK(see Chapter 2) for our methodology, in this case we design them for the architecture in Figure 5.1(a), which is a schematic view of that in Figure 2.2. For this purpose the basic structure a loop body is mostly composed by load instructions that hit in the L2 cache stresses the bus. For the sake of clarity in this specific chapter we distinguish between two kinds of RSK by their purpose no their composition:

Resource stressing kernels, *RSK*, place a configurable load on a given shared resource, so that running a task against a *RSK* may represent contention scenarios of interest.

In theory, one could design a *worst-contender* kernel that generates the maximum contention that a task τ_i can suffer. However, such kernel would be specific for the task to be interfered and for the target processor [31]. Consider for example, a single shared resource arbitrated by a least-recently-used policy, where the task that accessed the resource last gets the least priority. In that case, the worst-contender kernel should generate a request in exactly the same cycle as the task of interest, so that every request from that task gets delayed by the contender, and for the next round of arbitration the task has the lowest priority again. The level of control required on the application behavior and the granularity of intervention are too fine-grained and laborious to be used in practice [31].

Resource sensitive kernels, *RSeK*, are designed to upper bound the execution time increase suffered by any other task, with a smaller or equal signature, owing to the interference from a given template $\mathcal{L}_K$. Consider a scenario in which bus accesses hold the bus for a constant duration. Further assume that we want to determine Δ_A^K for τ_A, i.e its WCET bound increment due to a template $\mathcal{L}_K$ with k accesses. Intuitively, one could get an estimate of it by running τ_A several times against a *RSK* that makes k accesses. However, in order to gain confidence in the WCET bound obtained, the experiment should be repeated with different *alignments* of the *RSK*, so that the interleaving of accesses varies enough and the worst case can be observed in a measurement. In practice, this may require excessive experimentation effort. The need for repeating the experiments with different alignments stems from the uncertainty on the time distribution of accesses, which is hard, if at all possible, to measure and control by timing analysis technology. We can therefore conclude that studying the task under analysis against micro-kernels is not viable. Instead, we use micro-kernels *to model both the interfered and the (set of) interfering tasks*: *RSK* and *RSeK* are designed to account for bad alignments of requests: *RSeK* is made of instructions that cause accesses to the shared resource and that continuously contend with *RSK* requests.

We define $\Delta^{RSK}_{RSeK} = ET^{RSK}_{RSeK} - ET^{isol}_{RSeK}$, where ET^{RSK}_{RSeK} is the execution time when a given $RSeK$ with the same signature as task τ_A runs against a RSK implementing a template $\mathcal{L}_K$ with k accesses; and ET^{isol}_{RSeK} the execution time when the $RSeK$ runs in isolation. For task τ_A, let $\Delta^K_A = ET^K_A - ET^{isol}_A$ be the execution time increase τ_A suffers when it runs against $\mathcal{L}_K$. $RSeK$ and RSK are designed so that $\Delta^{RSK}_{RSeK} \geq \Delta^K_A$ holds for any request alignment of τ_A under $\mathcal{L}_K$ contention. To that end, we run the $RSeK$ in isolation and then against $N_c - 1$ copies of RSK so that all $RSeK$'s accesses to the shared resource suffer high contention, causing a measurable Δ^{RSK}_{RSeK} to emerge. In the next section we show how to derive the number of accesses of the $RSeK$ and the RSK, based on the number of accesses of the template and signature under consideration.

Δ^{RSK}_{RSeK} is used to compute the WCET bound estimate for τ_A as follows: $WCETbound^K_A = ET^{isol}_A + \Delta^{RSK}_{RSeK}$. $WCETbound^K_A$ is composable with any set of interfering tasks against which τ_A runs in parallel, if their total number of accesses is lower or equal to k. That is, the addition of the signatures of the interfering tasks is dominated by $\mathcal{L}_K$: $(S_i + S_j + ... + S_l) \precsim \mathcal{L}_K$. Interestingly, given a task τ_B whose signature is dominated by τ_A, i.e. $S_B \precsim S_A$, the obtained Δ^{RSK}_{RSeK} for τ_A can be used to upper bound τ_B's execution time: $WCETbound^K_B = ET^{isol}_B + \Delta^{RSK}_{RSeK}$.

Overall, RUs and RUl provide powerful abstractions for the interfered and the interfering tasks, which simplify the integration of multiple tasks by combining their signatures.

5.2.2 The case of a NGMP-like architecture

Our reference multicore architecture [25] comprises $N_c = 4$ symmetric cores, see Figure 5.1(a) (a schematic of that in Figure 2.2), each equipped with private instruction cache (IC) and data cache (DC). The cores have an in-order time-anomaly-free design [92]. Load operations are blocking, whereby the pipeline is stalled until the load is resolved. Each core has one 2-entry write-buffer that holds store requests until they are resolved, without stalling the processor. The processor is stalled solely to preserve memory consistency, when a store finds the write-buffer full or a load operation finds the write-buffer non-empty.

Bus. Our example processor implements round-robin bus arbitration so that if, in a given round, core $c_i, i \in \{1, .., N_c\}$ is granted access to the bus, the priority ordering in the next round is: $c_{i+1}, c_{i+2}, ..., c_{N_c}, c_1, c_2, ..., c_i$. A lower priority core can use the bus when all higher priority cores do not use it. The *bus access jitter* that a task incurs on access to the bus, depends not only on the number of co-runners but also on the way their requests interleave. The worst contention situation happens when a task τ_B

interfered	st			l2h			l2m		
interfering	st	l2h	l2m	st	l2h	l2m	st	l2h	l2m
Impact	2	7	2	2	7	2	2x2=4	2x7=14	2x2=4

FIGURE 5.2: Impact from/to the different access types to the bus.

assigned to core c_i requests the bus in a given round of arbitration, simultaneously with tasks in all other cores and the previous round was assigned to c_i.

L2 cache. The L2 cache processes up to one miss per core at a time and allows hit-under-miss and miss-under-miss so that when a miss from a core is processed, hit/miss requests from other cores can be served. The 4-way L2 is partitioned so that every core is allowed to use 1 way[1].

Memory controller. The L2 sends a request to the memory controller on every L2 miss. Requests are stored in a FIFO request queue, with one entry per core. The memory controller assumes a single DRAM device with close-page policy.

5.2.3 Bus

The bus handles three distinct request types, which differ in the contention they induce and suffer. Stores (st) either hit or miss on the L2, which are served immediately by the L2 and hold the bus for 2 cycles. L2 load hits ($l2h$) hold the bus for 7 cycles because they are not split by the bus and insert wait states on the bus for the hit latency of the L2 (5 cycles). L2 load misses ($l2m$) that are split by the L2 and perform a new arbitration whenever the L2 responds to the miss, holding the bus 2 cycles in each arbitration. Figure 5.2 shows the contention suffered by a source (interfered) request by another (interfering) request for all request types. $l2h$ generate the highest contention and $l2m$ are the most affected since they suffer two rounds of arbitration: $l2m$ can therefore be interfered twice by two concurrent contending requests, one round of arbitration per each such request.

Our approach based on RUs and RUl does not require knowing the exact time of request issue, but whether they have asymmetric timing behavior in the impact they suffer and they cause to other request types so that RSK and RSeK can be designed with the appropriate request types. The RSK and $RSeK$ for the bus are called BSK and $BSeK$:

BSeK (abstracting interfered task bus usage). The signature of a task τ_A running in this architecture may take different forms, with different levels of tightness and experimentation effort. The canonical signature for the bus contains the number of accesses

[1]The GR712RC and the NGMP do implement this feature.

of each type made by the task. That is: $\mathcal{S}_A^{bus} = (a_{st}, a_{l2h}, a_{l2m})$. This can be simplified by realizing that $l2h$ and st access the bus once whereas $l2m$ do it twice with exactly the same timing as $l2h$ and st. Moreover, the delay suffered by an access does not vary whether the access was generated by a $l2h$, st or $l2m$. Hence, signatures have the form: $\mathcal{S}_A^{bus} = (a_{st} + a_{l2h} + 2 \times a_{l2m})$.

BSeK can be implemented with either $l2h$ or st. $l2m$ are not appropriate as it is not possible to place high pressure on the bus with $l2m$ since they miss in cache and take long to be served from memory, leaving the bus idle in the meantime. $l2h$ and st instead can place very high pressure on the bus. Our approach considers *BSeK* to only have st operations.

BSK (abstracting interfering task(s) bus usage). Templates can be mono- ($\mathcal{L}_{1D}$) or bi-dimensional ($\mathcal{L}_{2D}$).

$\mathcal{L}_{2D}$. st and $l2h$ generate different impact on the bus (recall that $l2m$ are equated to 2 st). In particular, $l2h$ produces the highest impact and st the lowest. This allows generating bi-dimensional templates: $\mathcal{L}_{2D} = (k_{l2h}, k_{2\times l2m+st})$, whereby *BSK*s comprises load L2 hit accesses and store accesses to generate each respective type of interference.

$\mathcal{L}_{1D}$ templates comprise only $l2h$, which generate the highest interference. A given $\mathcal{L}_{1D} = (k_{l2h})$ with k $l2h$ accesses upper bounds the impact that one or several tasks, whose bus access count is lesser or equal to k, can generate on any other interfered task. $\mathcal{L}_{1D}$ are easier to generate and simplify experimentation, but they increase the pessimism of WCET bounds, since st are considered to generate the same impact as $l2h$.

Putting it all together. Deriving the access count for *BSeK* and *BSK* varies for $\mathcal{L}_{1D}$ or $\mathcal{L}_{2D}$ as we show next.

$\mathcal{S}_A - \mathcal{L}_{1D}$. Let a and k be the number of accesses in the signature $\mathcal{S}_A$ and the template $\mathcal{L}_K$ respectively. Running *BSeK* and *BSK* concurrently, we derive an upper bound to the increase in execution time (the delta) that k accesses of the template can have on the a accesses of the signature. If $k \geq (N_c - 1) \times a$ then each request of $\mathcal{S}_A$ suffers the impact of $N_c - 1$ contenting requests. If this is not the case, only $\lceil k/(N_c - 1) \rceil$ requests from $\mathcal{S}_A$ suffer impact.

The number of request accesses generated by the *BSeK* is given by $N = min(a, \lceil k/(N_c - 1) \rceil)$. By running this *BSeK* against $N_c - 1$ *BSK* copies, each having a number of accesses largely above N, we derive an upper bound to the impact that $\mathcal{L}_K$ has on $\mathcal{S}_A$. The impact that a task can suffer due to a template $\mathcal{L}_K$ with k $l2h$ is upper bounded

as: $\Delta^{BSK}_{BSeK} = ET^{BSK}_{BSeK} - ET^{isol}_{BSeK}$. The WCET bound derived for a given task τ_A and template $\mathcal{L}_K$ is: $WCETbound^K_A = ET^{isol}_A + \Delta^{BSK}_{BSeK}$.

$\mathcal{S}_A - \mathcal{L}_{2D}$. In this case we account for the fact that requests sent by the interfered task, τ_A, suffer different interference by the $l2h$ and $l2m/st$ sent by the interfering tasks, abstracted in $\mathcal{L}_{2D}$. In this approach we pair up every request in τ_A with $N_c - 1$ requests in $\mathcal{L}_{2D}$ causing the highest interference ($l2h$) on the former. If the number of those requests in $\mathcal{L}_{2D}$ is exhausted, we pair up τ_A requests with those in $\mathcal{L}_{2D}$ causing the second worst interference (st).

We generate two $BSeK$ and BSK pairs to capture the impact that accesses in $\mathcal{S}_A$ suffer from $l2h$ and $l2m/st$ in $\mathcal{L}_{2D}$ so that:

$$\Delta^{BSK}_{BSeK} = \left(\Delta^{BSK_l}_{BSeK_1} + \Delta^{BSK_s}_{BSeK_2}\right) \tag{5.1}$$

$BSeK_1/BSK_l$ and $BSeK_2/BSK_s$ capture the interference on τ_A's accesses caused by the $l2h$ and $l2m/st$ in $\mathcal{L}_{2D}$ respectively. $BSeK_1$ and $BSeK_2$ have different number of st operations, N_1 and N_2. BSK_l comprises $l2h$ operations whereas BSK_s comprises st operations.

Let us assume for example $a = 30$, $k_{l2h} = 60$, and $k_{st} = 80$. In this case, $BSeK_1$ has $N_1 = min(30, \lceil 60/3 \rceil) = 20$ st, which we pair up with 20 accesses in $\mathcal{S}_A$; and $BSeK_2$ has the rest of accesses in $\mathcal{S}_A$, $N_2 = 30 - 20 = 10$ st, which we pair up with 3×10 requests out of the 80 accesses in k_{st}. The remaining 50 st in k_{st} are not paired since they will not cause further impact on $\mathcal{S}_A$. Overall, an upper bound to the impact that an application can suffer due to $\mathcal{L}_{2D}$ is given by:

$$WCETbound^K_A = ET^{isol}_A + \left(\Delta^{BSK_l}_{BSeK_1} + \Delta^{BSK_s}_{BSeK_2}\right) \tag{5.2}$$

5.2.4 Memory Controller

Memory uses a close-page round-robin memory controller. The request queue holds one request per core. The memory has two characteristic latencies, the data latency and the request latency. The data latency is the latency between the instant in which the request is sent to the memory devices and the moment in which the data are already in the processor so that the L2 can be given an answer. Data and request latency are not the same, since the memory needs some time after providing the data (data latency) to close the row accessed by the request. Read and write requests accesses have different data latencies, but have the same request latency. The request latency is the one that defines the interference that each request suffers and generates, this means that all types

of combinations between read and write memory accesses generate and suffer the same interference, which is 23 cycles in our case although this information is not needed to use *RUs* and *RUl*. This means that a core accessing memory will have to wait 23 cycles for each other core in the queue.

We differentiate between three different types of interfering memory traffic: (1) read requests generated by store misses on the L2 (*stm*); (2) write requests generated by dirty misses on the L2 (*dtm*); and (3) read requests generated by load misses on the L2 (*ldm*). Read and write accesses have different data latencies, but have the same request latency. The request latency is the one that defines the interference that each request suffers and generates, this means that all types of combinations between read and write accesses on the memory generate and suffer the same interference which is the request latency (RL), times the number of contenders, i.e. cores, in the processor.

For the memory controller we follow the same principles as for the bus, with the particularity that the impact from/to the read/write request types is homogeneous. Hence we only need $\mathcal{L}_{1D}$ templates. Since the impact on the requests is homogeneous, we only need $\mathcal{L}_{1D}$ templates. The *RSK* and *RSeK* for the memory are called *MSK* and *MSeK*: $\mathcal{S}_A^{mc} = (a_{mem})$, where a_{mem} is the number of accesses made by the task to the memory controller, and the WCET bound derived for a given task τ_A and template $\mathcal{L}_K^{mc}$ is given by: $WCETbound_A^K = ET_A^{isol} + \Delta_{MSeK}^{MSK}$.

5.2.5 Multi-resource signatures

In the presence of multiple shared resources, the signatures and templates must cover the hardware features so as to soundly upper bound contention in each of them. For the reference architecture considered in this work, signatures and templates are as follows: $S_A^{bus+mc} = (a_{st} + a_{l2h} + 2a_{l2m}, a_{mem})$ and $\mathcal{L}_K^{bus+mc} = (k_{st} + 2k_{l2m}, k_{l2h}, k_{mem})$.

It is possible that a task suffers contention in several shared resources simultaneously, so that the impact of the contention does not accumulate but rather overlaps. However, determining trustworthy bounds to the degree of overlap in the contention suffered on requests to different resources is complex. Signatures and templates are intentionally made agnostic to the distribution of requests over time. As we focus on the number of requests to each resource rather than on their timing, it is difficult to determine how contending requests overlap. Our current approach assumes no overlap in contention, which in our time-anomaly free processor design is a safe assumption on the maximum impact of contention. Overall, in the presence of a template for the bus $\mathcal{L}^{bus}$ and the memory $\mathcal{L}^{mc}$ (a.k.a. $\mathcal{L}^{bus+mc}$), a task is assumed to suffer the sum of the contention generated by both templates:

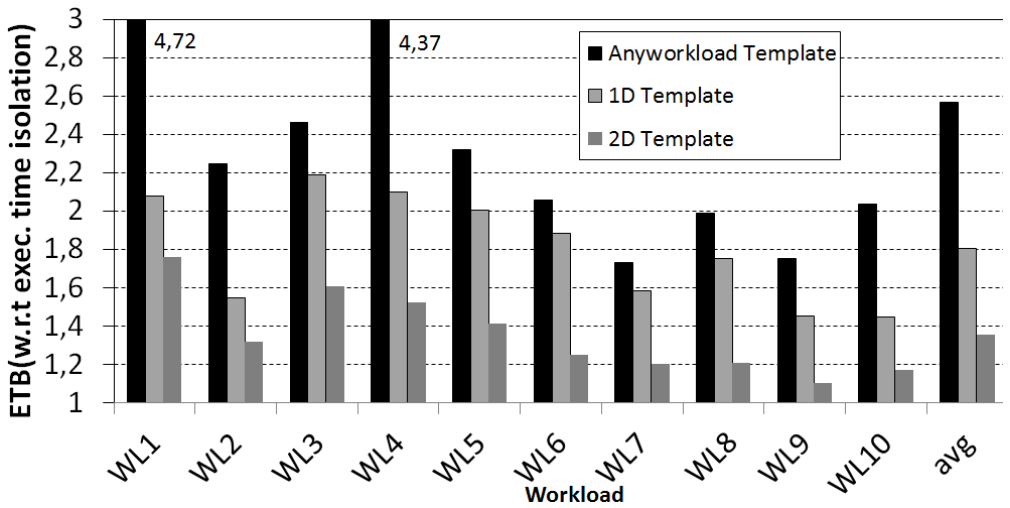


FIGURE 5.3: WCET bounds for different templates for 10 4-task workloads. Results are normalized to the execution time in isolation.

$$WCETbound_A^{\mathcal{L}_K^{bus}+\mathcal{L}_K^{mc}} = ET_A^{isol} + \Delta_{BSeK}^{BSK} + \Delta_{MSeK}^{MSK}$$

5.3 Evaluation

For our evaluation, we model a 4-core NGMP-like symmetric multicore [25] comprising a bus connecting cores to the L2 cache and an on-chip memory controller, analogous to that used in Chapter 4. This processor model is relevant as it constitutes a potential baseline for the space domain. As discussed in Chapter 2, the performance estimates provided by our simulator against a real NGMP implementation, the N2X [28] evaluation board, proved to be very precise.

Our *RSeK*/*RSK* approach works on the premise that the contention suffered by each request of the *RSeK* upper bounds the contention suffered in any other scenario. The authors of [86] show that round-robin arbitration can have anomalous cases when a higher number of contenders introduces less contention on the bus. In fact, we show in Chapter 4 that the *RSK* cannot necessarily generate the worst (maximum) contention on *RSeK*, due to the alignment of requests. To solve this, we applied a solution based on adding *nop* operations between *RSeK* requests to modify their alignment. For instance, in the case of the bus, since we use *store* requests for the *RSeK* (see Section 5.2.3), we prove that each *RSeK*'s request suffers the maximum contention [21]. In our reference architecture, if *load* operations were used in the *RSeK*, each request would suffer exactly one cycle less than the maximum contention on each request as shown before, which can be addressed with the solution presented in previous chapter.

5.3.1 Experimental results

Our evaluation was carried out along 2 axes. First, we compared the tightness of 1D and 2D templates against fully-time composable WCET bound, that can be obtained by software [26][31] or hardware [53] methods. Secondly, we compared 2D templates, for which tighter results are obtained, to the case in which the task under analysis runs against *RSK*.

1D vs 2D signatures. Figure 5.3 compares the scenario with a fully-time composable template, $\mathcal{L}_{TC}$, valid for any workload (*any workload template* in the figure), with 1D ($\mathcal{L}_{1D}$) and 2D ($\mathcal{L}_{2D}$) templates fitting the potential interference in the corresponding workload. We analyze 10 randomly generated workloads and show results for the benchmark running on core 0. Similar results are obtained for the other cores.

For instance, for workload W8 <*pntrch*(PN), *basefp*(BA), *a2time*(A2), *tblook*(TB)>, we consider PN as the task under analysis and a template that corresponds to the aggregate of signatures of the three other benchmarks. This causes $\mathcal{L}_{1D}$ to have $564,227$ bus accesses (as many as the addition of bus accesses of *BA*, *A2* and *TB*). This is abstracted by RUs/RUl so that only $564,227/3 = 188,076$ bus accesses from PN suffer high contention and the rest suffers no contention. To measure this effect, we run a *BSeK* with 188,076 accesses against 3 BSK with a large number of accesses. The same process is followed for the memory. $\mathcal{L}_{2D}$ is generated analogously, but considering separately *l2h* and *st* bus accesses.

Figure 5.3 shows the WCET bound for the first benchmark in the workload (under *anyworkload*, $\mathcal{L}_{1D}$ and $\mathcal{L}_{2D}$), normalized to its execution time in isolation. We observe that fitting templates to actual contention ($\mathcal{L}_{1D}$ and $\mathcal{L}_{2D}$) in the workload tightens WCET bounds significantly. This effect is particularly noticeable for WL1 and WL4. Also, in all cases $\mathcal{L}_{2D}$ provides tighter WCET bounds than $\mathcal{L}_{1D}$. This is so because with $\mathcal{L}_{1D}$ all accesses to the bus are assumed to be *l2h*, which generate the highest contention, while $\mathcal{L}_{2D}$ better captures the fact that there are two type of requests generating different contention (*l2h* and *l2m-st*). For instance, WL4 has a normalized WCET bound of 4.37 (more than 4x the execution time in isolation) when using a template valid for any workload. If we use $\mathcal{L}_{2D}$ for this workload, the WCET bound is only 1.53. Overall, our approach allows reducing the WCET bound from 2.57 to 1.8 with $\mathcal{L}_{1D}$ and 1.36 with $\mathcal{L}_{2D}$ templates on average for the 10 workloads.

Owing to strict page limits we are unable to report the contention impact generated by the memory. Notably however, in our processor set-up the bus has higher impact than the memory, as the L2 cache filters out most memory accesses. Of the contention impact in $\mathcal{L}_{2D}$, 78% stems from the bus and only 22% from the memory.

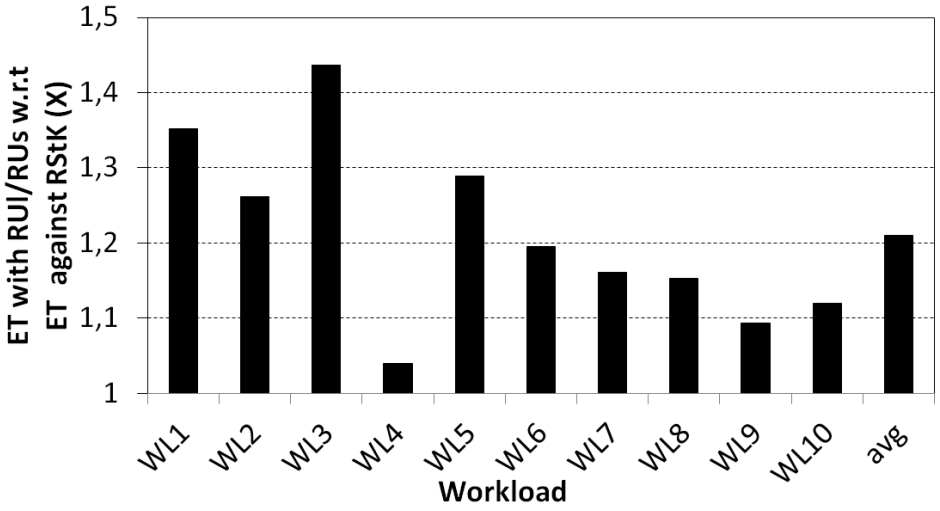


FIGURE 5.4: Overestimation incurred by RUs/RUl

RUs/RUl vs. EEMBC/RUl. In order to assess the pessimism incurred in WCET bound obtained with $\mathcal{L}_{2D}$ we compared them with the execution time for the task (i.e EEMBC), denoted ET, taken when the task run as part of a workload comprising RSK [26][31]. This workload represent a pessimistic yet possible contention scenario that the task can suffer. Figure 5.4 shows WCET bound obtained with $\mathcal{L}_{2D}$ relative to ET. Notably, the incurred pessimism was always below 45%, 20% on average. We contend that the benefits provided by RUs/RUl in the simplification of timing analysis upon system integration, pays off for the increase in WCET estimates.

5.4 Related Work

Contention on access to hardware shared resources has been thoroughly studied in the state of the art. A taxonomic summary of the relevant works has been presented in Chapter 3. Authors in [93] propose a methodology to obtain the signature of tasks and replace them with kernels that mimic their shared resource usage pattern as a way to reduce the variability in measurement-based analysis. Instead, we use signature and templates to abstract the contention tasks cause and suffer, bounding contention effect [21]. Works addressing off-chip contention assume no contention for on-chip resources, which are assumed replicated. Off-chip contention for the bus is handled with TDMA buses [32] whose analysis case is the worst possible alignment of the task requests to their TDMA slots. Works assuming dynamic arbiters [33] consider the particular pattern of accesses of each contender to the bus. For on-chip resources, two main approaches have been followed, both requiring some hardware support: isolation or bounded interference. The former uses TDMA arbitration and partitioned caches to prevent interaction among

tasks [84]. The latter bounds the maximum impact that one task may generate on co-runners [53]. However, as far as we can tell, such specialized hardware support is not fully or readily available to industry: while cache partitioning has been implemented in hardware, e.g. in the Cobham Gaisler NGMP and the ARM A9, for the bus and the memory controller instead such support is not provided. When the shared cache is not partitioned, alternative solutions – around the concept of *partial time composablity* – have been proposed to approximate the time composability properties provided by templates and signatures [88].

In the absence of hardware support in COTS processors, contention effects can be analysed, bounding the memory latency (for instance for Intel Core-i7 [77]), or even deriving WCET estimates (for Freescale P4080 [18]). In the latter research, authors use a static timing analysis approach with run-time monitoring of the resource usage that benefits from the knowledge of the workload to be able to derive tight WCET estimates. As a consequence of the limitations in the state of the art for COTS, the execution time of a task becomes dependent on its co-runners, which is a major impediment to incremental development and qualification. This is the challenge we have tackled with our approach based on resource signatures and templates.

5.5 Conclusions

We presented a novel approach to studying the contention on the bus and memory controller, building on the concept of Resource Usage Template(RUs) and Resource Usage Signature(RUl) that abstract the resource usage made by the task under analysis and by its contenders. These notions help abstract the interference impact suffered by the task under analysis and the interference effects generated by its contenders. The notions embodied in our proposal provide a simple yet powerful mechanism to aid time-composable integration of multiple tasks in a multicore. A wise selection of RUl allows obtaining tight upper bounds to execution time, for modest cost and effort, thereby facilitating incremental development and qualification for systems targeting COTS multicore processors.

Chapter 6

Surrogate Applications Generation

In integrated architectures, such as IMA or AUTOSAR, assessing during early design phases whether applications fit their timing provide several benefits. In particular, it allows each software provider to carry out the verification of its application on its own, before applications are integrated. However, deriving contention bounds in multi-cores require collecting measurements of the application under analysis running against contender applications developed by other software provider, which may not exchange applications due to IP confidentiality reasons.

We introduce the concept of Surrogate Applications (SurApp) as a means to attack this challenge. A SurApp copies the activities generated by the real (target) application on hardware shared resources, such as accesses to buses, memory, and caches. SurApps are automatically generated from information (e.g. memory access counts) collected from the execution in isolation of real applications.

Given applications A and B, our goal is making that the slowdown that A suffers when it runs against the SurApp of B ($SurApp_B$) matches the slowdown A suffers when it runs against B. Sharing SurApps during early design phase instead of applications, enables software providers to run their applications against other software provider's SurApps in the multicore and derive the slowdown their applications suffer due to contention. This approach incurs no violation of IP since SurApps only copy activities of the real applications, rather than their functional behaviour or source code; and can be developed during early design phase as soon as each software provider can generate a binary of its applications.

We tailor SurApps to the Cobham Gaisler NGMP processor [81], considered for on-board processing for future Space's missions. We reproduce the contention created by real applications in shared last level caches, bus and memory, whose impact in timing behaviour has been shown to be high [26, 73]. In detail, the contributions of this Chapter are:

1. We identify key performance indicators to be copied by SurApps to accurately mimic cache, bus and memory contention. In particular, *stack distance* is the main indicator of applications' behaviour. We further elaborate on how stack distances of the target application (running in isolation) can be obtained with the tracing support in real NGMP boards.

2. We present SurAppGen, an automatic framework that generates SurApps that copy the stack distances of the target application. We show the main elements of SurAppGen design when copying the real application's behaviour.

3. Our results show that SurApps mimic the stack distance behaviour of the real applications resulting in similar shared cache, bus and memory behavior, and so similar slowdowns on other contender applications on the NGMP processor.

6.1 Overall Approach and Target Platform

Time and space partitioning concepts from IMA-SP provide functional isolation, e.g. via Virtual Machines [94] like ARINC653-based Fentiss' XtratuM [95] or GMV's AIR [96]. Applications are provided quotas (budgets) on CPU time and resource usage (e.g. memory). In terms of timing, under ARINC653, time is divided into major frames that repeat over time. Each major frame contains several minor frames in which different applications (functions) run. Each software provider derives WCET estimates for its applications and determines whether each one (e.g. A) can finish in its assigned minor frame. Yet, in multicores applications affect each other's timing behavior due to sharing of hardware resources (e.g. buses and caches), which hinders WCET estimation since software providers have to derive an upperbound to the contention that other corunning applications cause on A.

early design phase covers several stages prior to software integration. In a first stage, before the application's binary is produced, timing estimates are derived at a "coarse grain" and do not consider elements like multicore contention [97][98]. In a second phase, once the binary is generated (e.g. by properly stubbing the function), contention can be factored in the timing estimates [99]. The function undergoes integration phases

where functions of the different software providers are consolidated into the final software image.

SurApps are deployed when the binary is generated but not integrated with other functions, which carries the difficulties described in the previous section. During early design phase no strict guarantees are required – rather required in late design phase. In early design phase the goal is to have good estimates of the WCET of the task with tendency towards over-estimation to prevent costly late design phase timing violations [100]. No particular figure is reported in the literature on the accuracy required during early design phase. Yet several works show that the impact of contention in the NGMP can reach 20x for some kernels and 5.5x for some benchmarks [26]. Hence, we deem the accuracy results obtained by our approach (13% on average) as high.

With SurApps, each software provider runs each application in isolation, see Figure 6.1. The software provider leverages the tracing support in the underlying board to obtain the key performance indicators used to copy his application behavior. In a second step, the software provider executes the SurAppGen that builds on those key performance indicators to create a SurApp that copies the behavior of the real application. software providers exchange SurApps, so for a given application App_A, the corresponding software provider can determine the slowdown it suffers due to hardware resource sharing with its contenders by running it against their SurApps. If there is no violation of the budgets, the scheduling plan is deemed as valid. Otherwise, the integrator may increase the budget given to a software provider or change the scheduling. On its side, the software provider can reduce the CPU requirements of its application.

Target Platform. We focus on a reference platform for on-board processing, the NGMP [81], presented in Chapter 2, and whose schematic is shown in Figure 6.2. In this work, in which we propose SurApps, we have applied them to the main memory path: bus, cache, and memory. While we expect the applicability of SurApp to I/O interfaces to be similar to that for memory, an analysis and evaluation of SurApps to other I/O interfaces is left for future work.

Scope. The general concept of SurApps can be applied to any multicore architecture. However, in this work we tailor SurApps to the NGMP. We do not foresee major roadblocks for tailoring SurApps to other similar multicore processors (e.g. ARM big.LITTLE architectures). We focus on the case in which only the LLC is shared. If multiple cache levels were shared, accesses to LLC would depend on contention in lower cache levels, so our approach would not be applicable directly. However, multicores being considered for critical real-time systems have nowadays at most a single shared cache level.

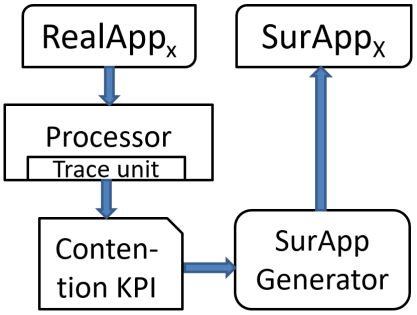


FIGURE 6.1: Diagram of SurApp generation.

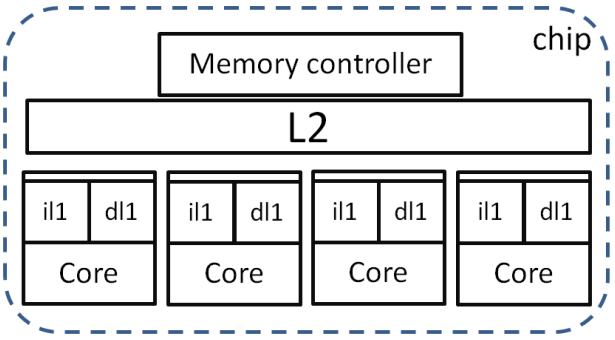


FIGURE 6.2: Simplified view of NGMP's main shared resources.

6.2 Surrogate Applications

When applications A and B execute concurrently in a processor with a shared L2 cache, bus and memory controller, contention depends on several parameters: some can be derived during early design phase while others depend on how applications interact, and hence, can be derived only during late design phase.

As early design phase parameters we identify the L2 access frequency of the task, which is determined by the miss rate in DL1 and IL1. Further, for write-through DL1, the number of stores affects the access frequency to the L2. Since DL1 and IL1 are private to each core and non-inclusive, their behavior is mostly unaffected by contenders.

As late design phase parameters we have the L2 cache miss frequency, which depends on (i) the task's own access pattern to L2 and (ii) contender tasks' L2 access patterns. achieves good accuracy results. Our initial experiments reveal small accuracy improvements when SurApps copy the distribution of application accesses over time. This might be related to the fact that in-isolation distributions change due to the contention with other tasks.

6.2.1 Stack Distance as a Proxy for Multicore Contention

Eviction policies like LRU present the stack property [101]: each set in a cache can be seen as a LRU stack, where lines are sorted by their last access cycle. The first line of the LRU stack is the most recently used (MRU) line, whereas the last line is the LRU. The position of a line in the LRU stack defines its stack distance. Further, those accesses with a stack distance (sd) smaller than or equal to the number of cache ways (w) result in a hit and vice versa: $n_{hit} = \sum_{i=0}^{w-1} sd_i$ and $n_{miss} = \sum_{i=w}^{+\infty} sd_i$. The stack distance of an access $@A_k$ is the number of unique, i.e. non-repeated, addresses mapped to different cache lines to the same set where $@A_k$ is and that are accessed between $@A_k$ and the

previous access to it, $@A_{k-1}$. For instance, in the sequence $@A_1@B_1@C_1@B_2@A_2$ of cache line accesses to a given cache set, the stack distance of $@A_2$ is 2, since repeated accesses to $@B$ are only counted once, regardless of how many accesses there are.

Since we focus on independent applications, L2 misses are not affected either by contention: if an access of a given application results in a miss when the application runs in isolation, it will also result in a miss in any workload in which the application runs. Hence, only hits to the L2 can become misses due to evictions of contender tasks. With stack distance we formulate the case where an access (e.g. $@A_k$) that hits in L2 when the application runs in isolation becomes a miss when the application runs in multicore mode. The in-isolation stack distance of $@A_k$ is smaller than w, and, in multicore it becomes equal or larger than w, i.e. $sd_{isol}(@A_k) < w \;\; and \;\; sd_{muc}(@A_k) \geq w$. $sd_{isol}(@X)$ defines the stack distance of an access $@X$ when the application runs in isolation, and $sd_{muc}(@X)$ when the application runs in multicore under a given workload. The increase in the stack distance occurs due to the accesses performed by contender tasks to the same set where $@A_k$ is mapped to. It follows that mimicking the stack distance of an application's accesses is critically important to accurately reproduce the impact it has on other applications.

6.2.2 Stack Distance per Kilo Instruction (sdki)

In our approach, we define stack distance k per thousand (kilo) of instructions or $sdki_k$ as the number of accesses with stack distance k every 1,000 executed instructions. The normalization to thousands of accesses is done since stack distance values per instruction are naturally very low (below 1). For each application we collect the stack distance vector (SDV) that has $w+1$ entries: $SDV = [sdki_0, sdki_1, sdki_2, ..., sdki_w]$, with $sdki_w$ counting all accessed with stack distance $\geq w$, since all them result in misses. Interestingly, with the $sdki$ formulation we can derive other well-known cache parameters, such as accesses per kilo instruction and misses per kilo instruction, $apki = \sum_{i=0}^{w} sdki_i$ and $mpki = sdki_w$ respectively. Note that we generate one SDV for loads and one for stores referred to as $ldSDV$ and $stSDV$ respectively, with $SDV = ldSDV + stSDV$ i.e. $sdki_i = ld\_sdki_i + st\_sdki_i \;\; \forall i \in [1, .., w]$.

Bus access count can be expressed as $n_{bus} = (ld\_apki + st\_apki) \times 1000 = \sum_{i=0}^{w}(ld\_sdki_i + st\_sdki_i) \times 1000$. That is, all load and store accesses to the L2 cache are bus accesses, regardless of their stack distance.

Memory access count: We derive access count to memory as misses per kilo instruction, and $n_{mem} = ld\_mpki + st\_mpki = ld\_sdki_w + st\_sdki_w$. That is, the number of L2 cache misses represent the number of memory accesses.

Overall, *sdki* provides a powerful abstraction of applications' bus, cache and memory usage that despite not copying late design phase-application information, provides tight results.

Obtaining sdki. To derive $sdki_k$ from target applications when run in isolation, we can make use of standard tracing facilities existing on many architectures. For instance, the LEON processor family allows collecting instruction and data addresses, opcode and timestamp of all instructions. GRMON is configured to dump this information via the debug interface (DSU). Other processors provide similar support, e.g. the Nexus Interface for NXP (formerly Freescale) or the Coresight for ARM.

Multiple execution paths. In this work we derive a single SurApp per application. In general, SurApp will build on top of existing tools like RapiTime [91]. The latter will derive the set of worst-case paths for the program on which SurApp will be generated. This does not change SurApp application process, just requires selecting the input vector(s) that trigger the desired execution paths when running the application for trace collection. We also focus on the main memory path (core-cache-memory) and leave as future work I/O paths. We do not expect relevant changes in the SurApp application other than taking care of the address ranges of every access to target the desired I/O devices.

6.3 Surrogate Application Generator

The SurApp Generator (SurAppGen) produces the code of a SurApp as described in the input parameters passed to it. These parameters are $ldSDV$ and $stSDV$; $ld\_iter[]$ and $st\_iter[]$; and $icount$. The latter is the instruction count of the real application. The former two parameters are the load and store SDVs as described in the previous section. Finally, $ld\_iter[]$ and $st\_iter[]$ describe the number of iterations to carry out the SurApp to reach the desired accesses per stack distance as described in $ldSDV$ and $stSDV$.

The main data structure of the SurApp is a vector ($dvec[]$) of size $dvec\_size = (w+1) \times wsize$, where w is the number of ways and $wsize$ the size of a cache way. Hence $dvec[]$ can be seen as having $w+1$ *chunks* of $wsize$ bytes, e.g., for a $32KB$ 4-way cache there are 5 chunks with $dvec[]$ having a total size of $dvec\_size = (4+1) \times 8KB = 40KB$. SurApp accesses this vector appropriately so that those accesses match the *sdki* described in $ldSDV$ and $stSDV$. Note that we have a $dvec[]$ vector for loads and another one for stores.

SurApp generates activity in the L2 via data accesses, which are easier to control than instruction accesses. On the other hand, its code is specifically designed to fit in the IL1 cache so that it does not create uncontrolled interferences.

Broadly speaking, a SurApp comprises a main loop that iterates $w+1$ times, traversing totally or partially *dvec*[]. In each iteration i, all accesses, both loads and stores, occur with a given stack distance – generated as respectively described by $ldSDV$ and $stSDV$, see Alg. 1. To that end, the `operation_block()` function is called, first to produce write (store) operations and then reads (loads). Hence, different code is generated for each stack distance.

Algorithm 1 Baseline structure of the generated SurApp.

```
1: procedure EXECUTE
2:     for (i = 0; i ≤ w; i++;) do
3:         operation_block(i, ld_iter[i], load, nopcount);
4:         operation_block(i, st_iter[i], store, nopcount);
```

As input parameter `operation_block()` gets the stack distance of the accesses to be generated, the number of iterations to perform to reach the desired *sdki*, whether the type of operations to generate are loads or stores, and the number of *nop* operations to generate (see Alg. 2).

Algorithm 2 Code executed for each stack distance

```
1: procedure OPERATION_BLOCK(sd,iter,type, ncount)
2:     for (i = 0; i < iter; i++;) do
3:         index = 0
4:         for (j = 0; j ≤ sd; j++;) do
5:             for (k = 0; k < w_size/cls; k++;) do
6:                 if (type == store) then store_ops(index);
7:                 else load_ops(index);
8:                 nop_ops(ncount);
```

In the inner loop of `operation_block()`, we access a chunk (i.e. cache way) of *dvec*[]. In particular, we access each line of the chunk once. Hence, the two inner loops traverse $sd+1$ chunks of *dvec*[]. Since each chunk occupies a complete cache way, this results in $(sd+1) \times wsize/cls$ accesses with a stack distance sd, where cls stands for cache line size. For instance, by continuously traversing the first $1 \times wsize$ bytes of *dvec*[], all accesses (except those of the first traversal) will have stack distance 0. Likewise if we traverse several times the first $2 \times wsize$ of *dvec*[], accesses starting from the second traversal have stack distance 1. And so on and so forth. Hence, by smartly selecting the part of *dvec*[] accessed we force accesses to hit/miss in a desired cache level and have specific stack distances in [0,w].

The outer loop iterates several times to match the desired sdki. For a given stack distance k and operation type, e.g. load, the number of accesses to carry out is ld_sdki_k

so the number of iterations to carry out is $iter = ld\_sdki_k/(wsize * (k+1))$. For each stack distance this value is stored in the $ld\_iter[]$ and $sd\_iter[]$ vectors for load and store operations respectively. In the outer loop the `reset()` function properly resets the access pointer to $dvec[]$.

The function `operation_block()` also generates non-memory operations so that the SurApp generates the same instruction count and *apki* as the real application. Note that $apki = \sum_{i=0}^{w}(ld\_sdki_i + st\_sdki_i)$. The number of nop operations to generate is given by $1000 - apki$. That is, every 1000 instructions, all the instructions that are not memory operations are nops. In the inner loop, for each memory operation generated the number of nops to generate is given by $ncount = (1000 - apki)/apki$.

When the memory operations to be generated are loads, the body of $memory\_operation(addr)$ is as shown in Alg. 3. Basically, the content of the desired position of $dvec[]$ is loaded into a dummy variable. Then a simple control operation computes the next address to access. We make that in every traversal and each access goes to a different cache line. Hence, we set *stride* to the size of a cache line (cls) divided by the size (in bytes) of each element of $dvec[]$. For stores the body of $memory\_operation(addr)$ just requires that a value is stored to the desired $dvec[]$ position.

Algorithm 3 Algorithm for memory operations with loads

```
1: procedure LOAD_OPS(index)
2:     dummy = dvec[index];
3:     index+ = stride;
```

Finally, the code of `core-operation()` is given in Alg. 4. It basically executes $Nnops$ nop operations to force a given instruction count and *apki* as described above.

Algorithm 4 Algorithm for core operations

```
1: procedure NOP_OPS(Nnops)
2:     for (i = 0; i < Nnops; i++;) do nop;
```

Other relevant aspects. For sake of clarity we have encapsulated all code in functions, which can be however inlined to reduce the overhead of control operations (i.e. `call` and `return`). Also loops can be simply removed unrolling the body as many times as needed – keeping in mind the restriction that the SurApp code must fit in IL1. When loops are used, the number of control (core) operations they generate is factored in by the SurAppGen to achieve the desired *sdki* defined in $ldSDV$ and $stSDV$.

We generate SurApps in C to improve portability, though it requires control on compiler flags so that the generated code has the desired behavior. SurApps could also be generated directly in assembly code, which would need however the use of architecture-dependent assembler instructions.

TABLE 6.1: 4-thread workloads used in this work (Benchmarks and Kernel full-names are listed in Figure 6.3)

scua	contenders	scua	contenders	scua	contenders
ep.e	pe.d mp.d me.t	gs.e	g7.e g7.d pg.e	g7.d	pg.e gs.d me.t
me.m	pe.e pg.e ad.e	pe.e	gs.d ep.e me.m	pg.d	me.o jp.d ep.d
ep.d	me.o pe.d ad.e	pg.e	mp.d ep.e g7.e	pe.d	ra.t me.m pg.e
jp.d	ep.d ad.d me.o	g7.e	g7.d mp.d pg.e	ad.e	pe.e ep.e me.o
ra.t	me.m ep.e pe.e	gs.d	g7.d g7.e ra.t	ad.d	gs.d jp.e pe.d
me.o	gs.e pe.d ra.t	mp.d	pe.d me.t gs.d	ob	ob de ob
jp.e	pg.e ep.d me.o	me.t	gs.d ep.e pg.e	de	de de ob

We used as core (i.e. non-memory) operations only nop operations. Our results show that copying the *instruction mix* of the target application provides no increased accuracy, so they are omitted. The instruction mix describes application's percentage of instructions of each type (e.g. INT, BR, FP, ...).

SurApps can be extended to copy the interaction (communication) patterns of the application with the OS and other applications by tracing appropriate software components, which allows capturing system-level effects during early design phases. Its assessment is left as future work.

6.4 Experimental Evaluation

6.4.1 Experimental setup

We use our simulation framework for the NGMP as described in Chapter 2.

Benchmarks. We use the MediaBench benchmarks as well as the Space kernels presented in Chapter 2.

Workloads. We create 4-thread workloads with one task under analysis (*scua*) and 3 arbitrary contenders, see Table 6.1. As *scua* we use each of the 19 MediaBench benchmarks. Contenders are those whose cache behavior we mimic with SurApps. The *scua* runs against contender tasks and their SurApps in separate experiments, so that we can compare *scua*'s performance in each case. A similar approach is followed for Space kernels: in this case, we run each kernel against mixes with both kernels, also shown in the bottom-right corner in Table 6.1. For these workloads we make a detailed analysis (see metrics section). We also provide results for a set of experiments comprising more than 17,000 randomly (non-repeated) generated workloads including all types of benchmarks.

Metrics. We use different metrics to assess accuracy in the slowdown caused in other tasks.

Bus and Memory inaccuracy (bbi and mbi) compare the number of bus and memory accesses performed by the real application and its surrogate. They are respectively computed as: $bbi = |1 - n^{sur}_{bus}/n^{real}_{bus}|$ and $mbi = |1 - n^{sur}_{mem}/n^{real}_{mem}|$, where n^{sur}_{bus} and n^{sur}_{mem} are the number of bus and memory accesses made by the SurApp, and n^{real}_{bus} and n^{real}_{mem} the counterpart values for the real application. bbi and mbi are 0 when the SurApp matches the access count of the real application. Both metrics can get arbitrarily large as SurApp results differ from those of the real application.

Cache behavior inaccuracy (cbi) assesses the accuracy of the SurApp copying the behavior of real applications. It compares the SDV of the real application and the SurApp as shown in Eq. 6.1, where $sdki^{real}_i$ and $sdki^{sur}_i$ stand for the $sdki_i$ of the real application and SurApp. Note that this is done for both loads and stores, e.g. $ldSDV$ and $stSDV$.

$$cbi = \frac{\sum_{i=0}^{w} |sdki^{real}_i - sdki^{sur}_i|}{\sum_{i=0}^{w} sdki^{real}_i} \tag{6.1}$$

cbi accumulates the deviation of all stack distances between the application and its surrogate. For instance, if $ldSDV^{real} = [20, 30, 50]$ and $ldSDV^{sur} = [25, 30, 60]$, and $stSDV^{real} = [40, 40, 20]$ and $stSDV^{sur} = [35, 30, 20]$ then $cbi = (5 + 0 + 10 + 5 + 10 + 0)/(200) = 0.15$. *cbi* is zero when the SurApp matches the behavior of the real application. It can be arbitrarily large as the accesses per stack distance of the SurApp differ from those of the real application.

Multicore inaccuracy compares the execution time (ET) of the *scua* when it runs against the real contender applications and when it runs against the contender's SurApps. It is computed as $scuai_{ET} = ET^{sur}_{scua}/ET^{real}_{scua}$ where ET^{real}_{scua} and ET^{sur}_{scua} are the execution time of the *scua* against the real contenders and their surrogates respectively. If both execution times match, then $scuai_{ET} = 1$ meaning that SurApps effectively copy the behavior of contenders. If the execution time of the *scua* when running against the SurApp is bigger than when it runs against the real application then $scuai_{ET} > 1$, and vice-versa.

bbi, *cbi* and *mbi* compare the behavior in isolation of the *scua* and its SurApp, whereas $scuai_{ET}$ compares the impact of contender applications and their SurApps on the *scua*.

SurApp generation methodology. Address traces have been directly dumped from the emulator part of SoCLib, although debug or pin tools can be used in most platforms for that purpose. Traces were processed automatically with Python scripts producing stack distances. Finally, SurApps are generated automatically with Python scripts building on the computed stack distances.

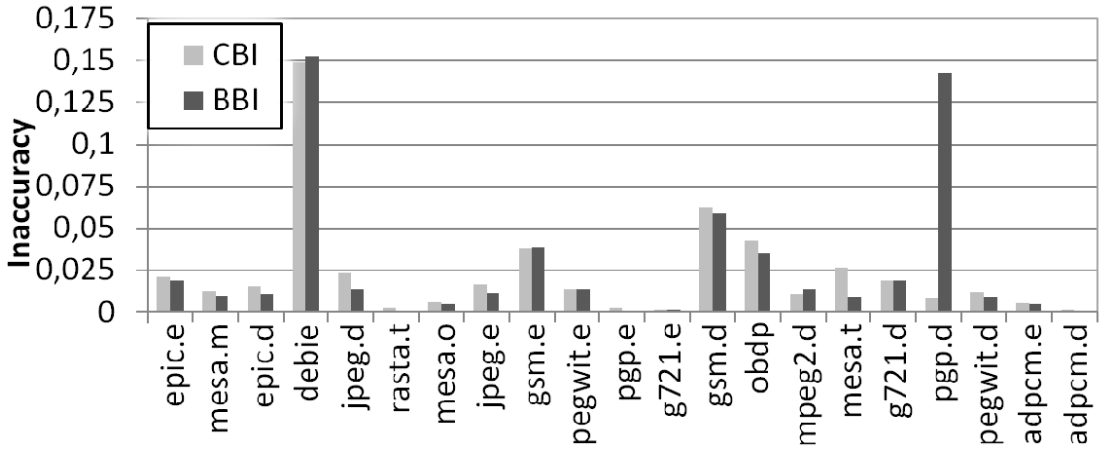


FIGURE 6.3: *cbi* and *bbi* accuracy results

6.4.2 Experimental Results

Behavior in isolation. In Figure 6.3 we observe that SurAppGen generates SurApps that tightly copy the bus and cache behavior of the real application. Observed inaccuracy values are extremely low (close to 0). Just *pgp.d* and *debie* have higher *bbi*, and *cbi* also in the case of *debie*, but they are still very low. This is due to the fact that *pgp.d* and *debie* space kernel sizes are smaller in cache usage and duration than the rest of the benchmarks so relative errors are higher.

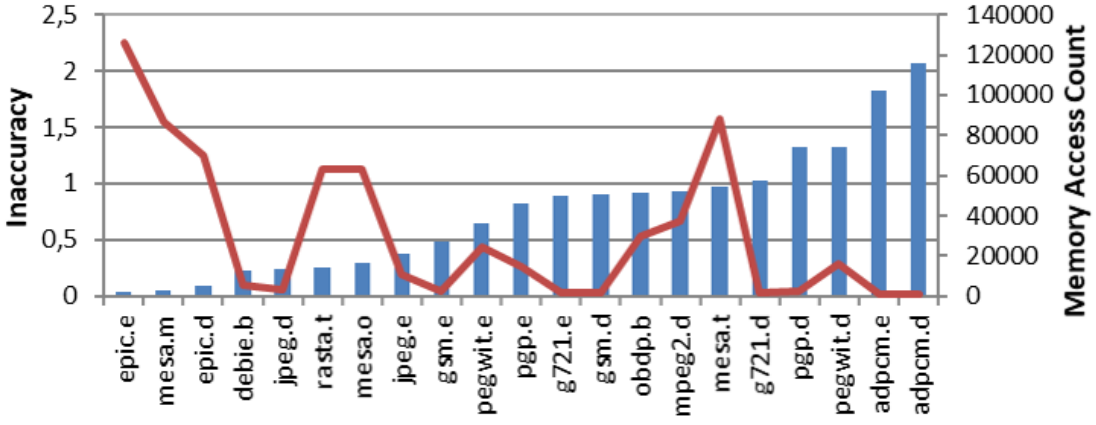


FIGURE 6.4: *mbi* (blue columns) and correlation to mem. access count n_{mem}^{real} (red line)

In Figure 6.4 we observe this phenomenon at a higher scale for *mbi*. Benchmarks are sorted based on their *mbi*. We observe that benchmarks with high *mbi* (blue columns) often have low memory access counts (red line). These discrepancies relate to initialization effects of SurApps, which introduce few additional memory accesses. Those accesses have negligible impact in absolute terms, but may have large impact in relative terms by increasing *mbi* when the total number of memory accesses of the application is low. For instance, *mbi* for *adpcm.d* is around 2, thus reflecting that its SurApp triples its number of memory accesses. However, this benchmark accesses memory once every 20,000

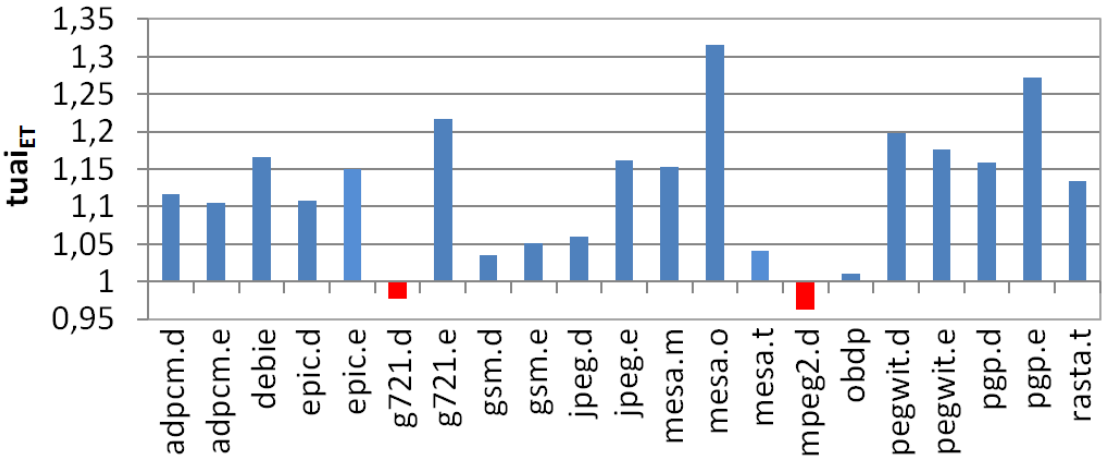


FIGURE 6.5: Multicore Execution Time Inaccuracy of the *scua* when executed in the workloads shown in Table 6.1 (against real contenders and their SurApps)

instructions on average and its SurApp just 3 times every 20,000 instructions. Hence, an additional memory access every 10,000 instructions has negligible impact in absolute terms and *mbi* is irrelevant for low memory access counts.

Multicore behavior. Figure 6.5 shows that the slowdown the *scua* suffers when run against 3 contenders is quite close to the one it suffers when run against the SurApps of those contenders. On the one hand, during early design phase, no strict guarantees on WCET estimate accuracy are required. Instead the aim is achieving good approximations to the real WCET. Since contention slowdown can be as high as 5.5x for some real benchmarks [26], maximum inaccuracy of 30% and 13% on average are highly accurate results. On the other hand, for almost all benchmarks predictions over-estimate contention impact. Under-estimates are few and very limited (down to 0.96). Over-estimation is preferred, since under-estimation could result in applications at late design phase not fitting their predicted budget, causing significant costs: either the integrator has to change the schedule granting more budget to tasks missing their deadlines, or the software providers are required to change their applications to fit their assigned budget. Still, results show that contention impact is under-estimated only occasionally and slightly. This behavior is expected since SurApps cause slightly higher miss counts with shorter survival times for data in cache than the original applications, thus leading to higher contention. Also, the fact that most applications scheduled have over-estimated time requirements allows compensating for those few cases where some slight underestimation occurs.

Absolute Contention Slowdown. Table 6.2 shows the slowdown the *scua* suffers when running with 3 other programs (either MediaBench or Space kernels), and the slowdown it suffers when running against their SurApps. Benchmarks are sorted by the actual slowdown suffered when run against other benchmarks. Slowdowns span from

TABLE 6.2: *scua* slowdown when it is executed against other benchmarks and their SurApp. Space kernels in italics. Programs sorted from lowest to highest RealApp slowdown

	ep.e	gs.d	gs.e	*ob*	g7.e	g7.d	jp.d	ep.d	me.t	mp.d	pe.e
RealApp	1.02	1.03	1.04	1.05	1.08	1.08	1.09	1.09	1.11	1.11	1.14
SurApp	1.18	1.06	1.09	1.05	1.32	1.06	1.15	1.21	1.16	1.07	1.34

	me.m	ad.d	me.o	pg.e	ra.t	pg.d	*de*	pe.d	jp.e	ad.e	
RealApp	1.18	1.19	1.20	1.20	1.21	1.23	1.23	1.25	1.26	1.29	
SurApp	1.36	1.33	1.58	1.53	1.37	1.43	1.43	1.50	1.46	1.42	

	0%	1%	10%	25%	50%	75%	90%	97%	100%
tuaiET ALL	0,82	0,95	1,01	1,05	1,11	1,21	1,39	1,72	2,99
tuaiET Space	0,98	1,00	1,02	1,05	1,09	1,12	1,14	1,17	1,24

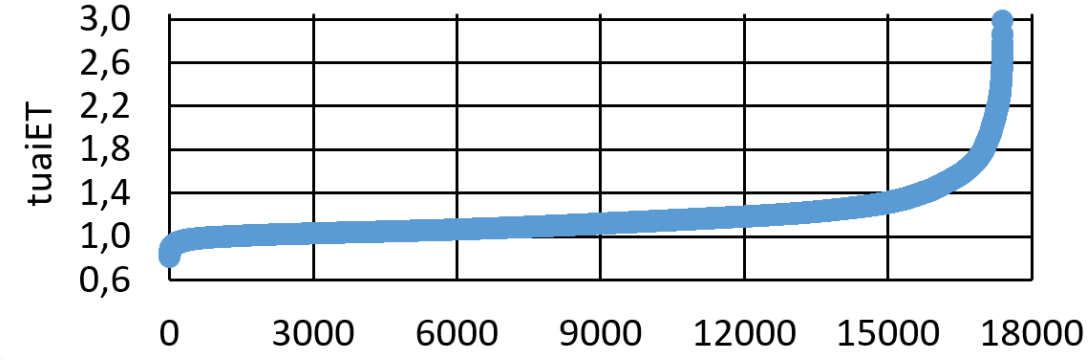


FIGURE 6.6: Inaccuracy Results

1.02 to 1.29, so SurApps tightly mimic real applications regardless of whether they are insensitive or sensitive to contention.

Wide experiment set. Figure 6.6 shows the inaccuracy results we obtain over a wide set of +17000 workloads, together with percentiles. We observe that for less than 1% of the workloads the inaccuracy ranges from 0.82 to 0.95. Likewise, on the upperside of the tail, for less than 3% of the workloads the accuracy is above 1.72. The latter occurs due to the fact that stack distances in SurApps are traversed in blocks, being the first block that for cache misses. Hence, whenever the *scua* has much lower execution time than contenders, it is exposed to the part of the SurApp producing highest interference, thus over-estimating impact of contention. Interleaving stack distance accesses in SurApps to mitigate these corner cases is part of our future work. Still, the presented technique shows to be very robust, with slight tendency towards over-estimation. When space applications are used as *scua*, $scuai_{ET}$ is much lower, see last row in the top table in Figure 6.6.

6.5 Related Works

In [99] authors focus on the case in which each supplier is not provided a board to develop its applications but a virtualized environment (i.e. virtual machine). The latter, which allows checking the functional behavior of applications, is extended to provide timing estimates factoring in multicore contention. In particular, authors propose a trace-driven model to predict the contention that tasks will produce.

For single-cores, other techniques are applied before the binary is written. Those works pursue the goal of providing the developer knowledge of the worst-case "as the code is written" [98]. Some works integrate timing in high level modelling environments such as Matlab/Simulink [97]. Some authors [102] propose a C-source-level abstract machine that is calibrated based on measurements to match a target real hardware. In this line, [100] proposes the *timing model code level* that combines measurements and a regression model to perform timing estimates of source code.

All previous approaches perform some type of modelling, either contention or per-instruction. In this work instead of contention models, we design surrogate applications that run on the target multicore to get a tight estimate of cache contention without the need to share applications.

Traffic generators either software, e.g micro-benchmarks [26], or hardware, e.g. built in the architecture as for the Zynq UltraScale +EG, usually aim at creating high stressful scenarios, whereas SurApps aim at copying the load on the shared resources of a given target application. Further, the particular pattern of accesses generated by those generators is arbitrary, whereas SurApps build on stack distances to mimic the pattern of accesses of the application, thus creating necessarily different effects.

Several approaches profile application accesses to different resources to optimize metrics such as performance or energy efficiency by, for instance, applying cache way locking and page colouring [103] to 'hot' segments. Our technique shares the fact that we also profile applications. However, the main novelty of our approach is that we automatically generate an application that mimics the target ones.

6.6 Conclusions

We introduced the concept of Surrogate Applications (SurApp) to copy the timing behavior of a given real (target) application. SurApps can be shared during early design phase without IP constraints, enabling software providers to run their applications against other software provider's SurApps in the multicore and derive the slowdown

their applications suffer due to contention. We show how SurApps are automatically generated from the stack distances of the target application on the NGMP processor, which we identify as the most relevant parameter to mimic, which is obtained running applications in isolation. Our results show that our automatic generator tightly copies the contention behavior of real applications. The observed accuracy of our approach, in terms of slowdown of the task under analysis, is high, providing evidence of the effectiveness of the SurApp approach on the NGMP processor.

Chapter 7

The ARM big.LITTLE architecture: the Juno Board and DragonBoard

Recently, some COTS processors from the consumer electronics domain have been considered for the implementation of critical embedded systems. In particular, the ARM big.LITTLE architecture, popular in many smartphones, has been considered for the implementation of automotive systems, as it is the case of the Renesas R-Car H3 platform [104]. However, evidence for certification on this platform has only reached a modest ASIL-B[1], thus lacking evidence of its readiness for the highest integrity levels (ASIL-C and D).

In this Chapter we make a step towards obtaining evidence of the timing guarantees that can be reached with ARM big.LITTLE architectures in critical embedded systems to deliver high guaranteed performance for critical real-time functionalities. In particular, we assess qualitatively and quantitatively to what extent these multicore platforms are resilient to the integration of multiple functionalities running simultaneously in different cores. We also identify the particular uses of this platform that may make timing guarantees fragile, thus challenging their use for the most stringent safety integrity levels. For that work, we build upon analysis of the specifications and empirical evidence with stressing benchmarks (aka microbenchmarks) [22] on top of a Qualcomm DragonBoard and an ARM Juno R2 board. The former is a commercial implementation of the ARM big.LITTLE architecture used in smartphones, whereas the latter is a development board implementing the same architecture.

[1]Safety-related systems in the automotive domain are described in ISO26262 [7], where safety-related systems are classified into different Automotive Safety Integrity Levels (ASIL), from A to D, being ASIL-D the most stringent category.

In particular, our assessment of the ARM big.LITTLE architecture on both implementations consists of the following steps:

1. We review the processor specifications and identify some key features for multicore contention analysis. Our analysis identifies how some features need to be configured, reveals missing detail information in the specification and provides hints on what specific elements need to be assessed quantitatively.

2. We make an attempt to tailor the methodology based on *signatures and templates* (see Chapter 5) to both ARM big.LITTLE processors. Concretely, we make an attempt to estimate the contention that an access to a shared resource may experience, which is a mandatory input for applying signatures and templates.

3. Finally, we perform a quantitative assessment with appropriate microbenchmarks with known expected behavior (see Chapter 4). Our results show that the behavior of several Performance Monitoring Counters (PMCs) is non-obvious and hard to correlate with experiments for the SnapDragon 810 processor (the one in the DragonBoard), thus defeating its use to model multicore contention tightly. Moreover, our empirical analysis reveals that documentation is erroneous in some critical elements. On the other hand, our analysis of the ARM Juno SoC (the one in the Juno Board) provides more reliable information since hardware, at least, behaves as expected according to the (scarce) documentation available.

Our analysis reveals that, while documentation and software support for commercial implementations is too scarce for their practical use in critical embedded systems, the architecture behind (ARM big.LITTLE) can be considered for critical embedded systems if detailed information is made available. Our results offer valuable evidence on the appropriate uses of the ARM big.LITTLE architecture in critical embedded systems. In particular, our results show that the cache hierarchy is a troublesome component if not used appropriately, thus making timing behavior highly volatile. Our analysis shows that the shared second level (L2) cache is a key resource challenging WCET estimation of critical real-time tasks due to the virtually uncontrollable interferences that tasks may suffer in L2, thus calling for integrations where the L2 is not effectively used by critical real-time tasks.

7.1 Goal and Scenario

The goal of this Chapter is assessing whether the ARM big.LITTLE architecture can be used in the context of critical real-time applications. We use the term critical real-time

to refer to any hardware or software component with any time criticality need: either mission, business or safety related.

7.1.1 Tracing and Events

We focus on measurement-based timing analysis (MBTA), widely used in most real-time domains. For instance MBTA is used in avionics systems [17, 105], including those with DAL-A safety requirements [106] (though on top of much simpler single-core processors).

In the context of MBTA, tracing events impacting shared resource contention, e.g. cache misses, has been shown fundamental to derive bounds for a task not factoring in the worst potential contention but a specific contention level [20]. It follows that MBTA techniques demand more and more advanced hardware tracing mechanisms.

For that purpose we build upon the existence of PMCs to derive the type and number of accesses each task does, since this is needed to account for the contention a task can experience from (or produce on) others [20]. We also build upon microbenchmarks, i.e. small user level applications, that are able to create very high contention with each access type to the target shared resource [22]. Note, however, that the number of Performance Monitoring Counters (PMCs) in the PMU is limited, so we cannot monitor all events in a single run. Instead, each experiment needed to be repeated twice with different event-to-PMC mappings to obtain all measurements. The experimental methodology we use is described later in Section 7.5.1.

7.1.2 The Platform

Next, we provide the most relevant details of the ARM big.LITTLE architecture implementations for this work. The Juno board includes the Juno SoC, whose general-purpose components are depicted in Figure 2.4. This ARM big.LITTLE design includes two computing clusters, being one of them equipped with 2 Cortex-A72 high-performance cores and another with 4 Cortex-A53 low-power cores. We refer to those clusters as *HPclus* and *LPclus* for short. Each cluster includes a local shared L2 cache and both clusters are connected to a shared memory controller. The SnapDragon 810 processor, included in the DragonBoard, has a similar design, with the following difference: instead of implementing 2 Cortex-A72 cores in the HPclus, it implements 4 Cortex-A57 cores.

A72 and A57 cores feature out-of-order execution, whereas A53 ones offer low-power in-order execution instead. For the sake of facilitating the interpretation of results, the assessment in this work will focus on A53 cores, so the LPclus, to discount the

measurement noise that out-of-order execution could introduce instead. Yet, conclusions reached in this work, as shown later, apply to both clusters.

As detailed in Chapter 2, each core includes a first level instruction (IL1) and data (DL1) cache. Also, each cluster includes a shared L2 cache. In order to assess the impact of shared resources on execution time we stress caches with increasing data sizes. The size of IL1 caches do not impact results since benchmarks tested are designed to be tiny ($<$ 1KB) in comparison to the IL1 size (32KB for LPclus and 48KB for HPclus for the Juno SoC). In the case of the Juno SoC, data cache sizes for the LPclus are 32KB 4-way 64B/line for DL1 and 1MB 16-way 64B/line for the L2, whereas for the HPclus sizes are 32KB 2-way 32B/line and 2MB 16-way 64B/line respectively. The SnapDragon 810 processor has some differences in cache sizes. In particular, DL1 and L2 for the LPclus are 64KB and 512KB instead of 32KB and 1MB.

The particular interconnect between cores and L2 caches is not described in the documentation. L2 caches are connected to the memory controller through an ARM AMBA 4 bus (ARM CoreLink CCI-400 Cache Coherent Interconnect). Other components, such as accelerators and peripherals (not depicted in Figure 2.4), are also attached to this bus. Since they are not used in our analysis, we omit details and keep them disabled or idle during test campaigns.

7.2 Qualitative Analysis of the ARM big.LITTLE Architecture: Specifications

The main source of information for the analysis of the SnapDragon 810 processor and the Juno SoC is the ARM Cortex-A53 processor technical reference manual [107]. As detailed in the manual, a number of A53 features are regarded as 'implementation dependent', thus meaning that the processor manufacturer has the flexibility to choose among different options available. For instance, this is the case of the DL1, IL1 and L2 cache sizes. From the information available in the A53 manual, we regard as particularly relevant for contention analysis the following:

- The arrangement of the main components in the A53 cluster, including DL1, IL1 and L2 caches, as well as data prefetching features in DL1 and coherence support in L2.
- PMCs for events occurring in the cores (e.g. DL1 and IL1 caches) and in the L2 cache.

However, some parameters are not available in the A53 manual, including the following:

1. Timing characteristics of the interconnect between DL1/IL1 and L2 caches.
2. Specific characteristics of the different cache memories such as, for instance, their sizes.
3. PMCs for events spanning beyond the A53 cluster such as accesses to the bus connecting A53 and A57 clusters with memory, and PMCs for the memory controllers.

From a detailed analysis of each of those missing parameters for real-time purposes in the A53 manual, we reached the following conclusions:

- The interconnect between DL1/IL1 and L2 caches, as the remaining in-cluster components, should be documented in ARM manuals. The lack of that information in the manuals makes us resort to software testing (e.g. microbenchmarks) to bring some light on the characteristics of this interconnect.
- Some instructions exist to read the particular characteristics of the cache hierarchy so they can be directly retrieved from the platform itself.
- PMCs and events beyond the A53 cluster should be documented in the SnapDragon 810 manual, since the processor manufacturer integrates those components, and so has access to the appropriate information for each component.

By the time we performed this work, ARM manuals were available, so we could retrieve them[2]. However, SnapDragon 810 manuals are neither publicly available in Qualcomm's website, nor included in the documentation coming along with the DragonBoard (whose processor is the SnapDragon 810), nor obtainable upon request. In particular, while we requested appropriate manuals through Qualcomm public services as well as through internal contacts, and NDAs are in place, we were unable to get access to them. We are also aware of other companies in the critical real-time domain have experienced similar issues. Therefore, to the best of our knowledge, no information has been obtained on what PMCs/events exist beyond the A53 cluster and how they could be used. We also tried to use information from the ARM Juno development board, which is an ARM big.LITTLE implementation by ARM instead by Qualcomm that offers further documentation but, as we suspected, ARM and Qualcomm implementations of this architecture differ and so Juno documentation did not help on reaching conclusions that apply to all ARM big.LITTLE implementations. From our analysis of the information available, we have reached the following conclusions:

[2]They have later become unavailable online and can only be retrieved upon request to ARM.

- The interconnect between DL1/IL1 and L2 can only be analysed empirically without specific guidance on its timing behavior. The confidence on those measurements is limited due to the unknown specification of the interconnect.
- DL1, IL1 and L2 features can be directly obtained from the board via control instructions.
- Specific instructions exist to disable the data prefetcher. This is particularly relevant to discount uncontrolled (prefetcher) effects during operation.
- The L2 is inclusive with DL1 for coherence purposes. Thus, one core can create interferences on the DL1 of other cores by evicting their data from the L2 cache.
- The L2 cache cannot be partitioned across cores. This feature, together with L2 cache inclusivity, leads to potentially abundant inter-core interferences if not controlled by software means.
- PMCs up to the L2 cache exist and are abundant, but no information is had about PMCs beyond the L2 cache.

Overall, several cache features challenge the calculation of inter-core interference execution time bounds, and the lack of documentation for the DL1/IL1-L2 interconnect and PMCs for events beyond L2 challenge the confidence that can be obtained on measurement-based bounds. However, some information about contention can still be retrieved empirically based on information available. In the next sections, we present the results obtained.

7.3 Quantitative Analysis of the SnapDragon 810 Processor

The number of hardware events that can be monitored in the SnapDragon 810 processor is limited according to ARM's documentation. For instance, while cache and memory accesses can be counted, it cannot derived whether DL1/IL1 and L2 cache accesses turn out to be hits or misses. This complicates the development of our methodology to measure the impact of contention in the access to shared resources.

7.3.1 Microbenchmarks

In order to access PMCs, we have developed a library with an interface to read/write PMCs. The main functions of the library include resetting/setting PMCs, activating/stopping PMCs, read/write PMCs and start/stop the Performance Monitoring Unit.

LISTING 7.1: Structure of a microbenchmark

```
R1 = 0;
for (i=0; i<N; i++) {
    reset PMCs;
    for (j=0; j<M; j++) {
        R2 = Load [@A+R1]; R1 = R1+STRIDE;
        R2 = Load [@A+R1]; R1 = R1+STRIDE;
        ...
        R2 = Load [@A+R1]; R1 = R1+STRIDE;
    }
    read PMCs;
}
```

To quantify the impact of contention in the access to the different shared resources, we have developed several microbenchmarks that stress each specific resource separately, in line with the method exposed in Chapter 4. This allows estimating the maximum delay that a request to a particular shared resource can suffer. Then this data is used to upper-bound contention impact. As starting point, we have developed microbenchmarks to account for contention in the access to the shared L2 cache and to the shared memory controller, see their structure in Listing 7.1.

Since measurement can be polluted, e.g. by the Linux OS running below, we collect several (N) measurements and remove outliers keeping only the mode. The iterator M and the number of LOAD operations in the loop are set to values sufficiently high so that the overhead of the loop (i.e. the control instruction) and the overhead to fill the IL1 cache become negligible (e.g. $M = 1000$ and 16 LOAD operations). The particular PMCs/events read and reset depend on the contention that is to be measured in a particular experiment. Finally, STRIDE relates to the distance between memory objects accessed so as to make sure that they either hit in L1, miss in L1 and hit in L2, or miss in L1 and L2. Vector size is properly set also with the same goal.

7.3.2 Disabling the Data Prefetcher

We disabled the data prefetcher so that read and write operations occurring in the different cache memories are only triggered explicitly by the instructions executed in the microbenchmarks, rather than being automatically generated by hardware. For that purpose, we have configured the `CPUACTLR` register as described in the A53 manual [107]. Unfortunately, the execution of these commands leads to a system crash.

In order to verify the source of the problem, we repeated the same experiment on a PINE A64 [108] board. The PINE A64 platform is built with the aim of being a low-cost open source platform. Its processor, Allwinner A64 chip, implements the same Quad-Core A53 Processor as the low-power SnapDragon 810 cluster. Thus, its interface is expected to be the same. In the PINE A64 platform, the commands to disable the

prefetcher worked properly and subsequent experiments revealed that the data prefetcher was effectively disabled on that board. However, such board is a low-cost and low-power general-purpose computer, so the board itself is not oriented to the industry in the mobile market. Instead, it is an open platform. Thus, mobile industry will unlikely use it since there is no a large enterprise that provides support in the long term.

Overall, we could not disable the prefetcher in the SnapDragon 810. This problem likely relates to potential modifications introduced by the processor manufacturer, from which we did not succeed in obtaining the information required about the SnapDragon 810.

As a confirmatory experiment, we run a microbenchmark accessing 88KB of data, thus exceeding DL1 capacity (64KB) but fitting L2, with a 8B stride. Hence every 8 accesses we have 1 DL1 miss and 7 DL1 hits due to spatial locality (DL1 line size is 64B). With the prefetcher disabled, we would expect that the number of L2 accesses was 1/8 those in DL1. We observed that the number of DL1 accesses matches quite well our expectations, but the number of L2 accesses is roughly 0, revealing that the prefetcher is active and fetches data into DL1 reducing L2 accesses (the PMC for prefetch requests confirms this hypothesis).

7.3.3 Assessing Microbenchmark Results

In order to assess the behavior of the PMCs in the A53 cluster, we have run our microbenchmark, which performs 11,000 load operations with a specific stride. This code is in a loop iterating 100 times, and we report average results across those 100 iterations to minimize the impact of cold misses in the first iteration and noise in the measurements.

We explore strides ranging between one 64-bit element (8 bytes) and 512 elements. With the smallest stride (8 bytes), we traverse a vector of $\approx 88KB$ (11,000 elements x 8 bytes), which does not fit in DL1, but it does in L2. Thus, the number of DL1 accesses expected is 11,000 approximately. Each load is expected to miss in DL1 when 64B boundaries (DL1 cache line size) are crossed, and should hit in DL1 otherwise.

Overall, for a 1-element stride we expect 1,375 (11,000/8) L2 accesses per data vector traversal. Then, since $\approx 88KB$ fit in L2, we expect roughly 0 memory accesses (13.75 in practice on average). When doubling the stride (so with a data vector of $176KB$), we expect L2 accesses to double until reaching value 11,000 (at stride 8), and then flatten. Memory accesses should remain roughly 0 until L2 cache capacity (512KB) is exceeded, at stride 8 ($\approx 704KB$), when all accesses become also L2 misses so we have 11,000 DL1, L2 and memory accesses.

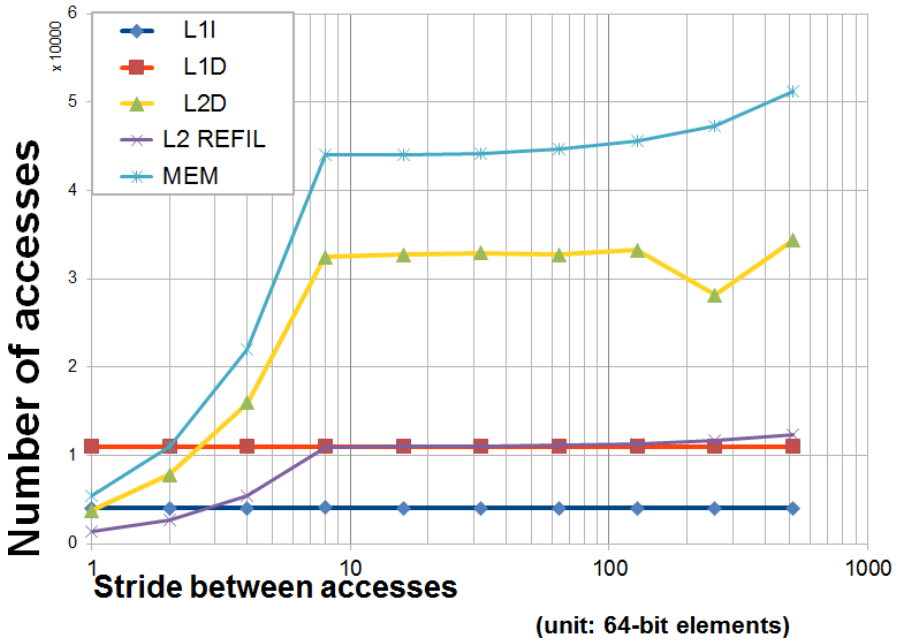


FIGURE 7.1: Avg. number of IL1 (L1I), DL1 (L1D), L2 (L2D) and memory (MEM) accesses, and L2 refills per loop iteration for different data strides (©2018 IEEE).

Figure 7.1 shows how DL1 accesses effectively match expectations while L2 accesses (L2D in the plot) show much higher values. Interestingly, L2 refills (`L2 REFIL`), i.e. lines brought explicitly on a DL1 miss, match our expectation for L2 accesses. This reveals that, apart from the DL1 misses, we have another source of L2 accesses, which seems to be the prefetcher. When looking at the number of memory accesses (MEM), we observe that it matches quite accurately L2 accesses plus L2 refills, thus reflecting a number of accesses largely above expectations. This reveals interferences from the prefetcher since, even when data should fit in L2 (up to stride 8) and so memory accesses should be negligible, we have plenty of them. Overall, this experiment reveals that the prefetcher is active and produces severe interferences that defeat any intent to control contention in shared resources in the A53 cluster.

7.4 Summary of Lessons Learned for the SnapDragon 810 Processor

In this chapter we analysed the difficulties entailed by using a popular microprocessor in consumer electronics, the SnapDragon 810, in the context of critical real-time applications. This microprocessor provides the level of performance needed by many critical real-time applications, but at the same time poses a number of challenges in its utilization, which we summarize next.

Uncontrolled resource sharing. The use of a fully-shared L2 cache across several cores poses some difficulties to control or tightly upper-bound inter-core interferences. In particular, one task running in one core is allowed to evict any line in the L2 cache, thus affecting the performance of other cores in non-obvious ways. This issue may be exacerbated by the fact that the L2 cache of this processor is inclusive with DL1 caches. Thus, a task may also get its data evicted from DL1 due to the inclusion property with L2.

The most promising approach to overcome this challenge builds upon cache partitioning. For instance, the Freescale P4080 processor, also representative of a high-performance processor of interest for real-time applications, allows configuring its shared L3 cache so that private regions are allocated to specific cores [109]. However, space partitioning may not be enough if buffers and queues are shared, which may still allow high contention across cores, thus leading to low performance guarantees [110]. However, as shown in this Chapter, some popular processors do not provide such support yet.

Need for documentation. For enabling MBTA based on PMCs, at least some documentation about components interfaces is mandatory. The information on hardware-to-hardware interfaces includes the way in which requests are managed (e.g. whether shared queues are used, what policies are used to serve requests). This allows reasoning about the theoretical worst-case scenarios so that microbenchmarks can be developed to stress them and obtain timing information via measurements.

Regarding software-to-hardware interfaces, which include precise information on how to enable/disable some features (e.g. prefetchers) or how to monitor hardware events through PMCs available, information released is often limited. Again, this prevents appropriate configuration and monitoring of the processor, thus defeating the intent of obtaining tight WCET estimates on top of the SnapDragon 810. The unavailability of this information often relates to IP protection and competition.

Both issues are exacerbated by the fact that many microprocessors incorporate IP from different suppliers, as in the case of the SnapDragon 810 processor, which includes at least IP from ARM and Qualcomm. In our view, detailed information will be made progressively available as market pressure increases and releasing details becomes the only way to make sales grow. Still, this shift towards openness will occur slowly.

7.5 Quantitative Analysis of the Juno SoC

We start presenting the experimental setup and then we provide results identifying pros and cons of this implementation of the ARM big.LITTLE architecture for its use in

Algorithm 5 Structure of a stressing benchmark

```
1: procedure SB_BODY
2:     for (i = 0; i ≤ 1000; i++;) do
3:         reset PMCs;
4:         for (j = 0; j < 2^N/16; j++) do
5:             R2 = Load [@A+R1]; R1 = R1+64B;
6:             R2 = Load [@A+R1]; R1 = R1+64B;
7:             ... (x 16 in total)
8:             R2 = Load [@A+R1]; R1 = R1+64B;
9:         read PMCs;
```

critical embedded systems.

7.5.1 Experimental Setup

To perform measurements, we use stressing benchmarks (SB), simple kernels that traverse a data vector [22]. The size of the vector is set to be 2^NKB and accesses to it are performed with a stride of 64B, thus matching cache line size. PMCs are reset before each traversal and read right after. Each experiment is performed twice reading different events so that all events of interest can be monitored.

Collecting measurements on a real board poses some challenges such as the difficulties brought by the monitoring software to interface the counters and the noise of the interrupts, which may also interfere measurements. Hence, some measurements can be abnormally high or low. To mitigate the impact of noise, traversals are repeated 1,000 times (per set of events read). The pseudo-code of the stressing benchmarks is shown in Algorithm 5, where the main loop is unrolled 16 times to reduce the relative overhead of the loop control instructions. Such an approach produces 1,000 measurements for each event. The first one is intended to warm up caches. Hence, its results can be regarded as irrelevant for our study. Then, to discount outliers we keep the median, which is still subject to some noise. However, filtering outliers automatically with this approach allows identifying trends as needed for this work.

We consider two experimental setups. The first one (see Figure 7.2 (a)) runs the monitored SB (SB_{mon}) in one of the cores of the LPclus. All the remaining cores remain idle. The only exception is one core of the HPclus, which runs the Real-Time Operating System (RTOS), marked with an asterisk. Such activity has been placed in a different cluster to the one being monitored to minimize unwanted interference. The second setup is identical to the first one, but all contender cores run SB also (see Figure 7.2 (b)).

Vector sizes per SB vary between 1KB and 2MB (so $10 \leq N \leq 21$), hence the SB_{mon} may either fit in DL1, exceed DL1 and fit in L2, or exceed L2 cache space. To simplify the discussion, we focus on the case in which N is identical across all SB in all cores.

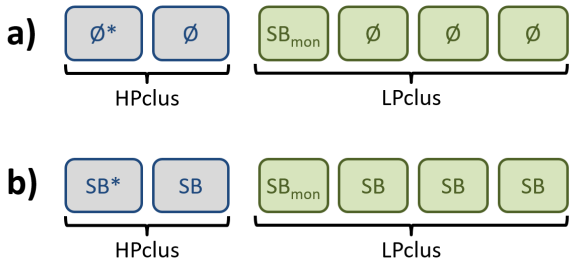


FIGURE 7.2: Experimental setup (a) in isolation and (b) with contention (©2018 IEEE).

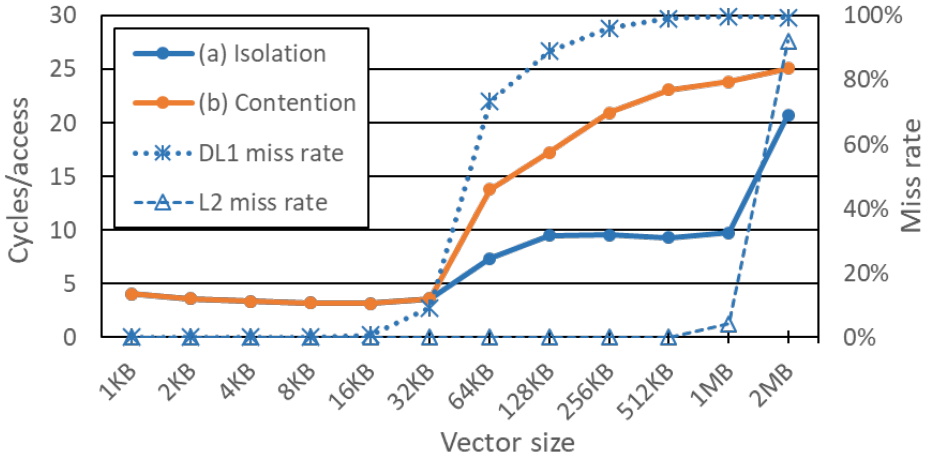


FIGURE 7.3: Cycles per access for the two setups when varying vector size (©2018 IEEE).

7.5.2 Assessing Stressing Benchmark Results

Figure 7.3 presents the results of the experiments in both setups in the form of cycles per (memory) access, or CPA for short (straight lines). Such a metric allows comparing all measurements regardless of the size of the vector. The plot also includes DL1 and L2 misses divided by *the total number of data accesses in the program* for the setup in isolation (dashed and dotted lines). Note that L2 misses are divided by the number of data accesses in the program instead of the number of L2 accesses to better allow reasoning about the impact of L2 misses in the execution time. Instead, L2 miss ratios w.r.t. L2 accesses could hide whether the absolute number of L2 misses is high or low. For instance, if there is only one L2 access, the impact on execution time is irrelevant, but L2misses/L2accesses would be 0% (if hit) or 100% (if miss).

Results in isolation. The blue solid (dark) line shows that the CPA is slightly above 3 cycles when the vector size does not exceed 32KB. Such a vector size fits in DL1 and hence, vector accesses are expected to hit, as reflected in the low DL1 miss rate (dotted line). Note that, since each memory access comes along with an arithmetic operation to increase the index, 3 cycles is expected to be the latency to execute both,

the memory access and the arithmetic operation. We observe that the CPA slowly decreases when moving from 1KB to 16KB. This occurs because the code inside the loop has some prologue and epilogue to set up and read the PMCs. For a larger vector size, the relative impact of such code decreases. We also note that the lowest CPA value is 3.15 for a vector size of 16KB, still above 3 cycles. This occurs because every 16 memory accesses there are few arithmetic instructions to check the loop condition. Also, whenever the vector matches DL1 size exactly (32KB), the CPA increases to 3.62. This occurs because prologue and epilogue fetch few cache lines that cause some DL1 evictions in each iteration and hence, some additional L2 cache accesses. Hence, DL1 miss rate grows from <1% to 9%.

For vector sizes in the range 64KB-1MB, the CPA reaches values slightly above 9 cycles, with the exception of the case for 64KB. In all these cases data does not fit in DL1, but fits in L2, as reflected in DL1 and L2 miss rates (dashed line). As for the case when data fit in DL1, we observe that larger vector sizes slightly decrease the CPA until we reach the exact L2 size (1MB), when CPA slightly increases. The CPA for 64KB is abnormally low. The source of this unexpected value is still under investigation, although it seems to relate, to some extent, to the DL1 miss rate, which is around 73% when we would expect it to be close to 100%. However, we are pessimistic on whether the cause can be identified given the limited documentation available, which omits details on, for instance, whether some form of buffering exists between DL1 and L2, or whether translation lookaside buffers (TLBs) could create further delays.

Finally, for a 2MB vector size, L2 cache space is exceeded and virtually all vector accesses reach memory, thus producing a CPA slightly above 20 cycles. This information is also reflected in the L2 miss rate.

Results with contention. Under contention, setup (b) in Figure 7.2, results for vector sizes between 1KB and 32KB match exactly those for setup (a) in isolation. This is expected since all cores hit in their respective DL1 caches and hence, no contention occurs in shared resources.

For vector sizes between 64KB and 256KB, the vectors of all cores in LPclus still fit in L2 (i.e. $256KB \times 4 \leq 1MB$). Hence, there is no significant contention for L2 space, since only some residual contention due to loop prologue and epilogue code is expected when vector sizes are 256KB. Therefore, the CPA increase w.r.t. the isolation setup can be attributed to contention in the L1-L2 interconnect and serialization of the accesses in L2. As the size of the vector increases, the number of consecutive memory accesses increases and hence, there are fewer non-memory instructions per access (due to loop condition check plus prologue/epilogue), and thus, the degree of contention per access increases. Performing an exhaustive assessment of all potential conditions to discover

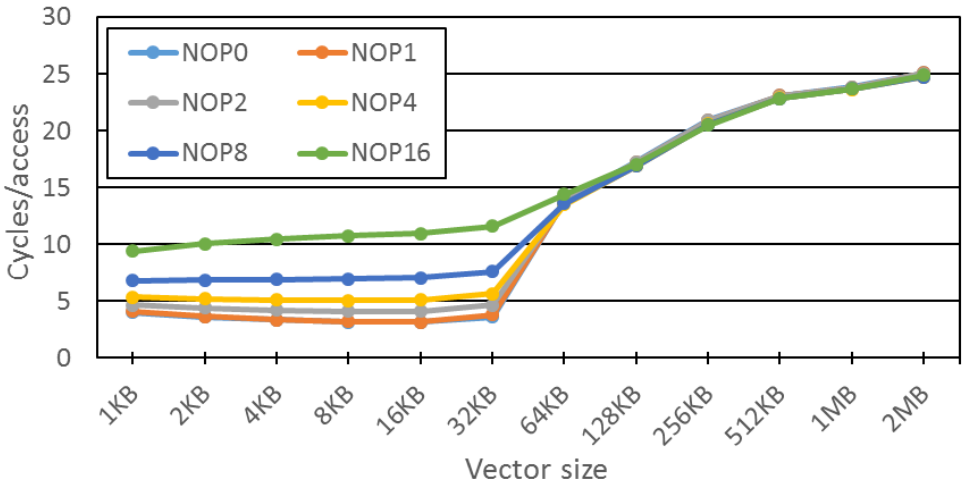


FIGURE 7.4: CPA with contention varying the number of NOPs between accesses (©2018 IEEE).

the maximum CPA is left for future evaluation, although the methodology needed to discover such a value has already been described in [22]. Note, however, that the degree of contention in the access to L2 is so high that, despite data fits in L2 for the 256KB vector size, CPA is 20.9, slightly higher than the CPA of experiments in isolation when L2 cache space is exceeded (20.7).

For vector sizes in the range 512KB-1MB, L2 cache space is exceeded in LPclus, so the CPA becomes 23.1 cycles for 512KB and 23.8 for 1MB. This indicates that moving from a scenario with high contention in the access to L2 to a scenario where L2 cache space is exceeded can only cause a modest CPA increase.

Finally, for a vector size of 2MB, the SB in the HPclus also exceed their L2 cache space, which is 2MB. Hence, the SB_{mon} experiences additional contention in its memory accesses, thus having a CPA slightly above 25 cycles. While such an increase can be noticed, it is also rather modest since DRAM memory is very fast in comparison with the Juno SoC, and contention in the access to L2 proves to be the main performance bottleneck.

Increasing NOP count. For the sake of completeness, we have considered the setup with contention, but placing an increasing number of no-operations (NOP) between memory accesses. Results for NOP counts between 0 (the default case) and 16 are depicted in Figure 7.4. As shown, as the number of NOPs raises, there is an increase in the CPA when data fits in DL1, since NOP latency can be hardly overlapped with DL1 access latency, thus impacting execution time. However, as soon as the vector size exceeds DL1 size (from 64KB onwards), execution time is completely dominated by the contention in the access to shared resources, so even 16 NOPs can be executed between two consecutive memory accesses without further increasing execution time, so

that CPA remains constant. This is reflected in the fact that the CPA is roughly the same regardless the number of NOPs for any vector size equal or higher than 64KB.

7.6 Summary of Lessons Learned for the Juno SoC

Our qualitative and quantitative assessment of the timing behavior of ARM big.LITTLE architectures through the ARM Juno board provides some valuable lessons:

- The variation in terms of latency between L2 hits and L2 misses is large. In our results in isolation they are 9 and 20 cycles respectively. Hence, a task hitting L2 cache often is highly vulnerable to L2 cache space interference, which may increase execution time by a factor above 2x.
- Contention in the access to L2 due to access serialization can be as significant as the impact of transforming L2 hits into L2 misses. In fact, L2 latency without and with contention is also 9 and 20 cycles respectively, hence a factor above 2x.
- Contention accessing DRAM ($\approx$ 25 cycles) is relatively low in comparison to that accessing L2 ($\approx$ 20 cycles), hence a factor around 1.25x.
- Tasks hitting in their local DL1 are quite insensitive to contention. Eventually they might suffer some DL1 evictions due to inclusivity, which are only expected to be significant when the degree of L2 thrashing caused by contenders is high. Unless unfortunate cache placement occurs where the task analysed and its contenders compete for very few L2 cache sets, one might expect that contenders need to thrash the complete L2 cache to evict all DL1 data of the task under analysis. Hence, the smaller the data set, the less frequently DL1 misses due to inclusivity evictions will occur.

7.7 ARM big.LITTLE Comparison

In this section we compare two implementations of the ARM big.LITTLE architecture: the Juno SoC and the SnapDragon 810 processor.

Several characteristics of these processors may enable their use in the context of critical embedded system or simply defeat any meaningful effort. In particular, while our results provide positive feedback on the use of Juno SoC for critical embedded system, our work on the SnapDragon 810 offers opposite conclusions. Next, we review those characteristics

TABLE 7.1: Juno Soc vs SnapDragon 810 comparison.

Feature	**Juno SoC**	**SnapDragon 810**
Prefetcher can be disabled	Yes	No (system crashes)
Documentation is reliable	Yes (no flaw found)	No (some items are wrong, including prefetcher info)
Details on PMCs for activity beyond L2	No	No
L2 cache partitioning	No	No
DL1-L2 inclusivity can be disabled	No	No

comparing both processors, also in Table 7.1 we summarize the main findings of our comparison.

- **Prefetcher**. ARM big.LITTLE architectures have a prefetcher that brings data from memory to L2 caches. We have successfully executed the command for disabling such prefetcher in the Juno SoC. However, such command, which has been shown to work also in other ARM platforms with a A53 quad-core cluster (e.g. the PINE A64 [108]), leads to a system crash when used on the SnapDragon 810 processor. Hence, its prefetcher could not be disabled.

- **Documentation**. A number of parameters in the documentation are often described as *implementation dependent*. Their characteristics can be generally retrieved by executing specific instructions that poll the hardware for those details. However, other parameters are given in the documents such as, for instance, the particular command to disable the prefetcher. In the case of the SnapDragon 810 processor, not only this command does not disable the prefetcher, but it also produces a system crash. The only conclusion we could get is that, as part of the modifications introduced by Qualcomm to optimize power and performance, the prefetcher interface was altered but documentation was not updated. While the Juno SoC has been built completely by ARM, who also delivers the documentation, the SnapDragon 810 combines IP from ARM and Qualcomm, and modifications introduced by Qualcomm are neither available in ARM's documentation nor released in any Qualcomm specification. Therefore, there is not practical view to retrieve such information in the case of the SnapDragon 810 processor.

- **Behavior beyond L2 caches**. ARM's documentation on the big.LITTLE architecture span up to L2 cache memories since they relate to each cluster. However, no public information is provided on the PMCs related to the activity beyond L2 caches. Hence, the timing characterization of the platform brings some uncertainty on its behavior beyond L2 caches. This relates to the fact that bad

performance cases can be triggered, but it is unclear whether those cases are close to the absolute worst-case behavior.

- **L2 cache interference**. ARM big.LITTLE processors do not implement L2 cache partitioning. This holds true for both, the Juno SoC and the SnapDragon 810 processor. Hence, L2 cache space is fully shared by all the cores in the cluster, which can evict each other's data.

- **DL1 cache interference**. While DL1 caches are private for each core, DL1 contents are inclusive with L2, thus meaning that, upon a L2 cache line eviction, if such cache line is present in any DL1 cache of the cores in the cluster, it will be also evicted to preserve inclusivity. This means that L2 cache interference across cores may also cause DL1 cache interference indirectly through the eviction of L2 lines stored also in DL1 caches. This feature is common for both, the Juno SoC and the SnapDragon 810 processor.

7.8 Final Remarks on ARM big.LITTLE Architectures

Based on the above observations, and on the comparison of the Juno SoC and the SnapDragon 810 processor, we can raise the following recommendations for the use of ARM big.LITTLE architectures in the context of critical embedded system:

- Critical real-time tasks must avoid hitting in L2. This implies either hitting in DL1 (in the smallest number of lines possible) or largely exceeding L2 capabilities. Examples of such tasks would be control tasks with limited working sets and streaming tasks. In fact, these types of tasks could easily coexist without jeopardizing their execution times due to contention.

- Exploiting L2 cache space must only be allowed for non-real-time tasks (or non-critical real-time tasks) since they are highly vulnerable to interference due to L2 cache access serialization and L2 evictions caused due to contention.

- Platforms with publicly available and reliable documentation are needed to exercise sufficient control on the platform. Otherwise, uncontrolled or unknown activity can defeat any attempt to master those platforms, as needed for critical embedded systems.

- Lack of documentation of some aspects (e.g. the behavior beyond L2 caches) increases the uncertainty around the findings reached. This relates to the lack of information on whether the scenarios triggered are sufficiently close to the worst-case ones. Since such uncertainty may lead to silent systematic failures, safety

measures need to be put in place to account for the in critical embedded systems. In particular, given that systematic failures may be unrecoverable, the only choice available to preserve safety consists of transitioning the system to a safe state, where availability is jeopardized but safety is preserved. Hence, if further documentation cannot be obtained, ARM big.LITTLE architectures may be regarded as amenable for some fail-safe systems, where a safe state exists, but not for fail-operational systems unless sufficiently independent and diverse redundancy is included.

Chapter 8

Conclusions and Future Work

Critical embedded systems industry is characterized for the need of evidence on applications' correct timing behavior. Such requirement clashes with the increasing need for computing performance to implement new software functionalities as a way to increase the competitive edge of developed embedded products. This is so because covering these performance needs necessarily builds on the use COTS multicores due to their well-known efficiency, availability, and low procurement costs. And unfortunately, multicores pose several new challenges for their timing analysis. In particular, obtaining reliable and tight WCET estimates, which is a must in critical embedded systems, becomes a complex process for COTS multicores. The objective of this Thesis is to ease this transition from singlecore to multicore processors in terms of timing and verification and validation. To that end we have proposed several methodologies to help deriving WCET bounds with increased confidence and time composability, including early software development phase.

8.1 Summary of Contributions

In order to take advantage of COTS multicores performance for their application in critical embedded systems, it is necessary to palliate the extra complexity that they bring to the timing validation and verification process. Next we summarize each contribution we have done to seek that end:

- Our first contribution is a study of the state of the art techniques focusing on multicore contention analysis and bound. We classify the existing approaches and propose a taxonomy. Since the usage of multicores in critical embedded systems

is a problem that has been tackled from different angles, with a large number of proposals, we analyze the assumptions made by each one the techniques.

- Our second contribution consists of a methodology to upper bound the maximum delay in the access to hardware shared resources in COTS multicores. Contention in the access to shared resources is the factor with highest impact on WCET estimate reliability and tightness. But this information is hardly available in the documentation, so here lies the usefulness of our methodology. It relies on measurements using resource stressing kernels so that reliable and tight contention bounds can be estimated despite the limited information available in processor technical specifications. We have tested the robustness of our methodology and verified it with two different arbitration policies controlling the bus and memory controller in a space platform.

- With our third contribution we enable the derivation of tight bounds for contention in shared hardware resources by accounting for the actual contention that tasks will suffer, but without deferring timing analysis to late design phases, when budget overruns would be too costly to fix. To that end, we introduce the concept of resource usage signature and template of a given task. These parametric constructs abstract the potential contention caused and suffered by tasks on a multicore, and allow obtaining WCET estimate.

- Our fourth contribution is a methodology to create surrogate applications during early design phase for WCET estimation. Those applications mimic the multicore timing behavior of the applications modelled. Surrogate applications can be shared during early design phase across software providers for tight WCET estimation without revealing IP. In particular, our surrogate applications mimic the impact on the hardware resources of the original applications, such as cache behavior and access patterns to shared hardware resources, which are the most relevant parameters for WCET estimation in multicore processors.

- Our final contribution consists of the application of our UBD estimation methodology on specific COTS multicore platforms. In particular, we target the ARM big.LITTLE processor architecture. We have used two different boards implementing this architecture: the DragonBoard and the Juno board. We shown that our methodology can be applied with limited success due to the particular timing characteristics of this architecture, which make it not particularly amenable for critical embedded systems. We show that scarce documentation and specific design choices allow arbitrarily high inter-task interference. Thus, we provide specific recommendations for the use of these hardware platforms so that contention can be controlled. Further, we show that commercial platforms whose documentation

is not only scarce, but also imprecise, challenge completely their use for critical embedded systems. This is the case of the SnapDragon 810 processor. While the hardware platform per se could be used for critical embedded systems, it could only be mastered if specific details (e.g. how to disable the prefetcher) are released, which is not the case to date.

8.2 Future Work

The results of our research, globally, show evidence that measurement-based methodologies can be suited on top of COTS platforms to derive information needed for WCET estimation. However, the incorporation of increasingly complex high-performance hardware platforms in critical embedded systems opens the door to extending the research in this thesis in a number of directions, which we detail next:

- Our proposals have been assessed on top of COTS multicores using some form of buses or crossbars to communicate cores with shared hardware resources. The increasing need for performance pushes for the adoption of higher performance platforms such as manycores where cores may be connected with arbitrarily complex (and distributed) networks-on-chip (NoCs) such as meshes and trees. Hence, our *rsk*-based methodology needs to be extended to obtain reliable and tight contention bounds on top of NoCs. We expect those approaches to follow the same principles developed in this thesis, such as exploring different time alignments of requests by injecting nops in between memory accesses, and building bounds on pure measurement-based approaches. However, this methodology needs to be tailored to specific NoC designs whose timing characteristics differ from those of a bus, and where multiple requests may coexist in the NoC without experiencing worst-case contention (if any).

- In this Thesis we have addressed the main hardware components in the access to memory such as shared buses, caches and memory controllers. However, high-performance processors incorporate an increasing number of accelerators such as GPUs, FPGAs, cryptographic units, multimedia units, etc. Those shared resources may have different timing characteristics related to their ability to process multiple requests simultaneously, their latency, and their interaction with memory interfaces among others. Hence, our approach based on *rsk* as well as on signatures and templates needs also to be tailored for these components. The principles developed in this Thesis are expected to hold, but they need to be tailored to specific architectures that may incorporate accelerators shared across cores.

- The research of this Thesis has been assessed on prototypes and benchmarks representative of industrial environments. However, integrating those methods on commercial tools, boards and use cases is key for the exploitation of this technology. While the road towards its adoption is long, due to the strict design and verification processes for critical embedded systems, this process has already started. In particular, the *rsk* methodology and an adaptation of templates and signatures has been ported to the Infineon AURIX TC27x automotive architecture very recently [111]. Such integration is currently undergoing an assessment on an industrial use case of Magneti-Marelli – an Italian automotive Tier1 company – in their own premises. Preliminary results are already promising showing that accurate contention models can be built on measurement-based methodologies delivering tight bounds. In particular, preliminary results for some functions show that contention can be proven to increase the execution time in isolation by less than 10%. Also, our methodology based on *rsk* is currently being integrated in a commercial toolset for WCET estimation and timing verification of critical embedded systems (RapiTime) together with Rapita Systems Ltd. – the owner of the tool. Still, this process needs to continue to port our technology to boards and tools relevant for as many critical domains as possible.

Bibliography

[1] Transparency Market Research. *Embedded System Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast 2015 - 2021*, 2015. URL http://www.transparencymarketresearch.com/embedded-system.html.

[2] Financial Times. *Internet of things drives Intel revenues*, 2015. URL http://on.ft.com/1oH1QXI.

[3] Financial Times. *Arm profits and sales up as shift away from mobile gains pace*, 2016. URL http://on.ft.com/1T6I8Bi.

[4] Intel. *Next-Generation Transportation*, 2017. URL http://www.intel.com/content/www/us/en/automotive/automotive-overview.html.

[5] Wilhelm Reinhard. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

[6] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), 2011. ISSN 0360-0300.

[7] International Organization for Standardization. ISO/DIS 26262 Road Vehicles – Functional Safety, 2009.

[8] Eric Heymann. The digital car: More revenue, more competition, more cooperation. Technical report, Deutsche Bank Research, Frankfurt am Main Germany, July 2017.

[9] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php.

[10] Christopher B. Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference(DASC)*, 2007.

[11] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.

[12] James Windsor, Marie-Helene Deredempt, and Regis De-Ferluc. Integrated modular avionics for spacecraft - user requirements, architecture and role definition. In *EEE/AIAA 30th Digital Avionics Systems Conference(DASC)*, 2011.

[13] Frank Kirschke-Biller. AUTOSAR - a global standard. *4th AUTOSAR Open Conference*, June 2012.

[14] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem: overview of methods and survey of tools. *Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[15] Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, June 2015. doi: 10.1109/SIES.2015.7185039.

[16] FreeScale. *e500mc Core Reference Manual. Rev 3*, 2013.

[17] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *Ninth European Dependable Computing Conference(EDCC)*, 2012.

[18] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems(ECRTS)*, 2014.

[19] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis–definition and challenges. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2016.

[20] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Resource usage templates and signatures for

COTS multicore processors. In *52Nd Annual Design Automation Conference (DAC)*, 2015.

[21] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *Proceedings of the 52Nd Annual Design Automation Conference(DAC)*, 2015.

[22] Gabriel Fernandez, Jaume Jalle, Javier Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Transactions on Computers*, 66(4):586–600, April 2017. ISSN 0018-9340. doi: 10.1109/TC.2016.2616307.

[23] *AMBA Bus Specification. http://www.arm.com/products-/system-ip/amba/amba-open-specifications.php.*

[24] Jan Andersson, Jiri Gaisler, and Roland Weigand. Next generation multipurpose microprocessor. In *DASIA*, 2010.

[25] *NGMP Preliminary Datasheet Version 2.1, May 2013.*

[26] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the NGMP multicore processor in the space domain. In *12th International Conference on Embedded Software (EMSOFT)*, 2012.

[27] Javier Jalle, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco Cazorla. Validating a timing simulator for the NGMP multicore processor. In *21st DASiA*, 2016.

[28] Cobham Gaisler. *LEON4-N2X Data Sheet and User's Manual*, 2013.

[29] Andreas Jung et al. The H2RG infrared detector: introduction and results of data processing on different platforms. In *ESA*, 2012. URL http://www.esa.int/Our_Activities/Space_Engineering/Onboard_Data_Processing/General_Benchmarking_and_Specific_Algorithms.

[30] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. *A Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007.

[31] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM*

Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers, 2012.

[32] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS)*, 2010.

[33] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe(DATE)*, 2010.

[34] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[35] Dakshina Dasari and Vincent Nelis. An analysis of the impact of bus contention on the WCET in multicores. In *IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[36] Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In *Workshop on WCET Analysis*, 2013.

[37] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *4th ACM European Conference on Computer Systems (EuroSys)*, 2009.

[38] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[39] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. IA3: An interference aware allocation algorithm for multicore hard real-time systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[40] Patrick Crowley and Jean-Loup Baer. Worst-case execution time estimation for hardware-assisted multithreaded processors. In *HPCA-9 Workshop on Network Processors*, 2003.

[41] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[42] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[43] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared data caches conflicts reduction for wcet computation in multi-core architectures. In *18th International Conference on Real-Time and Network Systems (RTNS)*, 2010.

[44] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *Workshop on WCET Analysis*, 2010.

[45] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.

[46] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 2013.

[47] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. *IEEE 32Nd Real-Time Systems Symposium (RTSS)*, 2011.

[48] Roman Bourgade, Christine Rochange, and Pascal Sainrat. Predictable two-level bus arbitration for heterogeneous task sets. In *26th International Conference on Architecture of Computing Systems (ARCS)*. 2013.

[49] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91 (1), Jan 2003.

[50] Precision Timed (PRET) Machines. http://chess.eecs.berkeley.edu/pret.

[51] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.(TODAES)*, 2009.

[52] MERASA. *EU-FP7 Project: www.merasa.org*, 2007-2010.

[53] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *36th Annual International Symposium on Computer Architecture (ISCA)*, 2009. ISBN 978-1-60558-526-0.

[54] Javier Jalle, Jaume Abella, Eduardo Quiñones, L Fossati, Marco Zulianello, and Francisco Cazorla. Deconstructing bus access control policies for real-time multicores. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.

[55] Miloš Panić, German Rodriguez, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. On-chip ring network designs for hard-real time systems. In *21st International Conference on Real-Time Networks and Systems (RTNS)*, 2013.

[56] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *IEEE/ACM Sixth International Symposium on Networks-on-Chip (NoCS)*, 2012.

[57] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, USA, 2007. ACM. ISBN 978-1-59593-824-4. doi: http://doi.acm.org/10.1145/1289816.1289877.

[58] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letters (ESL)*, 2009.

[59] D. Dasari et al. Identifying the sources of unpredictability in COTS-based multicore systems. In *SIES*, 2013.

[60] Reinhard Wilhelm, Christian Ferdinand, Christoph Cullmann, Daniel Grund, Jan Reineke, and Benoit Triquet. Designing predictable multicore architectures for avionics and automotive systems. In *Workshop on Reconciling Performance with Predictability (RePP)*, 2009.

[61] Reinhard Wilhelm and Jan Reineke. Embedded systems: Many cores - many problems. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2012.

[62] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic wcet analysis of real-time parallel applications. In *Workshop on WCET Analysis*, 2013.

[63] Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel software. In *Workshop on WCET Analysis*, 2012.

[64] Dumitru Potop-Butucaru and Isabelle Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 21–31, Dagstuhl, Germany,

2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-54-5. doi: 10.4230/OASIcs.WCET.2013.21. URL http://drops.dagstuhl.de/opus/volltexte/2013/4119.

[65] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014.

[66] Javier Jalle, Leonidas Kosmidis, Jaume Abella, Eduardo Qui nones, and Francisco J. Cazorla. Bus designs for time-probabilistic multicore processors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014.

[67] Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Time-analysable non-partitioned shared caches for real-time multicore systems. In *51st Annual Design Automation Conference (DAC)*, 2014.

[68] Patrick J. Graydon and Iain John Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *MCS Workshop*, 2013.

[69] Marco Paolieri, Eduardo Quiñones, and Francisco J. Cazorla. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s), 2013.

[70] Erno Salminen, Tero Kangas, Vesa Lahtinen, Jouni Riihimäki, Kimmo Kuusilinna, and Timo D. Hämäläinen. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *Journal of Systems Architecture*, 2007.

[71] Aniruddha N. Udipi, Naveen Muralimanohar, and Rajeev Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[72] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

[73] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *IEEE 34th Real-Time Systems Symposium (RTSS)*, 2013.

[74] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.

[75] Javier Jalle, Eduardo Qui nones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[76] Jalle Javier, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, 2016.

[77] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS*, 2014.

[78] *DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, 2008.

[79] Kingston. *KVR667D2S5/2G Datasheet*, 2011.

[80] Micron. *DDR2 256 Mbit datasheet*, 2006 .

[81] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - Data Sheet and Users Manual*, 2011.

[82] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *Concurrency Theory (CONCUR)*, 2013.

[83] Hardik Shah, Kai Huang, and Alois Knoll. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[84] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[85] Roman Bourgade, Christine Rochange, Marianne De Michiel, and Pascal Sainrat. MBBA: A multi-bandwidth bus arbiter for hard real-time. In *Embedded and Multimedia Computing (EMC)*, 2010.

[86] Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based WCET analysis for multi-core architectures. 22Nd International Conference on Real-Time Networks and Systems (RTNS), 2014.

[87] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Luca Fossati Tullio Vardanega, Marco Zulianello, and Francisco J. Cazorla. Introduction to partial time composability for COTS multicores. In *Symposium On Applied Computing (SAC)*, 2015.

[88] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Seeking time-composable partitions of tasks for COTS multicore processors. In *IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC)*, 2015.

[89] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4), April 2014.

[90] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.

[91] RapiTime. *www.rapitasystems.com.*

[92] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, 1999.

[93] Sylvain Girbal, Jingyi Bin, Daniel Gracia Perez, and Alain Merigot. Using monitors to predict co-running safety-critical har real-time benchmark behavior. In *International Conference on Information and Communication Technology for Embedded Systems (ICITES)*, 2014.

[94] Mathieu Patte and Vincent Lefftz. System impact of distributed multi core systems. Technical Report ESTEC Contract 4200023100, European Space Agency, 2011.

[95] *http://www.fentiss.com/en/products/xtratum.html*, .

[96] *http://www.gmv.com/en/Products/air/*, .

[97] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully automatic worst-case execution time analysis for matlab/simulink models. In *14th Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

[98] Trevor Harmon, Martin Schoeberl, Raimund Kirner, Raymond Klefstad, Kwang H. Kim, and Michael R. Lowry. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Trans. Industrial Informatics*, 8(2), 2012.

[99] David Trilla, Javier Jalle, Mikel Fernandez, Jaume Abella, and Francisco J. Cazorla. Improving early design stage timing modeling in multicore based real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[100] Jan Gustafsson, Peter Altenbernd, Andreas Ermedahl, and Björn Lisper. Approximate worst-case execution time analysis for early stage embedded systems development. In *7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, 2009.

[101] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2), June 1970.

[102] R. Kirner and P. Puschner. A simple and efficient fully automatic worst-case execution time analysis for model-based application development. In *Workshop on Intelligent Solutions in Embedded Systems*, 2003.

[103] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, 2013.

[104] Renesas. R-Car H3, 2017. https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.

[105] Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core COTS architectures. In *Conference: Embedded Real Time Software and Systems (ERTS^2)*, 2014.

[106] Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[107] ARM. ARM Cortex-A53 MPCore Processor. Revision: r0p4. Technical Ref. Manual, 2013.

[108] Pine64. Pine64 website, 2016. URL https://www.pine64.org.

[109] E. Bost. Hardware Support for Robust Partitioning in Freescale QorIQ Multicore SoCs (P4080 and derivatives), White Paper , 2013.

[110] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[111] Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the AURIX TC27x. In *55th Annual Design Automation Conference (DAC)*, 2018.