Improving Time Predictability of Shared Hardware Resources in Real-time Multicore systems: Emphasis on the Space Domain

Javier Jalle Ibarra

Computer Architecture Department

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

PhD in Computer Architecture

1st of January, 2016

Improving Time Predictability of Shared Hardware Resources in Real-time Multicore systems: Emphasis on the Space Domain

Javier Jalle Ibarra

December 2015

Universitat Politècnica de Catalunya
Computer Architecture Department

Thesis submitted for the degree of
Doctor of Philosophy in Computer Architecture

Advisor:	Francisco J. Cazorla, PhD, Universitat Politècnica de Catalunya
Co-advisors:	Luca Fossati, PhD, European Space Agency
	Eduardo Quiñones, PhD, Barcelona Supercomputing Center
	Jaume Abella, PhD, Barcelona Supercomputing Center

The work reported in this thesis has been conducted at the Computer Architecture and Operating Systems Interface (CAOS) laboratory of the Barcelona Supercomputing Center (BSC), and has been financially supported by the Network Partnering Initiative (NPI) project 4000102880, between BSC, the European Space Agency (ESA) and Rapita systems Ltd.

Dedication

This thesis is dedicated to the memory of my grandmother Amparo, which would have enjoyed this moment more than anyone else. I am also grateful to my loving parents that through their continuous support and sacrifice allowed me to get such success.

To my partner, Coral, for making me a better person and light up my life. Finally, I must acknowledge as well my sister, family and friends, whose support and company helped me focusing on what is important in life.

Acknowledgements

This thesis is funded by the European Space Agency (ESA) under a Network Partnering Initiative (NPI) (project 4000102880), between ESA, Barcelona Supercomputing Center (BSC) and Rapita systems Ltd. The topic of the NPI project is "'Architectural solutions for the timing predictability of next generation multi-processors"', which aims for easing the adoption of multicores by ESA and ultimately by European real-time industry.

This thesis has also received funding from the European Communitys Seventh Framework Programme [FP7/2007-2013] under the PROARTIS (grant agreement 249100), the PROXIMA (grant agreement 611085) and parMERASA (grant agreement no. 287519) projects. Also the Spanish Ministry of Science and Innovation (grant TIN2012-34557), the HiPEAC Network of Excellence and European Space Agency under contracts 4000109680 and 4000110157.

I would like to express my deepest gratitude to my advisors Eduardo Quiñones, Jaume Abella, Luca Fossati and Francisco J. Cazorla for their endless support and help. I would also like to acknowledge Marco Zulianello and the Microelectronics section from the European Space Agency, Guillem Bernat from Rapita Systems, Jan Andersson from Cobham Gaisler and Tullio Vardanega from University of Padua for their contributions to this thesis. Also thanks to all my colleagues/friends from the Barcelona Supercomputing Center for their help, support and friendship during this research.

Abstract

Critical Real-Time Embedded Systems (CRTES) follow a verification and validation process on the timing and functional correctness. This process includes the timing analysis that provides Worst-Case Execution Time (WCET) estimates to provide evidence that the execution time of the system, or parts of it, remain within the deadlines. A key design principle for CRTES is the incremental qualification, whereby each software component can be subject to verification and validation independently of any other component, with obvious benefits for cost. At timing level, this requires time composability, such that the timing behavior of a function is not affected by other functions.

CRTES are experiencing an unprecedented growth with rising performance demands that have motivated the use of multicore architectures. Multicores can provide the performance required and bring the potential of integrating several software functions onto the same hardware. However, multicore contention in the access to shared hardware resources creates a dependence of the execution time of a task with the rest of the tasks running simultaneously. This dependence threatens time predictability and jeopardizes time composability.

In this thesis we analyze and propose hardware solutions to be applied on current multicore designs for CRTES to improve time predictability and time composability, focusing on the on-chip bus and the memory controller. At hardware level, we propose new bus and memory controller designs that control and mitigate contention between different cores and allow to have time composability by design, also in the context of mixed-criticality systems. At analysis level, we propose contention prediction models that factor the impact of contenders and don't need modifications to the hardware. We also propose a set of Performance Monitoring Counters (PMC) that provide evidence about the contention. We give an special emphasis on the Space domain focusing on the Cobham Gaisler NGMP multicore processor, which is currently assessed by the European Space Agency for its future missions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Critical Real Time Embedded Systems

Real-time Embedded Systems (RTES) are computing systems with a specifically designed function whose correctness does not only rely on their *functional correctness* but also on their *timing correctness*. RTES have to meet the timing restrictions on the execution time of the system, which are typically represented with a deadline. RTES are ubiquitous nowadays and can be found embedded in music players, TVs, cars, airplanes, satellites, spacecrafts (among others). These systems interact with the physical world, where time is a central property of systems, responsible of the evolution of its state and dynamics. Inside RTES, we can differentiate an specific branch of systems called Critical Real-Time Embedded Systems (CRTES) [116]. In this context, *criticality* relates to safety, security, mission or business aspects of the system. For instance, a failure or disruption in a mission critical system will cause a failure to accomplish the mission or goal of the system. Likewise, the consequences of a failure in a safety critical system involve human injury, environmental harm, or loss or severe damage to equipment/property. As an example, the guidance and navigation control from a spacecraft, which is in charge of the correct position and orbit of the spacecraft, involves critical failures if a deadline is missed, since a wrong position or orbit means the complete loss of the spacecraft, due to loss of power (not pointing to the sun for solar powered spacecrafts) or communication (antennas are directional and have to be properly oriented). Other types of CRTES can be found in avionics, space and automotive systems, among others, such as the brake system in a car or the flight control in an airplane.

CRTES have stringent constraints in terms of size, weight, cost and power consumption, as many embedded systems, but also need to ensure timing correctness in addition to functional correctness. To do so, CRTES follow a verification and validation [3] process that provides a high level of assurance on the functional and timing correctness. The verification and validation process is specific to each domain, usually specified in safety-related standards, such as the ISO26262 [91] in automotive, DO-178B [152] in avionics or ECSS-E/Q-40 [57, 56] in space. Verification is the process used to check that the requirements of a system are satisfied, which can be classified into functional verification and timing verification. The former checks that the system is functionally correct while the latter verifies that timing constraints are met. Validation checks that the system performs what it purports.

1.2 Timing aspects of CRTES

To guarantee the *timing correctness* of CRTES, evidence has to be provided that tasks[1] do not overrun their deadlines. However, this is not an easy task since most programming models used on CRTES, such as C or Ada, do not have the notion of time [119]. As a result, assigning an execution time to a certain piece of code is not trivial and depends on many factors, such as programming language, compiler and ultimately depends on the underlying hardware architecture. For instance, a piece of code with different execution paths, such as those generated by control-flow (branch) instructions, may have different execution times depending on the input data of the software, which modifies the branch condition. On the other hand, there is a certain variation of the execution time of processor instructions depending on the input data and the execution history due to the fact that the hardware is not a stateless resource. For instance, the latency of a division in some hardware implementations depends on the input data (e.g., if the dividend is zero the result is delivered quickly) and also most cache memory designs completely depend on the execution history, which is exactly what they benefit from to obtain performance gains by exploiting temporal and spatial locality. As a result, the execution time of a task is not a fixed value but rather a complex distribution as seen on Figure 1.1. For that purpose, most systems use the Worst-Case Execution Time (WCET)[2] as an abstraction of the worst timing behavior of a task that reduces the complexity of providing evidence about the execution time. The WCET, or more precisely a bound to it (as discussed below) is computed for each software unit. This estimate is passed as input to the scheduler, which prioritizes the order of execution of the different software modules with the goal of asserting that from a timing perspective, the system behaves as expected, with no deadline misses. A summary of scheduling approaches, and the corresponding schedulability analy-

[1]In this thesis we use the terms task and application interchangeably. They both represent stand-alone software components able to run independently on a processor.

[2]Same for the Best Case Execution Time (BCET), although in this thesis we focus on the WCET.

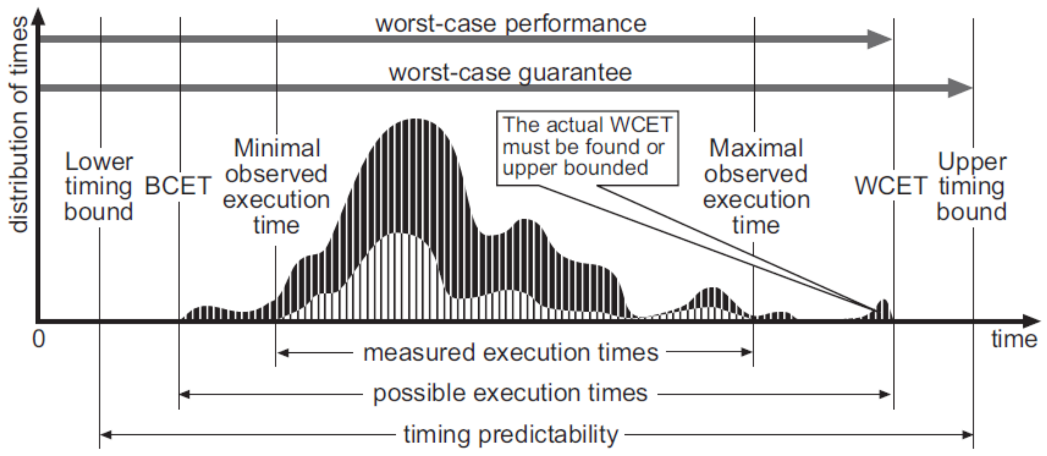


Figure 1.1: Basic notions concerning timing analysis of systems. Source: [180].

sis techniques, are discussed in [28, 48]. Thus, the WCET must be computed to guarantee that task's functionality finishes within its corresponding deadline.

However, the exact WCET is very difficult or even unfeasible to obtain at all [180], because the test space is too big to be covered with all possible input sets and/or the system might not be completely predictable. The ability to derive WCET estimates requires *time predictability*, which stands for the property of being able to bound the execution time of a task or a component, so that none of the possible execution times, exceeds the bound. When the exact bound cannot be computed, some abstractions are made by the tool or method to be able to have a time predictable system or component, usually overestimating it by introducing pessimism. The amount of pessimism depends on the methods used and the hardware characteristics. For instance, if we consider a cache memory, a possible abstraction that makes the cache time predictable is to consider all accesses as cache misses. In that case, the obtained WCET bound is never exceeded because of the cache effect and we avoid the struggle of knowing the execution history. However, the pessimism introduced can be too high to be useful since we are completely removing the performance gains of cache memories in the worst-case. A better solution might be to lock the cache contents and consider each request as a hit or a miss depending on the address and not on the execution history [168].

Deriving WCET estimates is a known challenge, for which different methods have been proposed. These methods can be classified into *Deterministic Timing Analysis* (DTA) and *Probabilistic Timing Analysis* (PTA) [2]. DTA techniques assume that under the same input data and initial state a piece of software running on a processor has a defined execution time. Therefore, DTA derives a single bound for the execution time of each piece of software running on a processor.

These bounds are used to estimate the WCET for different task fragments, which can be combined to obtain an estimate of the task's WCET [180].

PTA [52, 32, 46] has appeared as an alternative to classic timing analysis. PTA provides probabilistic WCET (pWCET) estimates, such that a WCET estimate can be exceeded with a given probability, thus leading to a timing failure. Such probability is called exceedance probability, which can be arbitrarily low (e.g 10^{-16} per run). PTA extends the notion of system reliability to timing correctness. In that sense, PTA expresses timing correctness with probabilities of failure, as occurs with current system reliability of CRTES that is expressed in terms of probabilities for hardware failures, software functional faults and for the system as a whole. An example of the outcome of PTA is shown in Figure 1.2 that shows the pWCET distribution for a task. In this case for different probabilities (e.g. 10^{-13} or 10^{-16}) we have different values of pWCET. It is clear that the exceedance probability can be arbitrarily low which results in bigger pWCET estimates.

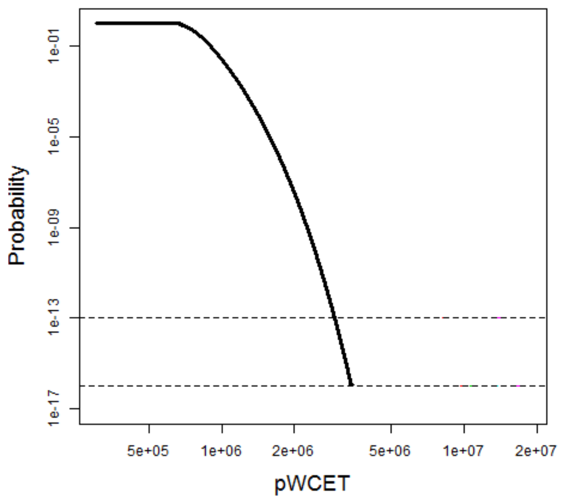


Figure 1.2: Basic notions concerning probabilistic timing analysis of systems.

PTA techniques require that execution times have a distinct probability of occurrence and are independent and identically distributed (i.i.d), i.e. the observed execution times are independent among them and must follow the same probability distribution. The main advantage of PTA is that it is less dependent on execution history, allowing to significantly reduce the amount of information required to obtain tight WCET estimates in comparison to classic techniques [175, 2]. Solutions for single-core architectures [110][46] show how processor cores can be easily adapted to apply PTA techniques.

Both, DTA and PTA, can be applied in a static or measurement-based manner. Static techniques rely on the construction of a cycle-accurate model of the system under test (not limited to the CPU), and the construction of a mathematical representation of the machine code timing behaviour on that model. The mathematical representation is then processed with linear programming techniques to determine a safe upper-bound on the execution time. On the other hand, measurement-based techniques are based on executing the program on the real platform with different input sets and collecting measurements, which are later operated in order to derive WCET estimates. In an intent to benefit from both, hybrid approaches try to improve the confidence of measurements by combining them with static information that gives control flow and coverage information for unobserved execution paths. An example of a hybrid tool is *Rapita Verification Suite* [147]. A summary of the different timing analysis methods is shown in Chapter 2 and can also be found in [3].

1.3 Challenges in future CRTES

Until recently, CRTES have been designed following a *Federated Architectures* [49] approach, in which different functions in a system were assigned different hardware units where the software component of each function runs. Physical separation facilitates timing and functional isolation, which allows providers to implement system functions independently from other suppliers. Despite these properties, as the number and complexity of functions increases so does the number of hardware units, reducing the efficiency of the system in terms of size, weight and power consumption.

CRTES are experiencing an unprecedented growth [10, 35] with rising performance demands to provide more value-added functionality. The need of more performance is shared across most real-time domains and ultimately results in an increasing demand for computation power.

- In the automotive domain, future systems will require much more processing power than current ones [30, 161] to deploy advanced driver assistance systems. A premium vehicle nowadays contains already more than 100 Million lines of code [136].
- In the avionics domain, Software is also growing to bring more functional value [51].
- In the space domain, the complexity and the amount of data to be handled by on-board software is rising [178] and the fact that space missions are becoming more autonomous accentuates this trend.

This has motivated several industries from these domains to shift towards an *Integrated Architecture* paradigm: a modular approach in which multiple functions are assigned to a single hardware unit. Examples of integrated architectures are the Integrated Modular Avionics (IMA) [176] in the avionics domain, the Automotive Open System Architecture (AUTOSAR) [18] [49] in automotive or IMA for space [182] for the space domain. A key design principle for integrated architectures is the *incremental qualification* [53], whereby each software component is subject to verification and validation – including functional and timing analysis – in isolation, independently of any other component, with obvious benefits for cost, time and effort. At functional level, this translates into providing mechanisms for functional isolation, such that no misbehaving function may corrupt the data used by other functions. At timing level, this requires *time composability*, such that the timing behavior of a function is not affected by the execution of other functions.

Time composability is a property that determines that the timing behavior of an individual component does not change in the face of composition when the system is integrated, and so, the timing analysis performed in isolation remains valid at system integration. During system development, time composability enables incrementally integrating applications without the need of regression tests to validate the timing properties of already-integrated applications, which heavily reduces integration costs. During operation, time composability enables updating functions and their associated software without the need for re-analyzing and re-qualifying the system. This is specially beneficial in domains like space where systems may operate during dozens of years and whose functionality is usually updated after deployment. Time composability therefore reduces the cost of system integration and qualification, which is one of the most critical challenges faced by system developers.

The high performance requirements for CRTES can be met by designing much more complex processors, e.g. with out-of-order execution, complex branch prediction and higher clock frequencies. However, applying those techniques to CRTES design is difficult, because such techniques usually have high energy requirements that affect the low-power constraints of embedded systems. More important for CRTES is that they complicate the timing analysis, usually requiring a large amount of information about the system in order to be able to provide reliable or trustworthy WCET estimates. Also they could introduce timing anomalies [126] due to their non-deterministic run-time behavior. Timing anomalies occur when a local worst case does not contribute to a global worst case execution time, which is counter intuitive and can significantly complicate the computation of WCET estimates. An example of timing anomalies is shown in [79].

1.3.1 Multicore processors for future CRTES

In this context, multicore processors can satisfy the growing performance requirements of CRTES while maintaining a simple core design, free from timing anomalies and with limited impact on timing analysis. Multicores also have a good performance-per-watt ratio, allow parallel execution [167] and enable co-hosting mixed-criticality workloads, i.e. composed of tasks subject to different criticality levels, which is of paramount importance in the embedded system market because hardware utilization is maximized resulting in an overall reduction in cost, size, weight and power requirements.

Mixed-criticality systems [169] can consolidate onto the same hardware applications with different criticality levels in terms of safety (and security). Safety standards define multiple safety integrity levels for each domain, e.g. DAL in avionics [155] and ASIL in automotive [141], in the space domain it is well accepted that on-board systems comprise two criticality levels [142]. One level covers *control* applications, which require real-time execution and are designed to meet requirements in the worst case. The second covers *payload* applications that are high-performance driven and only may have some (soft) real time requirements.

The use of multicores in CRTES is not straightforward. On the one hand, any application must be prevented from corrupting the state of other applications, paying special attention to preventing low-criticality applications from affecting high-criticality ones, providing functional isolation. This can be accomplished through software isolation [80] and has been done within the space domain [142].

On the other hand, timing behavior of multicore processors brings new challenges with respect to single-core processors, mainly due to the contention on the access to shared hardware resources, called inter-task interferences. *Inter-task interferences* appear when several tasks in different cores try to access a shared hardware resource at the same time, thus creating contention, potentially affecting the execution time of running tasks and threatening *time predictability*. For instance, if two tasks in two different cores connected by a shared bus try to access it at the same time, one of them will have to wait while the other one uses the bus. As a result, the execution time of a task is not independent of the rest of the tasks running simultaneously on the processor. Providing a meaningful timing analysis becomes difficult, since all possible interactions of any task in the workload have to be considered when analyzing a task, which jeopardizes *time composability*. In fact, this contention has been identified as one of the most prominent issues when using multicores on the real-time domain [31]. Examples of shared hardware resources in multicore processors are the interconnection network, shared caches and shared memory devices.

Despite all theses issues, the real-time industry is facing an increasing pressure to adopt multicores as its main computing platform because processor vendors are driven by the high performance demands rather than by the timing requirements of the comparatively small real-time market. Some available commercial-off-the-self (COTS) multicore processors considered by the real-time industry so far are:

the Infineon AURIX platform [84], consisting of three TriCore processor cores used in the automotive industry. In the avionics domain, the Freescale P4080 [70] has eight power architecture cores. The Cobham Gaisler NGMP [41] processor is used in the space domain, which consist of four LEON4 cores. However, none of these processors have been used for the most critical systems due to its timing analysis complexity and the lack of a full-fledged WCET estimation solution.

In this respect, multicore timing analysis is still in its infancy, especially for COTS multicores. The dependence of the execution time of a task running on a multicore with the rest of the tasks running simultaneously impacts directly on system design and verification by threatening *time predictability* and jeopardizing *time composability*. Both properties are requirements for the incremental development and verification model that contains qualification cost and development risks for industry. On the contrary, a time predictable and composable multicore system allows to analyze in isolation each task independently of the rest of the tasks, which can mitigate the problem of timing analysis in multicore systems. As a result, the cost of applying multicore solutions for CRTES is reduced. To that end, the effect of the inter-task interferences (contention) needs to be either determined or bounded independently of the tasks, so that the WCET analysis of a task does not depend on the rest of the tasks that may run simultaneously with the actual task, thus making the WCET estimate independent of the rest of the tasks.

1.3.2 The case of the space domain

Computing systems used by the European Space Agency (ESA) can be broadly classified as platform and payload. The former comprise the main functionalities required by the spacecraft system, such as power management, communication, guidance and navigation. The latter comprise the systems used for the specific mission of the spacecraft, such as infrared detectors [97], cryogenic systems [83], telescopes, etc. There are four different criticalities [58] for software in space, shown in Table 1.1: A,B,C and D. Criticality A is only used for human spaceflights and launchers, which can cause human loses if they fail. In a spacecraft, usually two criticalities are present, which are usually called control and payload [142]. Control (criticalities B and C) have real-time requirements and payload are performance-driven having soft real-time or no timing requirements (criticality D). Although less frequent, some payloads are also designed with high criticality in mind. For instance, the control of the cryogenic system of Herschel [83] had to keep the sensors at 1.7 Kelvin degrees for about 3 years. The failure of the payload system would have compromised the mission, hence making it mission critical. Developing software, specially for highest criticality, is pricey[3]. For that reason, space industry values solutions that do not require changes in the software stack, i.e., operating system or applications. The timing analysis is performed using static (SDTA) along with measurement-based (MBDTA) timing analysis in certain predefined scenarios.

Table 1.1: Space software criticalities classification [58]

Criticality	system failure
A	Catastrophic consequences
B	Critical consequences
C	Major consequences
D	Minor or Negligible consequences

ESA is leading the development of LEON processors [38, 44] based on the SPARC ISA. This enables taking into account space domain specific requirements, in particular the radiation hardness/tolerance needed to survive the space environment. Radiation affects silicon and requires countermeasures at all system levels: architecture, wafer process, cell layout and netlist design [120]. This requires availability and knowledge of the whole design process of a processor.

Following the same trend in other domains such as avionics, automotive or railway, in the space domain, the complexity and the amount of data to be handled by on-board software is rising [178]. The fact that space missions are becoming more autonomous accentuates this trend and ultimately results in an increasing demand for computation power. For instance, spacecrafts for Active Debris Removal [86], which have to remove space junk autonomously, require a complex autonomous Guidance and Navigation Control (GNC) system with image processing inside the control loop to be able to determine the rendezvous trajectory. The GNC system is critical, thus having real-time constraints, while image processing has high-performance requirements and soft real-time constraints. According to the requirements of these systems, a simple, cost-effective yet high-performance computing solution is needed, with a test bed environment that allows verification and validation of the complete solution. Current solutions for these requirements require replicating several single-core systems, which leads to more space, weight and power consumption. For instance, there are 249 high complexity integrated circuits in Sentinel-2 from which 21 are microprocessors [120].

The current multicore processor design envisaged by ESA and developed by Cobham Gaisler is called the Next Generation Microprocessor (NGMP). Processors of the NGMP family, however, still have never been used, apart from research, on real flights because of the loss of predictability and also missions because of the

[3]Around Million € depending on criticality and size.

lack of availability of flight models. Nevertheless, ESA is on a unique position, with respect to real-time systems, because it is able to influence the design of future multicore processors derived from the NGMP family.

1.4 Thesis contributions

This thesis proposes hardware designs to improve the timing predictability of shared resources in multicore processors in order to ease their adoption by the CRTES industry. This thesis covers solutions for both, deterministic and probabilistic timing analysis techniques. Proposed designs deliver both high average performance and low WCET estimates, thus increasing the number of functions that a single system can provide.

Time composability is maintained by all of our proposals, in the sense that the WCET estimation computed for a task in isolation is not affected by the rest of the tasks running on the system simultaneously.

The approach followed in this thesis is to bound or reduce the interference (contention) that appears when several cores try to access the same shared resource. This interference must have a guaranteed maximum impact in order to allow deriving WCET estimates. We note that the real-time industry has just started using small multicores with few cores, e.g. the Cobham Gaisler NGMP [42] architecture is four-core, the Infineon AURIX architecture [84] comprises 3 cores and the Freescale P4080 [70] 8 cores. For these core counts, the main shared resources are i) the interconnect that connects the core to ii) the shared cache hierarchy and finally the iii) memory controller that acts as an interface between the processor and the off-chip memory. On-chip buses suffice to provide the bandwidth needed in those architectures and provide more efficiency and simplicity than other interconnects for a low number of cores. Also several studies show that hierarchical bus configurations scale quite easily to large systems and provide a good area-performance trade-off, while retaining many of the advantageous features of simpler bus arrangements [156] and that bus-based networks can significantly lower energy consumption and simplify network protocol design and verification, with no loss in performance [166]. Shared caches pose significant challenges for real-time systems, since running tasks can cause the eviction of cache lines owned by different tasks. To remove this effect partition mechanisms exist, either hardware [138, 105] or software [122, 131, 103, 172], that remove the cache interference. In this thesis, we focus on processors that have partitioning mechanisms to remove the interference on the shared caches. Current CRTES manage large working sets which requires the use of off-chip memories such as DDR2 and DDR3 [92] memories. The memory bandwidth, which is arbitrated by the memory controller, is one of the shared resources with the highest impact on systems' performance and predictability [139, 183].

This thesis focuses on two of the most critical shared resources in multicore processors: the interconnection bus [31] and the memory controller [139, 183]. We use the NGMP processor as reference architecture, acknowledged as one of the multicores that can be used by the ESA. Nevertheless, most solutions can be applied to other processors and domains, like the Infineon AURIX [84] in automotive or the Freescale P4080 [70] in avionics. In this thesis, the impact of scheduling aspects is not considered and does not affect our solutions, in the sense that our solutions are transparent to them, enabling the time-composable WCET estimates, which are the input for scheduling algorithms.

More concretely:

- On-chip buses:
  - We analyze a specific implementation of a bus architecture, the AMBA bus, which is one of the most widely used on-chip bus interfaces in embedded processors and provide a classification of its features in terms of time predictability. We define a new bus based on AMBA which is time composable by construction, which greatly simplifies timing analysis.
  - We analyze and derive analytical models for the most relevant on-chip bus arbitration policies for real-time systems to be able to compare them in terms of time predictability and time composability.
- Memory controller:
  - In the context of mixed-criticality systems, predictable latencies and high bandwidth must be satisfied for different tasks at the same time. We propose a memory controller that can satisfy both requirements.
  - We propose a novel contention free memory controller solution able to simplify timing analysis and reduce pessimism on WCET estimates introduced by memory accesses.
- Probabilistic Timing Analysis in multicores:
  - We enable the use of PTA in multicore systems: In particular, we define probabilistically-analysable bus and memory controller designs for multicore processors, show their suitability for PTA and evaluate their hardware cost.
- Performance Monitoring Counters

  - We propose a new set of Performance Monitoring Counters (PMC) aiming to measure the actual contention in shared resources. These PMCs provide evidence and allow to verify and test the outcome of the timing analysis.
  - For systems in which hardware cannot be modified, we propose an interference prediction model based on PMC.

As a proof of concept, all the architectural solutions studied and proposed are tested on a space case-study consisting on a cycle-accurate simulation framework validated against a real implementation of the NGMP [41] processor and representative software of the ESA. The results will be used by the ESA to influence the future design of the NGMP processor.

1.5 Thesis structure

This Thesis is structured as follows:

- Chapter 2 covers the background and state-of-the-art needed for this thesis.
- Chapter 3 explains the methodology followed in this thesis.
- Chapters 4 and 5 cover the on-chip buses. More concretely:
  - Chapter 4 proposes an on-chip bus architecture that is time composable by design.
  - Chapter 5 analyzes bus arbitration policies that are appealing for real-time systems.
- Chapters 6 and 7 cover the memory controller. More concretely:
  - Chapter 6 proposes a dual-criticality memory controller.
  - Chapter 7 proposes a contention-free memory organization.
- Chapter 8 enables the use of PTA in multicores by evaluating and proposing new arbitration policies for the bus and memory controller.
- Chapters 9 and 10 improve the timing analysis of multicore processor. More concretely:
  - Chapter 9 proposes a new set of Performance Monitoring Counters that provide evidence about the contention.
  - Chapter 10 proposes a contention prediction model that factors the impact of contention in the timing analysis.

1.6 List of publications

1. J. Jalle, M. Fernndez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello and F. J. Cazorla. “Contention-Aware Performance Monitoring Counter Support for real-time MPSoCs” (Chapter 9), 2016 IEEE Symposium on Industrial Embedded Systems (SIES). May 2016.

2. J. Jalle, E. Quiones, J. Abella, L. Fossati, M. Zulianello and F. J. Cazorla. “Bus Slicing for Contention-Free Multicore Real-Time Memory Systems” (Chapter 7), 2016 IEEE Symposium on Industrial Embedded Systems (SIES). May 2016.

3. J. Jalle, J. Abella, L. Fossati, M. Zulianello and F. J. Cazorla. “Validating a Timing Simulators for the NGMP Multicore Processor” (Chapter 3), Data Systems In Aerospace (DASIA), May 2016.

4. J. Jalle, M. Fernndez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello and F. J. Cazorla. “Bounding Resource-Contention Interference in the Next-Generation Multipurpose Processor (NGMP)” (Chapter 10), Embedded Real Time Software and Systems (ERTS2), January 2016.

5. J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello and F. J. Cazorla. “A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation for a Space Case Study” (Chapter 6), IEEE Real-Time Systems Symposium (RTSS), December, 2014

6. J. Jalle, L. Kosmidis, J. Abella, E. Quiones and F. J. Cazorla, “Bus designs for time-probabilistic multicore processors” (Chapter 8), Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2014.

7. J. Jalle, J. Abella, E. Quiones, L. Fossati, M. Zulianello and F. J. Cazorla. “AHRB: A High-Performance Time-Composable AMBA AHB Bus” (Chapter 4), Real-time and embedded Technology and Applications Symposium (RTAS), April 2014.

8. J. Jalle, J. Abella, E. Quiones, L. Fossati, M. Zulianello and F. J. Cazorla. “Deconstructing Bus Access Control Policies for Real-Time Multicores” (Chapter 5), 2013 IEEE Symposium on Industrial Embedded Systems (SIES). June 2013.

9. J. Jalle, E. Quiones, J. Abella, L. Fossati, M. Zulianello and F. J. Cazorla. “Contention-Free Multicore Real-Time Memory System by Data Bus Splitting and Command Bus Sharing” (Chapter 7), Work-in-progress Poster in Design Automation Conference (DAC), June 2015.

10. J. Jalle, E. Quiones, L. Fossati, M. Zulianello, F. J. Cazorla. "Threats to Time Predictability on AMBA Bus" (Chapter 4), ACACES 2012, Poster Abstracts. 8th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES). Fiuggi (Italy), July 2012.

Collaborations

11. G. Fernandez, J. Jalle, E. Quinones, J. Abella, T. Vardanega and F. J. Cazorla. "Resource Usage Templates and Signatures for COTS Multicore Processors" Design Automation Conference (DAC), June 2015.

12. G. Fernandez, J. Jalle, E. Quinones, J. Abella, T. Vardanega and F. J. Cazorla. "Increasing Confidence on Measurement-Based Contention Bounds for Real-Time Round-Robin Buses", Design Automation Conference (DAC), June 2015.

13. D. Trilla, J. Jalle, M. Fernandez, J. Abella and F. J. Cazorla. "Improving Early Design Stage Timing Modeling in Multicore Based Real-Time Systems", to Real-time and embedded Technology and Applications Symposium (RTAS), April 2016.

Chapter 2

Background and State-of-the-Art

This chapter provides the background and state-of-the-art on timing analysis techniques we build upon, and on the existing hardware solutions to enable timing analysis of multicore designs.

2.1 Timing Analysis

As mentioned before in this document, two main approaches are currently being followed [3] to derive WCET estimates: Deterministic Timing Analysis (DTA) [180] or Probabilistic Timing Analysis (PTA) [4] techniques. DTA derives a single WCET for each piece of software [180] and PTA generates a distribution function, or pWCET function, that guarantees that the execution time of a program only exceeds the corresponding pWCET bound with a probability lower than a given target probability (e.g., 10^{-15} per run) [32].

Both, DTA and PTA, can be applied in a static or measurement-based manner. In an intent to benefit from both, hybrid approaches try to combine static with measurement-based techniques. A summary of the different timing analysis methods is shown in Table 2.1 [3].

Technique	Deterministic	Probabilistic
Static	SDTA	SPTA
Measurement-based	MBDTA	MBPTA
Hybrid	HYDTA	HYPTA

Table 2.1: Taxonomy of timing analysis techniques. Taken from [3]

2.1.1 Deterministic Timing Analysis

DTA is a well-known and studied timing analysis technique, with consolidated commercial and academic research tools. DTA techniques can be classified in *Static Deterministic Timing Analysis* (SDTA), *Measurement-based Timing Analysis* (MBDTA) and *Hybrid Deterministic Timing Analysis* (HYDTA) [3]. SDTA relies on the construction of a cycle-accurate model of the system under test (not limited to the CPU), what we call hardware-model, and the construction of a mathematical representation of the machine code timing behaviour on that model, called flow-facts, that take care of loop bounds, infeasible paths and annotations. These mathematical representations are then processed with linear programming techniques to determine a safe upper-bound on the execution time. The main advantage of SDTA is that the application of provably mathematical techniques gives a strong emphasis to the soundness and safeness of SDTA [180], which allow in principle to meet all the requirements set by the highest criticality standards. However, the validity of the bounds depends on the correctness of the abstract models.

Hardware models are difficult to develop and test, specially because of the lack of information about the processor implementation provided by the manufacturer, which is usually scarce and not accurate [179]. Even if the chip vendor explicitly provides this information in the manuals, documentation can be inaccurate or outdated with respect to the deployed chip implementation [3]. For instance, the FreeScale e500mc core documentation has already reached the third revision with details about non-negligible changes across revisions [72]. This forces SDTA to make assumptions, affecting the accuracy of the estimates and usually including more pessimism on them. All these difficulties have made that real-time industry and SDTA tool providers use measurement-based approaches to derive contention bounds [135]. On the other hand, flow-facts grow linearly/exponentially with code size and adding annotations is an error-prone process [3]. Only simple software running on top of simple and consolidated hardware designs have accurate abstract hardware models that can be subject to SDTA [3]. Examples of deterministic SDTA tools are *aiT* [60] and *OTAWA* [19].

MBDTA is based on executing the program on the real platform with different input sets and collecting measurements, which are later operated in order to derive WCET estimates. For instance, the outcome of recording the longest observed execution time and applying an engineering margin can be used as a WCET estimate. The advantage of MBDTA is that it does not require as much information as SDTA does which is why industry often relies on MBDTA [3]. In fact software/hardware testing always implies executing on the real platform even for functional testing, so measurements are always collected on the actual system to gain confidence. However, the main drawback of MBDTA is that its trustworthiness depends on factors such as the test conditions and input set, which cannot guarantee that they trigger the worst-case scenario, since it is usually not known and cannot be found due to the large input set space. An example of research on MBDTA techniques is shown in [177].

Hybrid approaches try to improve the confidence of measurements by combining it with static information that gives control flow and coverage information for unobserved execution paths. An example of a hybrid tool is *Rapita Verification Suite* [147], which is the one used in this thesis. RapiTime [146] uses path analysis techniques to build up a precise model of the overall code structure and determine which combinations of subpaths form complete and feasible paths through the code. RapiTime combines the measurement and control flow analysis information to compute measurement based WCET estimates in a way that captures the execution time variation on individual paths.

2.1.2 Probabilistic Timing Analysis

PTA generates a distribution function, or pWCET function, that upper-bounds the execution time of the program under analysis, guaranteeing that the execution time of a program only exceeds the corresponding execution time bound with a probability lower than a given target probability (e.g., 10^{-15} per run). PTA can be applied either on a Static manner (SPTA) [32] or Measurement-Based (MBPTA) [46]. PTA requires that the events under analysis, program execution times for MBPTA or instruction latencies for SPTA, can be modeled with independent and identically distributed (*i.i.d.*) random variables[1]. Each PTA technique has its own methods to combine results from different execution paths. We refer the reader to those methods for further details [32] [46].

SPTA uses very simple processor models, not suitable for more realistic processor designs [129][9]. In this thesis we focus on MBPTA as it is closer to industrial practice, and obtained pWCET estimates have shown to be competitive with the estimates obtained with other timing analysis techniques [175][174]. MBPTA derives probabilities by collecting execution time observations of end-to-end runs of an application running on the target hardware. The observed execution times fulfil the i.i.d. properties if observations are independent across different runs and a probability can be attached to each potential execution time. Hence, taking measurements from a program is equivalent to rolling a dice, with each face having a probability of appearance. Making enough rolls is enough to apply MBPTA, which derives upper-bounds of the execution time distribution by means of the *Extreme*

[1]Two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution function.

Value Theory (EVT) [115, 46]. The path coverage of MBPTA can be improved with techniques such as the ones proposed in [111, 190].

Conventional hardware/software platforms fail to provide the features required by PTA so that pWCET estimates can be computed. PTA has been shown implementable for single-core processors, either by using random-placement and random-replacement [110] caches or by using software randomisation that changes memory allocation at runtime [112] or by using different compiled binaries [113]. Also PTA works with multilevel caches [109], buffer resources [114], shared caches [163] and faulty caches [162]. Solutions for single-core architectures show how processor cores with a similar processor architecture to the LEON3 and LEON4 [81] can be easily adapted to achieve PTA requirements. In particular, limited modifications are required in the cache placement and replacement [110] policies to make them PTA-compliant. PTA has emerged as a recent alternative to DTA, still some further steps are required before PTA can be used on real products [3].

To the best of our knowledge, no multicore PTA compliant architecture has been presented so far. In this thesis we propose and evaluate multicore architectures that fulfil PTA requirements.

2.2 Multicores

Providing a meaningful timing analysis in multicores is difficult [106, 128, 3] due to contention on access to hardware shared resources [65]. A taxonomy of different works on the subject can be found in [62]. A possibility to deal with contention is to consider all possible interactions of any task in the workload when analyzing a task, which is called *joint analysis*. For instance, in [184, 121, 188, 78] authors analyze all the possible effects a task may have onto another one in a shared cache. In [33] they integrate the shared cache and shared bus timing analysis with other architectural features like the pipeline or branch prediction. In reality, this type of analysis is complex and requires knowledge of the tasks running on the system, which breaks the time composability property and increases the complexity of the analysis [134].

Other solutions propose changes to the hardware that simplify the timing analysis, for instance [138] uses Round-Robin and [151] TDMA for managing shared resources. Based on the method in [138], authors in [159] use hardware support to artificially delay each request when measuring the WCET. The COMPSoC [77] multiprocessor achieves predictable and composable latencies by using TDMA arbitration and avoiding cache memories. PRET [123] architecture uses scratchpad cache memories and TDMA based memory scheduling [148] to achieve a predictable multicore processor. Also the time-triggered architecture [107] provides a predictable a composable hardware by making use of a time-triggered schedule, similar to TDMA, and global synchronization.

Existing solutions using Commercial-of-the-shelf (COTS) multicore processors assume the worst-case impact of contention [138], for instance in [135] they propose a method to compute an interference sensitive WCET and monitor its safeness at runtime on top of a COTS processor. Also in [63] and [102] the contention of a COTS multicore is bounded for the bus and the memory respectively. A methodology that replaces tasks with kernels that mimic their shared resource usage is used in [21] as a way to reduce the variability and improve measurement-based analysis. Authors in [64] abstract the contention tasks cause and suffer using signature and templates that allow bounding the contention [63] suffered by a task from its contenders.

Finally, is it worth mentioning cache partitioning, either by software [122, 103, 172, 131, 34] or hardware [105, 138], is a common solution in the context of CRTES [29] due to the complexity of estimating WCET accurately on top of shared caches. Nevertheless, shared caches can be found in different contexts such as probabilistically analysable systems [163], static deterministic timing analysis [33], measurement-based techniques on top of COTS processors [61] and the *joint analysis* examples cited above.

In this thesis we focus on hardware solutions that simplify the timing analysis for multicore processors with shared caches partitioned by hardware. In particular, this thesis focuses on providing solutions for the two following components: bus and memory controller.

2.2.1 On-chip buses

Buses in real-time systems are used for off-chip and on-chip communication. Examples of off-chip buses are CAN [90] and FlexRay [66] used in automotive, Spacewire [55] used in space or the Time-Triggered Architecture [108]. Those buses are designed to connect different processing units and peripherals. Off-chip bus contention can be handled with TDMA buses [158] assuming the worst possible alignment of the task requests with their TDMA slots or with dynamic arbiters [157] that consider the particular pattern of accesses of each task to the bus [50]. Transactions on those buses are visible to the software, which enables scheduling the requests of the different running tasks so that interferences in the use of the bus can be avoided.

This thesis focuses on on-chip buses, which are used at much lower granularity than off-chip buses. For instance an on-chip bus is used to communicate cores and the L2 cache in the NGMP processor [42]. At this granularity, the requests (e.g., L2 cache accesses) of the different tasks cannot be efficiently scheduled to prevent interferences. The responsibility to handle interferences is left to the hardware, in this case to the arbiter of the on-chip bus.

Most of the previous work on on-chip bus architectures for real-time multi-cores like [151, 138] focuses on the bus arbitration policies. These policies vary from TDMA [151], Round-Robin based [138] or budget schedulers [8]. In [101] they consider the effect of a shared on-chip TDMA bus on the timing analysis. Also variations of these policies can be found: in [25, 26] they proposed the MBBA scheme that uses grouping [138] on a Round-robin policy to reduce WCET estimates. In [160], they use Priority Division arbitration, which behaves like a TDMA schedule with different priorities on each slot, so that no slot remains empty when there are requests to be sent. In [159] authors propose a method based to derive WCET estimates with measurements, when the maximum delay for a round-robin bus is known [138].

The assumption of a generic simple bus that has a known bounded (or fixed) transaction latency is common across these works. As we show in this thesis, this is not always the case in real implementations, as happens for the AMBA bus. The Advanced Microcontroller Bus Architecture (AMBA) [67] is one of the most – if not the most – broadly used bus interfaces. AMBA is used in a wide range of architectures, providing flexibility in the implementation and backward-compatibility with existing AMBA interfaces. AMBA is increasingly being used in multicore processors for real-time industry, e.g. LEON3-based GR712RC [36] and LEON4-based NGMP [42].

To our knowledge, existing studies on AMBA buses focus on RTL models and efficient implementations of the different AMBA interfaces. As an example, [45] analyzes several arbitration algorithms for AMBA AHB in terms of latency and power dissipation. However, no work considers the AMBA specification for real-time time-composable systems.

In this thesis we evaluate the most appealing bus arbitration policies for CRTES and propose a time-composable AMBA AHB interface that provides time composability by design.

2.2.2 Memory controller

Modern DRAM systems [92] comprise a memory controller and DRAM memories, which are usually organized into SIMM or DIMM modules that contain the DRAM devices or chips. The memory controller acts as an interface between the DRAM memories and the processor.

DRAM memories are organized into *channels*, *ranks*, *banks* and *arrays*. The processor accesses the memory through one or more independent memory *channels* with separated command, control and data buses for each channel. Each memory *channel* consists of one or several *ranks* that can be accessed in parallel through the same memory bus. A *rank* consists of several DRAM devices or chips connected in parallel. Since DRAM devices have narrow data width, several of them are needed

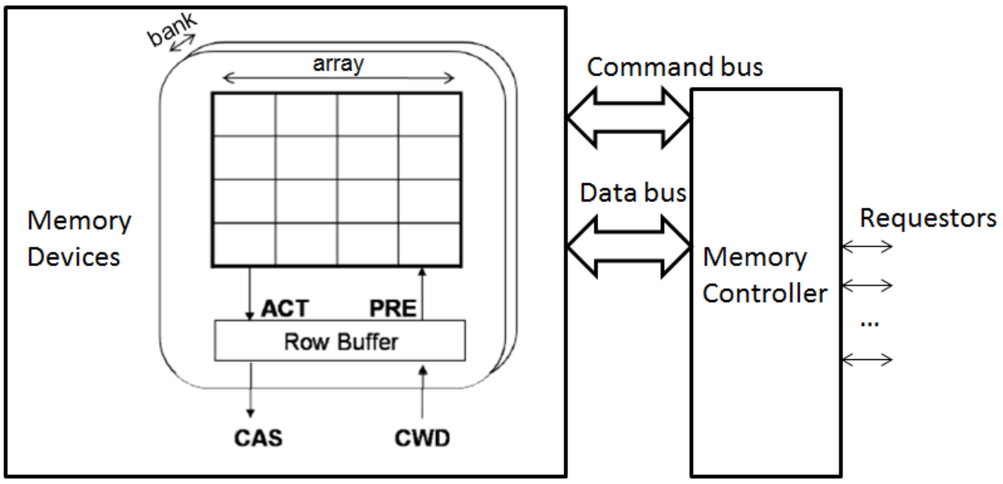


Figure 2.1: DDRx DRAM memory system.

to provide a wide data bus, e.g. 8x8 bits DRAM chip gives a 64 bit memory bus width. Every DRAM device contains several memory *arrays* organized into *banks* that can be accessed in parallel. Since memory operations take several cycles, different banks, as well as different ranks, can be accessed simultaneously, which is called the *Memory Level Parallelism* (MLP). An scheme of a DRAM device organization is shown in Figure 2.1.

DRAM devices require several commands to operate them due to their internal behavior. To serve a memory request, an entire row from a bank has to be loaded on the per-bank row buffers, which is done by issuing an *activate* (ACT) command. This action is also referred as "opening a row". Once the row is on the row buffer, a column *read* (CAS) or *write* (CWD) can be issued to get the data. If the next request targets the same row that is open on the row-buffer, column read or write commands can be issued directly. Otherwise a *precharge* (PRE) command needs to be issued before activating a different row, to write back the open row to the memory arrays, which is also called "closing the row". Also, in DRAM memories, all memory rows need to be periodically read out and restored for data integrity due to leakage in memory cells. This is done by issuing a *refresh* (REF) command. The impact of memory refreshes on execution time is limited and can be bounded as shown in [17, 20].

There are two different ways to manage memory rows, also called pages, from the point of view of the row buffer: (1) close-page policy that precharges a row immediately after the column access and (2) open-page that leaves the row open on the row buffer to exploit the locality of future accesses, called *Row Buffer Locality* (RBL). In a close-page policy, all requests perform the same actions: activate, column access and precharge. In an open-page scheme, depending on if the access is a *row-hit* or a *row-miss*, a request behaves differently. If the access is a *row-hit* it accesses the same row as the previous access, and hence it can directly perform the column access. In the case of a *row-miss*, the request has to precharge the actual row and activate the new one before performing the column access.

All commands sent to the DRAM devices have to satisfy the timing constraints specified on the JEDEC standard [96], depending on the type of memory. The most important timing parameters are the column read latency t_{CAS}, write latency t_{CWD}, activate latency t_{RCD} and precharge latency t_{RP}. Annex II in [95] provides a detailed list and definition of the timing constraints for a DDR2/DDR3 memory [92] used on this thesis. A complete list of the timing parameters can be found in [96].

The memory controller is in charge of scheduling the different requests coming from the same or different processors and translating the requests into the appropriate commands. The *Memory Mapping Scheme* (MMS) defines the mapping of physical addresses from the processors to the actual memory blocks in the memory devices. The MMS impacts both MLP and RBL. For instance, if the MMS maps sequential addresses to the same row, it benefits the RBL. Instead, if it maps consecutive addresses to different banks, they could be accessed simultaneously, exploiting the MLP.

Real-time Memory Controllers

The timing analysis of a memory controller depends on three main design choices that affect a request's latency: (1) The *row-buffer policy*, (2) the *MMS* and (3) the memory request *scheduling policy* [92]. Each of these design choices has an impact on determining the upper bound latency of a memory request, required in real-time domains. The row-buffer policy states how to take advantage of the row-buffer locality, which allows consecutive request targeting the same memory row to have smaller latency. However, when those consecutive request target different rows, they pay a penalty in terms of latency. The memory mapping scheme defines how addresses are mapped to physical devices and allows to exploit memory level parallelism since several memory banks can be accessed in parallel on a memory system. The scheduling policy defines how to schedule requests from the same or different requestors and impacts how inter-task interferences appear.

Row Buffer Policy. Most real-time memory controllers [7, 75, 140, 148] implement a close-page policy as row-buffer policy to ensure that memory banks are in the same state after every request, independently of the bank and row previously accessed, reducing the memory jitter. Time predictability provided by close-page comes at the cost of preventing the exploitation of spatial locality of multiple requests accessing the same row. In order to address this issue, [74] presented a variation of the close-page policy, named conservative open-page policy, in which multiple requests to the same row are allowed to be issued while the row is open in a close-page policy. In this case, the locality can be exploited only during a small time-window in which the close-page policy keeps the row opened due to timing constraints. The worst-case is not affected, which remains the same as with normal close-page, thus maintaining the predictability in the worst-case and increasing the memory bandwidth for the average-case. Open-page, which exploits row-buffer locality, is also used in real-time systems taking into account the effect of row-hits and row-misses [139], usually assuming private banks [183], which remove the dependence from other tasks. Open-page is also used in COTS processors, that can be analyzed to obtain the worst-case latency [102].

MMS. A common choice in real-time designs is the interleaved bank scheme [7, 75, 140, 74] in which each request accesses all banks exploiting bank-level parallelism and reducing bank conflicts among memory requests. Private bank schemes are also used [148, 183], which remove the bank conflicts across requestors.

Memory scheduler. In real-time systems the scheduler is designed to bound the impact of interferences among memory requests coming from different cores. To do so, the scheduler is based on the core the request has been issued using CCSP [7], TDMA [75, 74, 148] or round-robin [140] arbitration policies. All these techniques allow deriving the maximum delay a request may suffer due to interferences when accessing the memory. In [102], authors derive bounds on the interference with FR-FCFS [150] and in [183] they use a FIFO policy instead of FR-FCFS, removing the reordering effect in order to be able to derive tighter bounds on the request latency.

In this thesis we propose a memory controller that can provide high performance and time predictability for performance-driven and real-time tasks respectively, at the same time. We also explore new solutions that remove contention on the access to memory by physically separating the data buses.

2.2.3 Accounting for Multicore Contention

Performance Monitoring Counters (PMCs) have been traditionally used to measure average performance and power consumption [125]. For instance, the IBM POWER family, starting with the POWER5, have developed a Cycles Per Instruction (CPI) stack based on PMCs that covers the resources on which each task spends its cycles. The CPI stack reports the cycles spent in each core resource such as the load/store Unit (LSU), which happens when a load/store operation is stalled [98]. One of the few works that addresses contention monitoring between tasks is [189], which uses cache scouts to monitor contention on shared caches. In [159] authors use custom PMCs to derive the Worst Contention Delay (WCD) and WCET estimates with measurement-based timing analysis on a bus-based system. Authors assume that the WCD for the bus is known, which is not always the case in real implementations, as shown for the AMBA AHB bus [94].

The contention in multicore processors has been characterized mostly using Resource-Stressing Kernels (*rsk*) [144] that have been used to expose the contention on certain resources of an architecture. In [65] authors use them to characterize the NGMP [40] processor or in [134] to characterize the Freescale P4080. This approaches based on *rsk* are criticized due to the low confidence that can be obtained with *rsk* [1]. Authors in [63] increase the confidence that can be obtained with measurements based on *rsk*.

Authors in [135] propose a *runtime monitoring* to control the resource usage of tasks running on a multicore, preventing tasks from having resource usage limit violations. Authors make use of access count PMCs, such as bus access counts. However, it has been shown that bus latencies may differ across different types of accesses and even for the same type [94]. As a result, access counts do not provide the actual impact of contention time which has to be estimated. The ACD instead provides in an exact manner contention delay for each task.

This thesis provides means to evidence the contention of the processor based on a new set of PMC. We also follow the theoretical approach in [64] that proposes a methodology to obtain the resource access 'profile' of a given task that defines the use of resources that the task makes on a target shared resource. That profile is used to derive the contention tasks suffer and generate when accessing that resource.

Chapter 3

Methodology

The methodology used along this thesis consists in collecting execution times of tasks running on the processor and obtaining WCET estimates. For that purpose, different tools and methods are used. In the following sections we present the main components that allow to extract this information: the processor, the benchmarks considered, the simulator and how to obtain WCET estimates.

3.1 The NGMP multicore processor

The *Next Generation Microprocessor* (NGMP) architecture [42], shown in Figure 3.1, whose latest implementation is the GR740 [39], is the reference architecture considered in this thesis. The NGMP is envisaged by ESA as the main computing platform for its future missions. The NGMP is a quad-core multiprocessor system-on-chip (MPSoC), based on the LEON4 SPARCv8 [89] architecture. The four LEON4 cores are connected with an AMBA AHB bus to a shared level 2 cache. After the L2 cache, the off-chip DRAM memory is accessed through another AHB bus and the memory controller. Some write-buffers exist somewhere, not specified, in the memory hierarchy of the processor. The rest of the NGMP's functional units consists of debug support units and I/O peripherals. In this thesis, we do not consider I/O related activities, which we assume managed at software level.

LEON4 core: As shown in Figure 3.2, each LEON4 core comprises a seven-stage scalar in-order pipeline and private instruction and data first-level caches. The pipeline consist of 7 stages: *fetch*, *decode*, *register*, *execute*, *memory*, *exceptions* and *commit*. Floating point operations are done in a separate functional unit. The instruction and data caches are accessed by the *fetch* and *memory* stages respectively. Both caches have 4 ways and 16KiB (32 bytes line) and the data cache is writethrough.

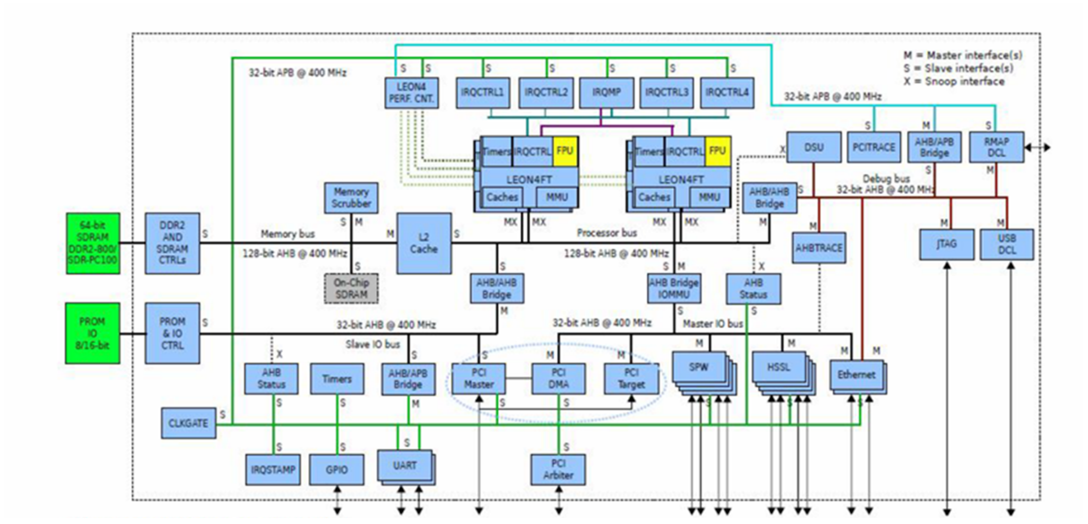


Figure 3.1: NGMP architecture. Source [42].

AHB Processor Bus: The 128 bit AMBA AHB bus [14] connects the LEON4 cores to the L2 cache and the I/O bridges. The first consideration to make in the case of the bus is that there are different types of requests that have different behavior: bus reads (loads) that either hit (*l2h*) or miss (*l2m*) on the L2 cache and bus writes (stores) that either hit (*s2h*) or miss (*s2m*) on the L2 cache. These accesses behave differently because hits hold the bus while they are served. Instead, misses wait on a miss queue and are split, i.e. the L2 cache releases the bus while processing the miss, so that other cores can use the bus. Also writes are immediately responded by the L2 cache, that keeps them in a queue until they are processed. In the NGMP, the AMBA AHB bus implements round-robin arbitration.

L2 cache: The 256 KiB, 4-way, write-back second-level cache is shared across the four LEON4 cores. In this thesis we use the master-index feature of the NGMP that partitions the L2 assigning one L2 cache way to each core. Hence, a given core suffers no contention interference in the L2 due to other cores' evictions. Each of the request types identified before (*l2h*, *l2m*, *s2h* and *s2m*) has its own L2 access latency. Interestingly, the latency of requests of the same type can be variable. That is, for each request type access there is jitter, which is caused by the previous requests, despite they might belong to a different task and hence go to a different cache partition.

Memory Controller The memory controller acts as an interface between the processor and the DRAM memory and can be modelled as a FIFO queue that uses a close-page policy to access the memory. We differentiate two types of request in the memory: read and write. According to the DRAM protocol, each

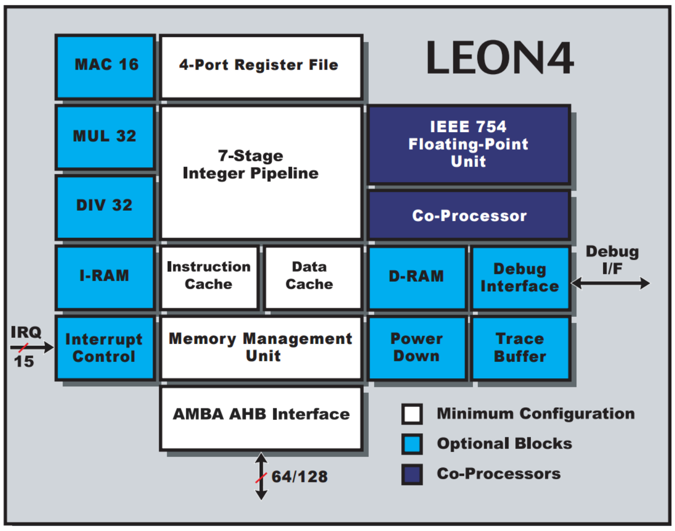


Figure 3.2: LEON4 structure. Source [43].

request has a different latency to be responded depending on whether it is a read or write request respectively. The off-chip memory used is DDR2 [96] memory.

3.1.1 NGMP implementations

We use two different processor boards, the commercially available GRN2X [41] and a FPGA prototype of the GR740 [39] which was provided directly by the designer company. These boards have two different uses, the most important one is to provide a reference model for the validation of the simulation platform, as it is explained on Section 3.3.1. The second one is to test and account contention as shown in Chapters 9 and 10.

To use the boards, the GRMON2 [37] commercial debug tool is used to interface with the boards and obtain the measurements from the available Performance Monitoring Counters (PMC).

3.2 Benchmarks

We use three families of benchmarks: representative for the real-time community: EEMBC [143], real space applications obtained from the ESA and microbenchmark or synthetic kernels that are designed to stress certain behaviors on the processor. The individual benchmarks are described in Table 3.1.

3.2.1 EEMBC automotive benchmark

The EEMBC automotive benchmark suite is a well-known benchmark suite for CRTES in the automotive domain, and it is particularly useful for evaluating the capabilities of embedded microprocessors, compilers, and the associated embedded system implementations. Experiments shown in [143] illustrate the diversity of the EEMBC benchmark suite as well as the specifics of each workload's activity. This diversity ensures that designers can use combinations of EEMBC workloads to represent most real-world workloads and use this characterization data as a starting point to make effective design choices.

Table 3.1: Benchmarks used in this thesis

EEMBC Autobench	
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
Space applications	
obdp	Near infrared HAWAII-2RG detector algorithm
aocs	Attitude and Orbit Control System (AOCS) from the EagleEye project
debie	DEBIE instrument control software from PROBA-1 satellite
vega	Thruster vector control of the Vega launcher
Microbenchmarks	
rsk	Synthethic kernels to trigger different processor behaviors

3.2.2 ESA real applications

As real applications we use some software provided by the ESA. As payload applications we use the On-board Data Processing (OBDP) and DEBIE benchmark. OBDP contains the algorithms used to process raw frames coming from the state-of-the-art near infrared (NIR) HAWAII-2RG detector [97], already used on real projects, like the Hubble Space Telescope to detect cosmic rays. DEBIE is the software that controls an instrument, which was carried on PROBA-1 satellite, to observe micro-meteoroids and small space debris by detecting impacts on its sensors, both mechanically and electrically. As control application we use the Attitude and Orbit Control System (AOCS) from the EagleEye project [23] and the thruster vector control of the Vega launcher (VEGA). The AOCS contains the Guidance and Navigation Control system from the spacecraft in charge of the correct position and orbit of the spacecraft. It is one of the most critical systems of a spacecraft, since a wrong position or orbit could mean the complete loss of the spacecraft, due to loss of power (not pointing to the sun for solar powered spacecrafts) or communication (antennas are directional and have to be properly oriented).

3.2.3 Microbenchmarks

We use microbenchmarks or Resource-Stressing Kernels (RSK) [144][31] comprising a single loop with instructions of the same type that are chosen to stress a certain resource: either an instruction or a hardware resource. The reason to use a loop is to avoid icache misses and exercise the latency or certain behaviors in the processors long enough to be able to minimize the overhead of instrumenting and measuring the execution time. Each loop iteration will contain N instructions. RSKs are designed to reduce the overhead to a minimum by developing them in assembly code and compiling for bare-metal execution, which means that no OS is present.

3.3 Simulator

We build a model of the NGMP simulator upon the SoCLib [164] simulation environment, which we properly modify to resemble the NGMP processor. The purpose of the simulator is to have a cycle accurate simulator environment, in order to correctly measure the execution cycles. However, our goal is not to model all the complexity of the processor, because there is a tradeoff between complexity (or simulation time) and accuracy. In case some accuracy has to be traded off for a more simple model, simplicity is achieved by modelling the most common case.

The simulator is conceptually designed to separate the functional part from the timing behavior, which creates two different design spaces: (1) the *functional emulator* (emulator from now on) and (2) the *timing simulator*. The simulator structure is shown in Figure 3.3. The emulator part executes the instructions according to a particular Instruction Set Architecture (ISA) and provides all the information about an instruction like the instruction address, registers, type, re-

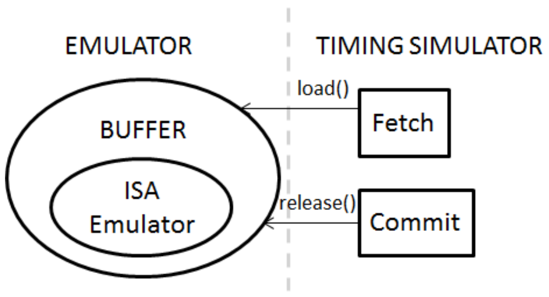
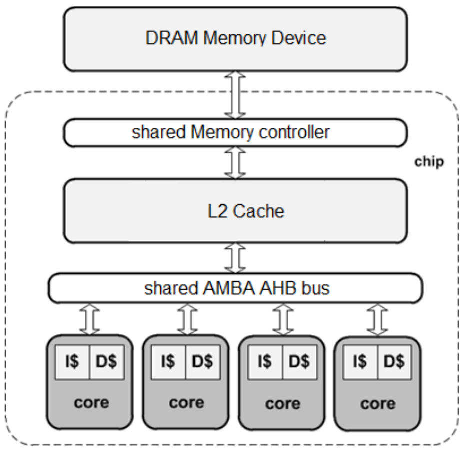


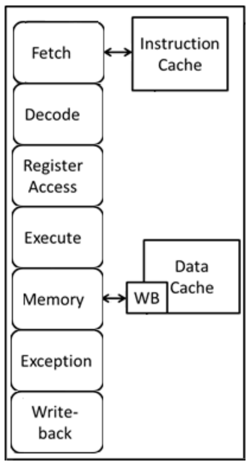
Figure 3.3: Simulator structure

sults and memory address in case it is a memory operation. The timing simulator (simulator from now on) simulates the timing behavior of the instruction for a given hardware implementation, for instance, if it is a cache hit or miss and the delay introduced by the bus access in case it has to reach a higher level cache or the memory. The architecture of the simulator serves two purposes: 1) to resemble the actual processor structure which helps to have a more accurate modeling of its behavior and 2) to ease implementation of new hardware features such as bigger caches.

The simulator architecture, as shown in Figure 3.4a, consists of 4 cores with their respective private caches, connected through a bus to a shared L2 cache and the shared memory. All the caches are configured to resemble the caches in the NGMP. The bus behaves as a 128-bit, round-robin AMBA AHB bus. DRAMsim2 [171] is used for the memory, which is a well known C-based memory system simulator developed by the University of Maryland. It is highly configurable, parameterisable and implements detailed timing models for different types of existing DRAM memory systems. With DRAMsim2 we model the memory controller and a 2-GB one-rank DDR2-667 [104] with 4 banks, burst of 4 transfers and a 64-bit memory data bus, which provides 32 bytes per access, i.e. a cache line, as in the real processor setup used along this thesis. The internal structure of the core is depicted in Figure 3.4b, which comprises the 7-stage pipeline with private caches and a write-buffer. Both pipelines, integer and floating point, are embedded in the execution stage, that assigns latencies to instructions. The timing of an instruction consists of the instruction latency, that is defined by the pipeline, and the memory hierarchy latency for the instruction access and data access, in case of load and store operations. The memory hierarchy latency includes a third contributor, the multicore contention, which involves the effect that the rest of the cores can have on the timing of the core being analyzed, such as waiting for the bus because it is being used by another core.



(a) Multicore architecture



(b) Single core details

Figure 3.4: Simulator architecture

3.3.1 Validation Methodology

Our validation methodology is based on *microbenchmarks* or RSK (see Section 3.2.3) with a tow-fold objective: Providing confidence on the fact that the timing simulator is accurate by comparing their execution time on the simulator and the real board, and if the execution time is different, it helps adjusting the timing parameters of the simulator to improve accuracy for the specific components used by the microbenchmark.

General RSK. Each RSK comprises a single loop with instructions of the same type that are chosen to stress a certain hardware resource or a set of them. The first step is factoring out the execution time of the loop overhead in both, the timing simulator and the real board[1]. This overhead is removed from all the execution time readings to obtain the exact latency of the operations performed in the loop body. The latency of the body helps tuning the latencies of the different parts of the processor with very high accuracy. The loop overhead is measured with a loop containing only the loop-related instructions, which include roughly an arithmetic instruction to compute the loop index and a branch instruction, and also possibly a comparison instruction in other ISA. The latencies of these instructions can be obtained from the datasheets, if available. Otherwise, they can be derived empirically. For that purpose, we use a loop containing N times the instruction whose latency is to be measured, that is programmed in a way that IL1 misses can only occur in the first loop iteration, i.e. the loop fits in the

[1]When we talk about real board, a RTL level cycle-accurate simulation could also be used.

IL1. This way we can approximate the latency of the instruction analyzed very accurately, since the overhead of the loop is minimized. Once each instruction latency is obtained we can set up the simulator with the appropriate latency for each instruction type.

In terms of implementation, we proceed as follows.

Instruction timing. We (automatically) create a set of RSK, each one having a different instruction type under analysis in the loop. All instructions are forced to incur cache hits in order not to include the memory hierarchy latency on the execution time. First, we measure the execution time of the empty-loop RSK. This execution time needs to be substracted from the execution time of any of the RSK that we produce to analyze any instruction. Then, the execution time difference between the specific RSK and the empty-loop RSK is divided by $N \cdot M$, where N is the number of instances of the instruction under study in the loop and M the number of iterations carried out. In order to validate the behavior of each instruction type, as we will see in the results Section 3.3.2, the execution time of the loop obtained in the simulator and the real board have to match for each type.

For branch instructions, in case there is a different latency for taken and non-taken branches, the RSK can be adjusted to use non-taken or taken branches by setting the branch address to the exactly sequential instruction so that the control flow is exactly the same and they can be modeled separately. In our case, both cases behave equally. For input-dependent instruction latencies, as it might be the case for divisions or multiplications, several tests covering the different cases can be used to profile the different instruction latencies of that instruction type.

Memory hierarchy. As next step, we address the memory hierarchy by testing all types of hits and misses on it. For that purpose, the instruction loop generates hits or misses on the instruction, data L1 and L2 caches. The same procedure followed here can be extended for processors with more hierarchy levels. All caches in our processor have LRU replacement policy, which is the most common in practice. To generate misses we perform $W + 1$ accesses, being W the number of cache ways, to the same set on each loop iteration. For instance, in the NGMP, caches have 4-ways so performing 5 accesses to the same set for different addresses causes that the first 4 accesses fill the 4-ways of the set and the 5th one evicts the 1st one from the set. When the access sequence repeats, the 1st address misses and evicts the 2nd address, which misses in turn and evicts the 3rd address and so on and so forth. The memory accesses systematically evict subsequent data to be accessed next and thus all accesses miss in the cache systematically. An easy way to access the same set is making cache accesses have an address offset equal to the way size of the cache, i.e., for the 16K L1 cache, 16K/4 ways = 4K. The resulting RSK is shown in Figure 3.5. By doing this we guarantee that we miss on that cache, but we hit on the bigger next level cache. To generate hits

```
for i = 0 to iterations - 1 do
    ld [0x00000000], $0
    ld [0x00001000], $0
    ld [0x00002000], $0
    ld [0x00003000], $0
    ld [0x00004000], $0
end for
```

Figure 3.5: RSK example with *loads* with 4K offset

on a certain level of cache, we use the same procedure, making a loop that misses on the previous level. In the case of the L1 caches, accesses to the same address generate hits. For instruction misses, we use five branches physically separated by the required offset that jump sequentially. For data misses we use load and store operations with the given address offset. This benchmark structure is particularly devised for LRU and FIFO replacement policies. For other types of replacement policies similar structures can be built to produce systematic cache misses.

Multicore contention. As last step, we address the multicore contention on the shared bus. To that end, we use a recently published method [63] that allows to derive the latency caused by the contention on the bus. This method allows us to expose the interference that the cores generate for different types of request in a given shared resource. We use two different RSK at the same time, a sensitive RSeK and a stressing RStK. Both RSK perform continuous accesses to the shared resource under analysis by using the same technique presented before to miss in the appropriate cache levels. The RSeK runs on the core under analysis and the RStK run on the rest of cores, thus creating a high contention scenario. The method explained in [63] shows that round-robin arbitration behaves as a time-multiplexed scheme under a high contention scenario. In this situation, the access time of the RSeK accesses with respect to the time-multiplexed window can be varied by inserting a variable amount of nops between RSeK accesses to the shared resources. For each access time, the interference experienced has a different value and follows a sawtooth behavior. The maximum interference delay matches the frequency of the sawtooth. Similar analysis can be carried out in the memory, however, in our case, the interference occurs mostly on the bus because the bus serializes the traffic by stalling accesses until the request being processed is served[2].

After the contention on the bus is properly adjusted, we run tests based on a RSK that uses a large fraction of the L2 cache on each core to create interference

[2]This is caused because the board used for the experiments does not use split transactions on the processor bus. Other boards, such as the GR740 [39] use split on the bus. In general, we assume that the processor uses split on the bus despite such processor was not available for the validation.

between the cores. These can be adjusted to interfere only on the bus or the memory by exceeding L1 and L2 capacity respectively only when run together. For instance, given N cores, each RSK can be designed so that the cache space used, C_i, matches the following constraint: $\frac{L2size}{N} < C_i \leq L2size$.

3.3.2 Validation Results

In this section we describe the process followed to adjust the timing parameters and the validation procedure. The processor we use as real reference, a.k.a. *board*, is the commercially available GRN2X [44] board. In order to get an accurate measure of the execution time, we use hardware PMC that are available in almost every processor architecture. More precisely we use the "execution time" counter, which is one of the most common PMCs. Additional counters that are useful are cache hit and miss counters and cache, bus and memory access counters. These counters can be read directly using GRMON [37] debug tool from the processor vendor[3]. In all cases, we measure the PMC of interest for the execution of the loop, that is, removing all the overhead of the rest of the program. This can be done by reading the PMCs strictly before and after the loop. The difference between both readings corresponds to the value of the loop itself.

Instruction timing

As first step in our methodology we run one RSK for each instruction in the SPARCv8 ISA in the simulator and the board. Available parameters from the processor manual [42] have been used to adjust the instruction latencies before the experiments. We group the instructions according to their different nature: load, store, integer/float, short/long and branch. Figure 3.6 shows the results of the instruction validation test for the different instruction types. All results are normalized with respect to the *nop* case. We performed tests for all instructions, except for the ones classified as *special* due to their complexity, impact or dependency on the processor. Furthermore, *special* instructions, such as writing into special processor registers, appear infrequently on a normal execution. We also did not include the detailed analysis of the floating point instructions since they have their own hardware component (FPU), present a much more complex behavior than the rest of instructions and their common-case behavior is well documented in their specification. We can observe that the simulator can accurately model instructions executed within the core incurring all cache hits. There is a significant difference not shown in the figure for division instructions, classified as *int long*

[3] Note that these types of PMC and mechanisms to access them can be found in most processor designs, thus allowing to port our methodology to other architectures and processor models.

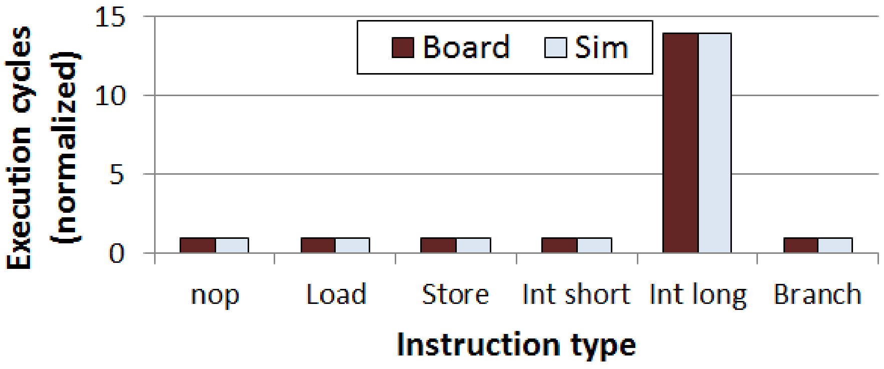


Figure 3.6: Instruction timing

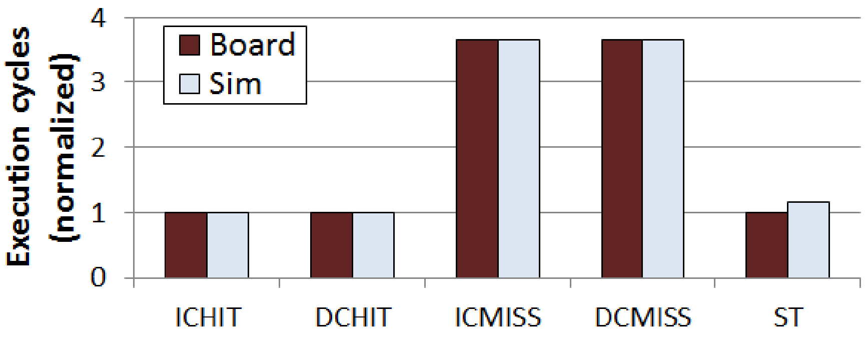
(together with multiplications), due to input data dependencies. In our case we take the worst-case value, to provide an upper-bound of the execution time, in case some inaccuracy is introduced. This can also be mitigated, if needed, by building a table of latencies indexed with the input values for division (i.e. dividend is zero, divisor is 1). In our case we decided to keep the simulator simple.

Memory hierarchy

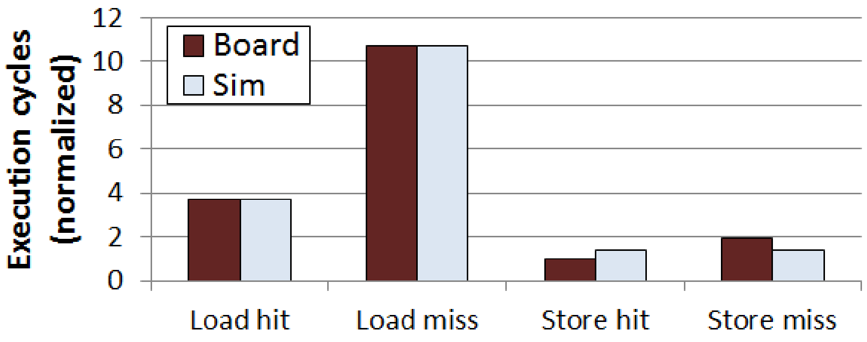
In this step we validate the behavior of the memory hierarchy using RSKs that hit or miss on a given cache level. There are two metrics involved for each memory hierarchy level: number of accesses, such as hits, misses and bus accesses, which are used as a sanity check to see that the RSK does what was expected; and latency measurements used to validate the timing behavior. For the sake of convenience, we only show the latency analysis but similar analysis was done for the accesses, given that both analyses provided analogous conclusions. We begin with L1 instruction and data caches. Figure 3.7a shows the results for load hit and misses in both L1 caches, normalized w.r.t. the execution of *nop* operations. Store operations in the L1 data cache, which is write-through, behave exactly the same whether they hit or miss since data are forwarded to the next level anyway.

There is a source of inaccuracy from our model that can be seen on this figure. Store operations are not exactly modelled due to the presence of one or more write-buffers that are not accurately described in the documentation. Nevertheless, most of the time the latency of a store is equal to a nop operation because the write-buffer is effectively masking store latency, so we only model one write-buffer.

In the L2 cache, we have four different types of accesses: loads and writes that either hit or miss. Loads can be instruction or data requests, since the L2 cache is unified. Figure 3.7b shows the values obtained for the different RSKs that generate each type of request. We can observe again the difference for store operations. The



(a) L1 caches access



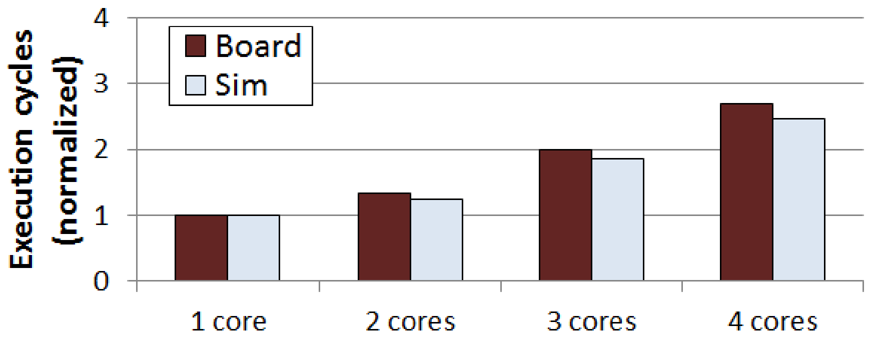
(b) L2 cache accesses

Figure 3.7: Memory hierarchy timing

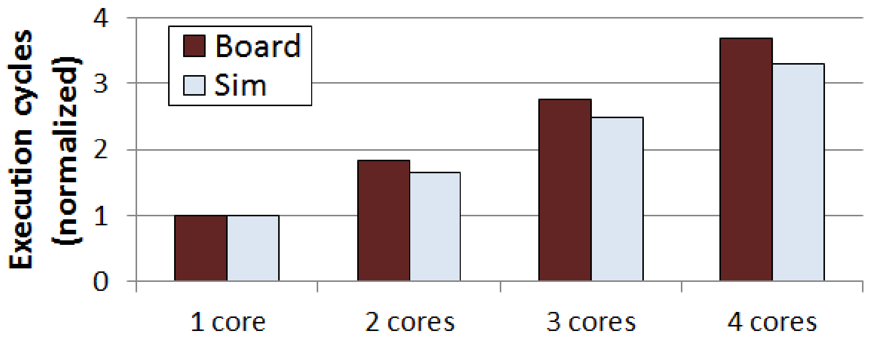
L2 cache case is more complicated, since according to the manual [42], L2 latencies are variable and depend on previous requests that were accessing the L2 cache, which can come from any of the 4 cores. However, this behavior is complex to model or test, since requires cycle-level control of the contention on the L2 cache which is impractical in reality. In this case, we choose to model a fixed latency that corresponds with the case of no previous request, which reduces the accuracy but simplifies the L2 model.

Multicore contention

We use the same technique that was presented in [63] to adjust the impact of contention on the bus. This technique allows to obtain the worst-case interference for round-robin buses. We applied the technique for each different type of access on the bus and the memory to obtain the interference delay and adjust the simulator properly for each request type. Once adjusted, we run experiments to exercise



(a) Execution times for L2-half benchmark



(b) Execution times for L2-full benchmark

Figure 3.8: Multicore contention with different number of cores

the interference and see the matching. In the actual processor, contention occurs mainly on the bus due to the lack of split transactions. For that purpose, we run benchmarks competing with the rest of the cores for the L2 cache to test the accuracy of the contention modeling. Figure 3.8 shows the normalized increase on execution time for two benchmarks accessing L2 executed in a different number of cores. In the case of Figure 3.8a, each benchmark accesses half L2 capacity, and so contention becomes much more significant when using 3 cores, thus exceeding L2 capacity. In Figure 3.8b, almost all L2 capacity is accessed by each benchmark, which causes that when executed in 2, 3 or 4 cores, the execution time increases much more due to the contention. We observe that some inaccuracies, mainly due to the L2 cache variable latency, are already present.

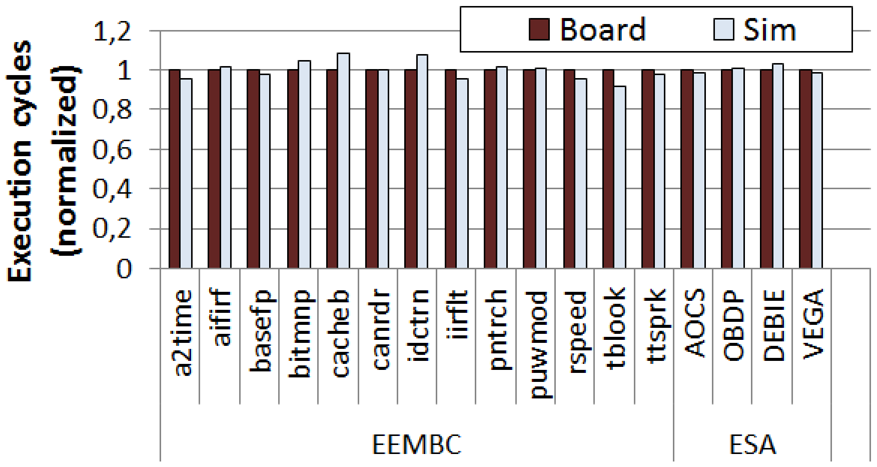


Figure 3.9: EEMBC and ESA benchmarks accuracy results.

3.3.3 Final accuracy results

In this section we show how the simulator performs with different benchmarks and real applications to show its accuracy. For the validation with general benchmarks we use the EEMBC Autobench suite [143] and the real applications provided by ESA: AOCS, OBDP DEBIE and VEGA (see Section 3.2).

Figure 3.9 shows the accuracy, at cycle-level, for EEMBC automotive benchmarks and the four ESA applications. In this chart we can clearly see, that our simulator offers accurate results, with 3% of error on average. The loss in accuracy is caused by the design choices of not exactly modeling some of the behaviors. For instance the variable integer long instruction latency, variable L2 miss latency and the write-buffer. Also the difficulties to properly control and measure the contention in multicore workloads, makes their modeling difficult.

Our simulator is able to reach up to 1 MIPS. If we compare it with the performance of the actual board (around 100 MIPS for the benchmarks), we observe that the tradeoff between accuracy and performance allows us having a detailed timing model of the processor at a moderate performance cost. There is still room to improve the execution time of the simulator by tuning its code, however this is beyond the scope of this thesis.

3.4 Timing Analysis

In this thesis we use two different techniques to obtain WCET estimates, depending on whether deterministic or probabilistic methods are used (see Section 2.1 for a description of both).

Deterministic Timing Analysis: This thesis uses measurements directly collected from the simulator or processor boards and RapiTime [146], which combines the measurements with control flow analysis information to compute WCET estimates.

Probabilistic Timing Analysis: In this thesis we use measurement-based probabilistic timing analysis (MBPTA) [46], as it is closer to industrial practice. We collect execution time measurements of end-to-end runs, testing that the data fulfills the i.i.d properties for random variables using the following tests: the two-sample Kolmogorov-Smirnov (KS) [59] test evaluates the fulfillment of the identical distribution property, and the runs-test [27] the fulfillment of the independence property. Then we apply the MBPTA methodology as explained in [46].

3.5 Summary

This chapter introduced the methodology used along this thesis. The main tool considered in this thesis is a cycle-accurate simulator to perform the design space exploration of hardware components. The most important metrics on this thesis are the WCET and execution time that measure the time that a task spends running on the processor in the worst-case and average-case respectively. Therefore, the simulation must be accurate enough so that execution times collected can be used to derive trustworthy WCET estimates. This thesis also considers real processor boards based on ASIC or FPGA technologies to validate the simulator and evaluate some proposals. The simulator is used to implement custom hardware features not available on the commercial version of the processor. The software used corresponds to well-known benchmarks and real applications from the space domain.

Chapter 4

Time-Composable Bus

4.1 Introduction

One of the most important shared resources in current MPSoC for real-time systems is the backbone bus that connects the different cores with the memory/cache subsystem (and possibly other devices or subsystems). The Advanced Microcontroller Bus Architecture (AMBA) [67] is one of the most – if not the most – broadly used bus interfaces. AMBA is used in a wide range of architectures, providing flexibility in the implementation and backward-compatibility with existing AMBA interfaces. This chapter focuses on the Advanced High-performance Bus (AHB), one of the distinct buses defined in the AMBA specification, which aims at high-bandwidth, low-latency, high-frequency and low-complexity. AMBA AHB (or simply AHB) is increasingly being used in multicore processors for real-time industry, e.g. LEON3-based GR712RC [36] and LEON4-based NGMP [42], so providing tight and time-composable bounds to the access latency becomes a desirable property for AHB to deliver. Unfortunately, AHB was not designed with time composability in mind that exists when the timing properties of a software component in isolation, i.e. its WCET estimate, do not change when the system is integrated. In order to make AHB-based MPSoC systems to fulfill the desired time composability property, this chapter makes three contributions:

Contribution 1. We provide a detailed analysis of the AMBA AHB features and their impact on the timing behavior of connected components (i.e. master and slaves), and by inference, on the applications running on the MPSoC. We identify the AHB features that affect the timing behavior of applications and how AHB-compliant masters and slaves can break time composability.

Contribution 2. We propose to use only a restricted subset of the AHB features, named restricted AHB (*resAHB*), such that if master/slaves adhere to it, the maximum delay that any bus access may suffer due to inter-task interferences can be computed, so fulfilling the time-composability property. The main advantage of *resAHB* is that it is compliant with the AMBA AHB specification, providing the same functionality as the original one. Our results show that this solution is attractive only when the AHB-connected components have a considerably higher latency than the AHB arbitration itself.

Contribution 3. We extend *resAHB* by introducing a new set of *operation modes* currently not specified in the AMBA AHB specification. We call this new AHB specification, *Advanced High-performance Real-time Bus* specification or *AHRB*. AHRB efficiently isolates the timing behavior of the different components connected to the AHRB, allowing to derive tight and trustworthy WCET estimates.

AHRB specification ensures that if all connected components follow it, they will enjoy tight and time-composable bounds for their bus access latency. As a result, although the timing behavior of any IP component[1] may be unknown, its effect on the bus is bounded under *AHRB* without any information or requirement from the component at hardware and software levels because timing requirements are already included in the specification. This is of paramount importance in the context of future multi-IP mixed-criticality MPSoC.

AHRB extends AHB by adding *master and slave modes* that allow specifying the use of the bus that each master and slave does, in such a way that it can be taken into account by other master/slaves at design time. This results in tighter (and bound) access times to the bus, which in turn leads to tighter WCET estimates for the applications running on the MPSoC.

Our results show that, unlike the original AHB, both *resAHB* and *AHRB* enable obtaining time-composable WCET estimates for all tasks running in the MPSoC. Those estimates are independent of the other tasks being run simultaneously. WCET estimates resultant of using *AHRB* are tighter than those for *resAHB*, reducing WCET estimates by 3.5x on average. We also observe that in our setup inter-task interferences cause an average performance degradation on the EEMBC benchmarks ranging from 6% to 35% depending on the workload when using a conventional AHB bus. Such degradation reduces with *resAHB* and *AHRB* to 1%-5%.

The chapter is organized as follows: Section 4.2 describes AMBA. Section 4.3 analyzes the effects of AMBA AHB features on timing and provides examples of how AMBA challenges time composability. Sections 4.4 and 4.5 describe *resAHB* and AHRB respectively. Section 4.6 shows the experimental results. Sections 4.7 presents the summary of this chapter.

[1]An IP (intellectual property) component is a block of logic or data that is designed to be ported and reused across different products (ASIC or FPGA).

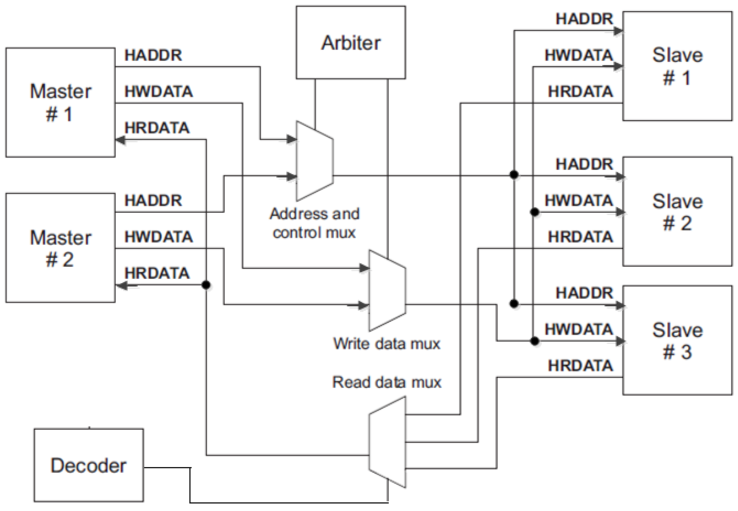


Figure 4.1: AMBA AHB main components: Masters, slaves, arbiter and decoder. (Picture from AMBA Specification Rev 2.0)

4.2 The Advanced Microcontroller Bus Architecture

The Advanced Microcontroller Bus Architecture (AMBA) is a standard bus interface for high-performance embedded microcontrollers, aimed at high-bandwidth, low-latency, high-frequency and low-complexity on-chip communication. Several studies have shown that hierarchical buses scale, in terms of performance and energy consumption, to systems with processor counts in the range 32-64 cores [156, 166]. Thus, AMBA is expected to remain in the near future as one of the standard bus interfaces for real-time MPSoCs.

The AMBA specification [14] defines three distinct buses: the *Advanced High-performance Bus* (AHB), the *Advanced System Bus* (ASB) and the *Advanced Peripheral Bus* (APB)[2]. In this chapter we focus on the AHB, which is used in several existing architectures such as the Cobham Gaisler GR712RC and the NGMP. AHB has been designed to be a high-performance backbone bus that efficiently connects cores, on-chip memories and IP components. The AHB basic architecture, shown in Figure 4.1, comprises four different components: a set of *masters*, a set of *slaves*, an *arbiter* and a *decoder*. These components use or control three main buses: the address and control (HADDR in the Figure), read data (HRDATA) and write data

[2]AMBA 3 specification also includes *Advanced eXtensible Interface* (AXI), an interconnection protocol independent from the interconnection network topology used. It is later covered in Section 4.5.5.

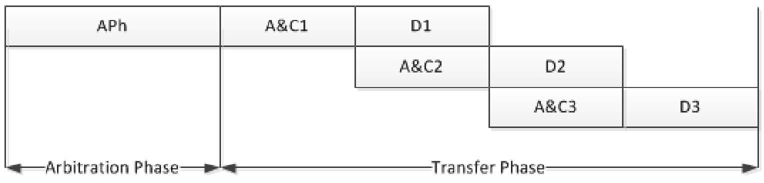


Figure 4.2: AHB transactions consist of an arbitration and a transfer phase. The latter is divided into beats with an address and control phase and a data phase.

(HWDATA) buses.

Figure 4.2 shows an AHB *transaction* between a master and a slave. A master initiates a read/write transaction to a slave by requesting to the arbiter the access to the address and control bus. During the *arbitration phase*, the arbiter handles the contention across masters by granting access to the bus only to one master at a time according to a predefined arbitration policy (not defined by the AMBA specification), so that concurrent transfers are not allowed. Once a master is granted access to the bus by the arbiter, the *transfer phase* starts. The transfer is split into several *beats* if data cannot be sent through the bus all at once (i.e. *burst transfers*). For instance, the transfer in Figure 4.2 requires three beats. Every AHB transfer beat consists of two subphases that overlap across beats, as defined in the specification: an *Address-and-Control phase* that lasts for one cycle in the absence of contention, and a *Data phase* that lasts one or more cycles (in case the master/slave cannot provide the data at that moment). The decoder controls the multiplexers to send the address to the appropriate slave, and two data buses are used to send/receive data to/from the slave. When the transaction finishes, the master relinquishes the buses.

4.3 AMBA AHB and Time Composability

As MPSoCs integrate an increasing number of IP blocks coming from different suppliers and the functionality provided by MPSoCs keeps diversifying, the trend towards MPSoC comprised of multi-party IPs will further exacerbate. Moreover, it is also the case that real-time system IPs may be subject to different safety and security levels. Hence, in order to enable *mixed-criticality functionalities* to be run on the same MPSoC, the impact that one IP component can create on the timing behavior of the others must be limited and this cannot be left to the IP provider, especially to those subject to less-restrictive criticality levels. The safety of the integrated system in terms of timing behavior should be provided by design (construction), in our view, and thus by the bus protocol specification itself. Since time composability is the central element for reducing timing verification and validation costs, it should be provided by design.

Time composability imposes that *the maximum time any request of any task waits to be granted access to the bus is bounded and the bound does not depend on the particular co-running tasks in the MPSoC* [138, 139]. This requires that (1) the arbitration phase for every transaction is bounded and this bound does not depend on the behavior of other components; and (2) the duration of the transfer phase of every transaction is also bounded. Note however, that the arbitration-time and transfer-time bounds may depend on hardware implementation details, such as the number of cores contending for the bus. This however does not jeopardize time composability, since those features are known at design time, when WCET estimates are computed for each task.

If every access to hardware shared resources fulfills the time composability property defined above, WCET estimation for a task, either with static timing analysis or measurement-based techniques [180], is independent of the accesses that other co-running tasks may do on the same hardware shared resources. This effectively enables the computation of WCET estimates for each task in isolation, bringing the benefits of reducing time validation and verification costs as explained in the previous section. Unfortunately, at the time AHB was released, time composability was not one of its design goals. As a result, AHB-compliant master/slaves lead to non time-composable behavior that makes difficult analyzing the timing properties of an AHB-based MPSoC.

In this section we review AHB features and classify them according to their effect on timing. Table 4.1 summarizes the features we analyze.

1) *Number of masters*: AHB allows up to 16 bus masters contending in one bus. The number of masters does not break time composability because it is a feature known at design time, like the number of cores in a multicore platform. However, it affects the tightness of WCET estimates. In real-time scenarios, the worst-case situation to consider happens when, on the event of a master trying to get access to the bus, all the other masters want to access the bus the same cycle, all of them having higher priority.

2) *Handover*: Changing the ownership of the bus from one master to another is called handover. AHB has a one-cycle master handover, so a master being granted access to the bus in a given cycle, gets the bus in the next cycle, so the minimum arbitration time is one cycle. This effect can be taken into account for each request of a task by adding this extra cycle of delay to the arbitration time. This feature is time-composable and has negligible effect on timing tightness.

3) *Back-to-back execution*: Back-to-back execution occurs when two transfers from different tasks are executed consecutively one after the other, without any idle cycle in between. This is feasible because of the pipelined execution of AHB, where address and data phases of different transfers overlap. This feature has

Table 4.1: List of AHB features analyzed

ID	Feature	Breaks TC?	Affects WCET bounds?
1	Number of Masters	No	Yes
2	Handover	No	No
3	Back-to-back	No	No
4	Burst operation	Yes	Yes
5	Flow control	Yes	Yes
6	Split transactions	No	Yes
7	Locked transfers	Yes	Yes
8	Bus width	No	Yes
9	Protection control	No	No
10	Error response	No	No
11	Retry response	No	No
12	Idle transfers	No	No
13	Early burst termination	No	No
14	Arbiter	Yes	Yes

neither an effect on timing nor on time composability.

4) *Burst operation*: Burst operation allows to perform transactions composed of more than one beat. AHB allows to have undefined-length bursts, 4-beat, 8-beat or 16-beat burst transactions. Undefined-length burst can be of any length, although its limit is constrained by the fact that the address cannot cross a 1 KB boundary. The burst length affects WCET estimates since, to compute them, it is needed to assume always the maximum burst length allowed in the system, even though it might be too pessimistic to be useful.

5) *Flow control*: Flow control in AMBA AHB can be performed by both the master and the slave. The slave can extend the data phase of a transfer by inserting *wait states* if, for instance, it needs extra time to process the transaction. Similarly, the master can insert *busy* transfers with the same purpose of extending the time between data transfers.

The original AHB specification states that the number of wait states is limited, but it does not set any specific limit. The absence of a specific limit breaks time composability because it eliminates the possibility of computing how long a transaction can take, so how many other tasks in the system can be delayed because of such transaction. Further note that both, wait states and busy transfers, increase WCET estimates since for every transaction, the maximum number of wait states and busy transfers must be assumed.

6) *Split transactions*: Split transactions provide a mechanism for slaves to release the bus when they need some more time to respond. This allows other

masters to get access to the bus rather than waiting for the slave to finish. When a slave signals a split transaction, the arbiter masks the request of its corresponding master until the slave indicates that the response is ready, so the master is considered again in the arbitration. Note that *the master has to wait again for arbitration*, and only when the arbiter grants the bus access, the slave can respond to the request. Signaling a split transaction needs a two-cycle response. This effectively adds two cycles to the size of a transfer for WCET estimates.

Therefore, a split transaction only affects the timing behavior of the master that received it, as two arbitration phases occur: the first one when the request is issued, and the second one when the slave is ready to respond. This is not the case for the rest of masters, in which a split transaction is seen as two independent transactions. It is important to remark that, although the master can keep the slave component busy during a split transaction, thus affecting others, this contention is not because of the bus, but because of the slave.

7) *Locked transfers*: Locked transfers allow a master to keep the ownership of the bus until the locked sequence has been completed. Locked transfers ensure that a transfer is done without disturbance, which is necessary, for instance, in the case of read-modify-write requests to an AHB connected cache or memory device. However, locked transfers can be very harmful for time composability, because a master can issue a locked transfer and keep the bus busy indefinitely, thus affecting other masters. Section 4.4.1 provides a discussion of how the functionality provided by locked transfers can be implemented avoiding the issue of affecting the other tasks in a timing unpredictable way.

8) *Bus width*: AMBA AHB allows different bus widths: 8, 16, 32, 64, 128, 256, 512 and 1024 bits-wide buses. It is recommended a minimum of 32 bits. This feature does not affect time composability but it has an effect on timing, because the wider the bus, the fewer the number of transfers needed. Note that the bus width is known at design time, like the number of masters.

9) *Protection control*: AMBA AHB has protection control signals that provide information about a bus access, to be used by any module that implements some level of protection. These signals indicate whether the access is instruction or data access; user or privileged access; bufferable and cacheable. Since no functionality is attached to these signals, because it is something optional and for higher level protocols, it is completely harmless for time composability.

10) *Error response*: A slave can respond to a transfer with an *error*, to indicate that something went wrong. Although errors may break timing behavior of a task, it is a responsibility of the master-slave higher level communication protocols to take care of errors and it does not affect other masters. Hence, it does not affect time composability. Errors require two extra cycles for signaling, as in the case of *split transactions*.

11) *Retry response*: A slave can respond also with a *retry* response, to indicate the master to perform the transfer again. Like *error responses*, retries may also break timing behavior of a task, but this is again responsibility of the master-slave higher level communication protocols and does not affect other masters. Hence, it does not affect time composability. Retries also require two extra cycles for signaling, as in the case of *error responses* and *split transactions. Retry responses* can also be used as an alternative to *split transactions* when the slave is unable to provide the response (e.g., due to its high latency). The difference is that with *retry response*, the normal arbitration priority scheme will be maintained. With *split transactions*, the arbiter masks the request of the split master until the slave indicates that the response is ready, which improves performance, since that master will not be granted access unless the response is ready.

12) *Idle transfers*: Idle transfers are used to indicate that no data transfer is required. AMBA uses a default master when all other masters are unable to use the bus. When granted, the default master must only perform *idle transfers. Idle transfers* can also be used in case a master cannot continue a burst. It is completely harmless for time-composability.

13) *Early burst termination*: Early burst termination is a mechanism that allows slaves to detect when a burst transfer is incomplete. If during a burst transfer, a slave detects an idle transfer or a non-sequential transfer, it means that the previous transfer finished before it was completed. This may occur if the arbiter changes the ownership of the bus, or if the master cannot finish the burst. This feature is completely harmless for time composability.

14) *Arbiter*: AMBA does not define any restrictions on the arbiter, which means that any arbitration policy can be used. This can completely break time composability as we show in the example in Section 4.3.1. The arbitration policy affects the timing behavior of all masters, because it defines how much time a master has to wait to be granted access to the bus, which indirectly depends on the other masters' behavior.

4.3.1 Example of Non Time-Composable Behavior

Unfortunately, at the time AHB was released, time composability was not one of its design goals. As a result, AHB-compliant master/slaves lead to non time-composable behavior that makes difficult analyzing the timing properties of an AHB-based MPSoC. Figure 4.3 shows an example that illustrates three AHB-compliant non-time composable behaviors of a master/slave communication. Concretely, it shows the timing diagram of an AHB bus of two masters, $M1$ and $M2$, each sending transactions A and B respectively, both with a burst transfer size of 4 beats. The transfer part of the transaction from master $M1$ occurs between cycles $n + 2$ and $n + 6$. In those transfers, phases overlap across beats. For instance,

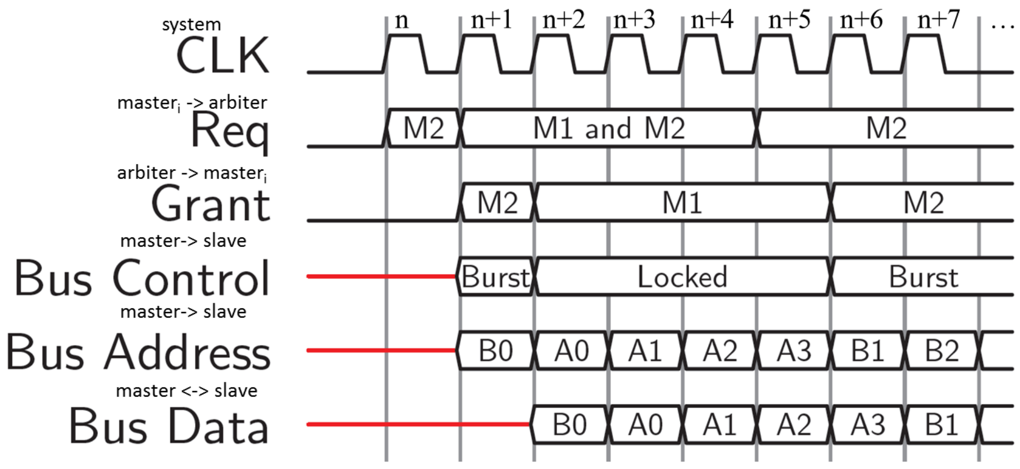


Figure 4.3: Example of non time-composable behavior.

address phase of transfer $A1$ overlaps with the data phase of transfer $A0$ given that they use separate buses. Non time-composable behaviors are as follows:

a) $M1$ requests the bus in cycle $n + 1$ and gets it in cycle $n + 2$, thus having an arbitration time of 1 cycle. Master $M1$ issues a locked transaction at cycle $n + 2$. As defined in the AMBA specification the arbiter cannot relinquish the bus during a locked transaction, so that this transaction from $M1$ can affect the timing of other masters in an unpredictable manner potentially preventing them from using the bus indefinitely (4 cycles in the example). Hence, the existence of locked transactions is a feature that breaks time composability.

b) Masters can issue unspecified-length burst transfers, which does not allow bounding the latency of a transaction. For instance, in Figure 4.3, $M1$ issues a burst length of 4, which means that for WCET estimation we have to assume that every possible transaction from $M1$ is at least 4 beats long. However, if a new master is connected with a burst-length of 8, it invalidates the previous analysis.

c) Finally and more revealing, AHB specification does not put any requirements on the arbiter behavior, and only the arbiter interface is specified. For instance, the arbiter can take away the bus grant from a master, as occurs in cycle $n + 2$ when $M2$ is sending a burst transfer and the arbiter changes the bus ownership to $M1$. This can happen due to the fact that any arbitration policy to select the next master to grant access to the bus can be used. Under some arbitration policies, there may not be an upper bound on the time one master can delay the others to access the bus, thus, breaking time composability. For instance, if $M1$ has higher priority than $M2$ and both tasks attempt to access the bus simultaneously, $M2$ is stalled until $M1$ finishes. However, if before the first request from $M1$ finishes, another request from $M1$ becomes ready, $M2$ will also wait for the second $M1$

request to finish as well. Thus, the bus contention that $M2$ suffers depends on $M1$. Even though this effect can have an upper bound and be computed knowing $M1$, it breaks time composability, because if we change $M1$ behavior (e.g., the task running on top of such master) it invalidates the timing analysis for $M2$.

In summary, in order to achieve time composability several features of AMBA AHB must be either limited or disabled. In particular, unrestricted locked transfers must be avoided, only time-composable arbitration policies must be considered, and burst length, wait states and busy cycles must be limited.

4.4 Restricted Time Composable AHB

As shown in previous section, AMBA AHB specification does not guarantee time composability at bus transaction level. This section defines a restricted usage of a subset of the AHB features, such that if every master and slave follow this restricted AHB specification (*resAHB*), time composability can be *guaranteed by construction* at transaction level. The main goal of *resAHB* is to derive tight upper bounds for every bus transaction, regardless of the actual behavior of other master/slaves components, while keeping AMBA AHB functionality.

In order to achieve time composability, the AHB features that must be considered are burst length, wait states and busy cycles (i.e. flow control), locked transfers and the arbitration policy.

Burst length: AHB allows defining burst sizes of 4-beat, 8-beat, 16-beat and undefined length. Undefined burst sizes break time composability as they prevent bounding the impact of a bus transaction on other transactions belonging to different tasks. Therefore, we enforce maximum burst length to be 16 beats, which will be compatible with existing AHB components. Components needing bus transactions larger than 16 beats must split the transaction in several transactions. It is important to remark that the burst sizes of undefined length can still be set assuming any length smaller than 16 beats. If the burst length of an undefined transaction is above 16 beats, the corresponding master or slave will have to split the transaction.

Flow control: Flow control defines the number of *wait states* and *busy transfers*. As we have seen in the previous section, the original AHB specification states that the number of wait states has to be limited, recommending not to insert more than 16 wait states but it does not specify any limit. In order to provide time-composable AHB transactions, we restrict the number of wait states to 16, which is the maximum recommended by the specification and corresponds to 16 bus cycles. In case a slave needs more than 16 wait states, it can signal either a *split transaction* or a *retry* (in case the resource is busy).

On the master side, busy transfers need also to be limited. Similar to wait states, we enforce masters not to introduce more than 16 busy transfers, which corresponds to 16 bus cycles. If a master needs more than 16 busy transfers, e.g. it cannot continue the current transfer, it will have to signal an *early burst termination* which releases the bus. The transaction can then be performed once the master is ready.

It is worth noting that both, wait states and busy transfers, introduce pessimism in WCET estimates as the worst-case scenario must be considered, i.e. every transfer incurs 16 wait states and busy cycles.

Arbiter. The AMBA AHB specification does not put any restriction on the arbitration; this is not suitable for time composability, so *resAHB* constraints the timing behavior of the selected arbiter.

On the one hand, *resAHB* restricts the arbitration policy to those that allow to derive upper bounds on the time a master needs to wait to be granted to use the bus, e.g. TDMA (*Time Division Multiple Access*) and Round-robin [138]. For example, under round-robin arbitration policy, in the worst case a master waits $N - 1$ rounds of arbitration before it gets the bus, where N is the number of masters. Hence, the longest arbitration latency a master suffers due to inter-task interferences is bounded by $Bound_{RR} = (N-1) \times (t_{tran} - 1)$, where t_{tran} is the bus transfer time from which one cycle is subtracted because in consecutive transfers address and data phases overlap. Any other arbitration policy that allows to derive time composable upper bounds on the arbitration time can be considered. We elaborate more on this topic in Chapter 5.

On the other hand, the arbiter must not change bus ownership during a transaction. That is, under *resAHB* a master cannot be preempted once it has been granted access to the bus.

Locked transfers. Under *resAHB*, locked transfers are not allowed to take longer than the maximum transaction latency, that is 50 cycles as shown in Section 4.4.2. Locked transfers are used to eliminate disturbance when doing an access or a sequence of accesses. If the accesses can be served with a single transfer, locked transfers are not needed, since, as pointed above, bus transactions are not preempted, i.e., other masters cannot use the bus until the current transaction finishes. In the case of a sequence of accesses that require more than one transfer, e.g. read-modify-write accesses, the atomicity provided for individual transfers does not suffice to provide the same functionality as *locked transfers*. In Section 4.4.1 we present a mechanism to provide this functionality without requiring locks.

4.4.1 Providing the same functionality as AHB

resAHB restricts the use of locked transfers to avoid masters to lock indefinitely the bus, which would simply kill time composability. However, locked transfers provide a functionality that may be required by masters. For instance, cores (masters) may want to perform an atomic operation on a shared memory to enable the use of higher level locking protocols, like the priority ceiling protocol [145], for which masters use locked transfers. In order to be able to maintain this functionality, we propose two different approaches.

The first approach benefits from the fact that under *resAHB* the arbiter cannot relinquish the access to a master once it is granted access. Hence, if the sequence of accesses requiring an atomic operation (e.g *rmw*) fits within the maximum possible transaction length (i.e. 50 cycles) it will be carried out correctly. This is normally the case, since (1) at software level locking time is reduced as much as possible to improve performance and (2) locked transfers are usually issued to cache memories which have short latency. Note that the first time the master asks for a data with an 'atomic' request (which can be identified using the *HMASTLOCK* signal), the slave has to fetch it, which can take longer than 50 cycles. In that case the slave simply responds with a split or retry response and starts fetching the data. Once the datum is cached, the atomic operation can be carried out in less than 50 cycles.

The second approach consists in moving some functionality to the slave component (e.g., a cache). For instance, in the case of the SPARCv9 architecture, *rmw* operations can be implemented with the *compare-and-swap* (CAS) operation that works as follows:

```
int64 CAS (int64 *word, int64 test_value, int64 new_value )
{
        int64 old_value;

        atomic {
            old_value    = *word;
            if ( *word == test_value )
                *word= new_value;
        }

        return( old_value );
}
```

The core (master) sends a read operation of address $*word$ to the cache. While handling this request, the cache must prevent any other request from accessing $*word$. Once the core receives the answer, i.e. the data in $*word$, it relinquishes the bus. Then the core compares the content of $*word$ with the *test value* provided in the CAS operation. In case of match, the core starts a new transfer on the bus to write the *new value* to $*word$. Otherwise, if there is no match, the core starts a new transfer to write the *old value* to $*word$. The cache will not accept new accesses to $*word$ until the write operation is served. Note that the AHB signal *HMASTLOCK* indicates the slave that the transfer is a locked transfer, allowing the slave to be aware of locked transfers.

In any case, the only restriction is that any locked transaction cannot exceed the maximum transaction length.

4.4.2 Deriving bounds to access latency

The *resAHB* specification allows deriving upper bounds on the time an AHB bus transaction takes. The upper bound is defined as $\tau = t_{arb} + t_{tran}$, where t_{arb} is the longest time it takes a master to be granted access to the bus since the cycle in which it requests access to the bus. t_{tran} is the transfer time, the longest time a master is entitled to use the bus once it is granted access by the arbiter.

t_{arb} depends on the arbitration policy, the number of masters and the maximum time each master can use the bus once it is granted access, i.e., t_{tran}. For instance, for round-robin this time is $t_{arb} = 1 + (N - 1) \times (t_{tran} - 1)$, which corresponds to the bound of round-robin plus one extra cycle needed for the initial *handover* [3]. t_{tran} is given by the longest possible transfer that corresponds to a 16-beat burst transfer with 16 wait states and 16 busy transfers. It also includes two extra cycles for signaling an error/retry/split response. This totals 50 bus cycles*, regardless of the specific hardware and software details of master and slave components*. Note that t_{tran} is decreased by 1 cycle since data and address phases across transactions overlap in 1 cycle.

For instance, in a 4-master bus with round-robin policy, the maximum, time-composable arbitration time is $t_{arb} = 1+(4-1)\times(50-1) = 148$. By computing the WCET estimate for a task assuming this arbitration time, will make its WCET estimate independent of the other tasks it runs with. This occurs because the worst effect that other tasks can cause on the bus is already taken into account by the maximum arbitration time.

4.5 Advanced High-performance Real-time Bus

AMBA AHB can be made time composable by construction by restricting the use of some AHB features. Though this is an attractive proposition since it keeps compatibility with AMBA, it leads to pessimistic WCET estimates. For instance, we have seen that with 4 masters and round-robin arbitration policy, the arbitration time that a given request has to assume in order to be time composable is as big as 148 cycles.

In this section we extend the AHB specification in the form of extra features that improve AHB time composability properties. Obviously these extended features make the new specification non AMBA-compliant. We call our AHB specification extension AHRB *Advanced High-performance Real-time Bus.*

[3]The worst-case corresponds to every master accessing the bus at the same cycle which requires an initial cycle for handover. The rest of handovers are not consuming any extra cycle because of the back-to-back execution of requests.

4.5.1 Master and Slave modes

The focus of AHRB is on reducing τ which in turn requires reducing t_{arb}, the arbitration time, in order to produce tighter WCET estimates. t_{arb} directly depends on t_{tran} which is the maximum allowed transfer length, which was 50 cycles under *resAHB*. t_{tran} depends on the duration of (1) the burst length, (2) the wait states and (3) the busy transfers that are allowed to be inserted. Busy transfers and burst length are introduced by the master and depend on the amount of data to transfer (e.g., a cache line) and the master internal behavior; Meanwhile, wait states are introduced by the slave and are used to cover the latencies of the slave components.

If t_{tran} could be determined taking into account the specific timing requirements of the particular masters and slaves using the bus in terms of their burst lengths, wait and busy states, the value of t_{tran} would certainly reduce with respect to the upper bound value used under *resAHB*. For instance, let us assume a 4-core processor with a shared L2 cache, 16-byte cache lines and a 128-bit wide bus, so that each transfer needs 1-beat to send a whole cache line. L2 cache hit latency is 4 cycles. In this scenario, by defining a maximum transfer length of 1-beat plus 4 *wait states* to allow the cache to serve the access, we cover all L2 hit accesses with one transaction with the minimum required length. The maximum transfer time will be $t_{tran} = 1 + 4 + 1 + 1 = 7$ cycles[4], and the maximum arbitration time under round-robin policy $t_{arb} = 1 + (4 - 1) \times (7 - 1) = 19$ cycles. These values are less pessimistic than the ones we would obtain with *resAHB*, i.e, $t_{arb} = 148$, which significantly improves worst-case performance. In case of a L2 miss, the L2 cache signals a *split transaction*, spends all the required latency to bring the data and pays another arbitration penalty whenever the response is ready. However, if the access to main memory takes in the order of dozens or even hundreds of cycles, the relative overhead of two arbitration rounds is low. Moreover, the benefit of issuing a *split transaction* for L2 misses is that the bus is free while the miss is being resolved, enabling other masters to use it. Conversely, putting a lower limit on the number of *wait states*, for example only 2, would make that any hit in the L2 signals a split transaction, since the hit latency is at least 4 cycles. This would mean that any L2 hit access would have to perform 2 arbitrations, the regular one and an extra one when the L2 has the data ready to be sent, which would deteriorate performance, because hits are frequent. In Section 4.6.4 we show this behavior with experimental results.

Overall, adapting t_{tran} to the specific needs of the masters and slaves reduces the overhead required to achieve time composability. The challenge lies on defining in the bus specification a mechanism that allows masters and slaves specifying their

[4] 1 cycle to send the address, 4 cycles to read the data from L2, 1 cycle to transfer the data, and 1 cycle in case the slave asks for retry or signals an error.

needs in the use of the bus. To that end, we introduce the concept of *master and slave modes*, which we will call simply *modes*. In the AHRB specification we define modes from 1 to 32, according to the number of busy transfers and burst beats in the case of the master (up to 16 each), and the number of wait states in the case of the slave (up to 16).

The arbiter assigns a master and slave mode when granting the bus to a master that forces the master and the slaves to operate in that mode. Under each master and slave mode of operation, we can easily derive tight bounds for t_{tran} and hence t_{arb}. For instance, a master i with *master mode 4* ($mm(i) = 4$) will insert at most n-beat bursts and m busy transfers where $n+m \leq 4$, and slaves with *slave mode 4* ($sm(i) = 4$) will insert at most 4 wait states, plus the two extra cycles for signaling an error/retry/split response:

$$t_{tran}(i) = mm(i) + sm(i) + 2 = 4 + 4 + 2 = 10 \tag{4.1}$$

Under round-robin deriving the maximum arbitration time for a master i, $t_{arb}(i)$, is straightforward. $t_{arb}(i)$ is the addition of $t_{tran}(k)$ for every master $k \neq i$, i.e., with round-robin policy:

$$\begin{aligned} t_{arb}(i) &= 1 + \sum_{\substack{k=0 \\ k \neq i}}^{N-1} (t_{tran}(k) - 1) \\ &= 1 + \sum_{\substack{k=0 \\ k \neq i}}^{N-1} (mm(k) + sm(k) + 1) \end{aligned} \tag{4.2}$$

4.5.2 Deriving tight WCET estimates

For each task a WCET estimate is computed under (1) each of the modes of the master and slaves that the task uses and (2) each of the modes the other masters/slaves may have. For a task k running on master i, under each combination of (1) and (2) a different WCET estimate is obtained, $WCET_k^{t_{arb}(i),t_{tran}(i)}$.

For instance, Table 4.2 shows t_{tran} and t_{arb} for an example with 4 masters and a slave where different modes are assumed for each master and slave. In particular we use 2 modes for the masters, 1 and 4, and modes 2 and 4 for the slave. t_{tran} and t_{arb} are computed with Equations 4.1 and 4.2 respectively. Obviously, not all combinations of all potential master and slave modes have to be considered, but only those combinations with the lowest values that still upper bound the most common requests.

At integration time, the scheduler selects the mode that each task will have for each master and slave. Task k does not violate its deadline for any master i such that:

Table 4.2: $t_{tran}(0)$-$t_{tran}(1)$-$t_{tran}(2)$-$t_{tran}(3)/t_{arb}(0)$ under different master and slave modes in a 4-master configuration. $t_{arb}(0)$ is for master 0 and $t_{tran}(i)$ for master i.

	Master modes			
slave mode	1-1-1-1	1-1-1-4	1-1-4-4	1-4-4-4
2	5-5-5-5/13	5-5-5-8/16	5-5-8-8/19	5-8-8-8/22
4	7-7-7-7/19	7-7-7-10/22	7-7-10-10/25	7-10-10-10/28

$$WCET_k^{t_{arb}(i),t_{tran}(i)} \leq d_k \tag{4.3}$$

where d_k is the deadline of task k. Tasks are not allowed to change the mode of any master or slave. Only the Operating System scheduler is allowed to change those modes, for which we envisage a privileged instruction such as writing to a special purpose register. This prevents, tasks with different criticality levels affecting each other timing behavior beyond the setup set by the scheduler.

Task scheduler and the master/slave modes, an example

The following synthetic example details how the scheduler interacts with the arbiter and the master/slave modes. For simplicity we focus on the case in which each master can be used by only one task. In our example we assume three tasks (t_0, t_1 and t_2) scheduled in three different cores (masters) and using one or more slaves. Each master can operate under mode 1 or 4, and all slaves operate always under mode 2. The deadline and period for all tasks is 2ms. Table 4.3 shows the WCET estimate in milliseconds for each task under each master/slave mode combination. For instance, *114-2* corresponds to the case where t_0 and t_1 operate in master mode 1, t_2 in master mode 4 and the slave in mode 2.

In the example, t_0 and t_1 send short transfers, so master mode 1 provides them good performance. Conversely, t_2 achieves lower WCET estimates under master mode 4 since its transfers are longer and master mode 1 produces split transactions. Therefore, the lowest WCET estimates for t_0 and t_1 occur under 111-2 modes, and for t_2 under 114-2 modes.

Different approaches can be followed to choose the proper modes for each task. However, in all cases 441-2 and 444-2 are not eligible because t_2 would violate its deadline (2ms). Out of the other combinations of modes, 114-2 would be the one minimizing the WCET estimates for all tasks, $1.4+1.4+1.3 = 4.1$, hence reducing the CPU capacity used by those tasks, which can be left to other tasks. 111-2 modes minimize the CPU capacity required by t_0 and t_1.

Once a particular combination of modes is selected (e.g., 114-2), the arbiter is in charge of properly configuring masters and slaves on each arbitration every

Table 4.3: WCET (in milliseconds) for three tasks under several mm/sm

Tasks	mm/sm modes							
	111-2	114-2	141-2	411-2	144-2	414-2	441-2	444-2
t_0	1.2	1.4	1.4	1.4	1.6	1.6	1.6	1.8
t_1	1.3	1.4	1.4	1.4	1.5	1.5	1.5	1.7
t_2	1.7	1.3	1.9	1.9	1.5	1.5	2.3	2.1

time it grants access to a master. For instance, if the master where t_0 runs is granted access, the arbiter sets it to master mode 1 and the slaves as slave mode 2. Similarly, when the arbiter grants access to t_2 master, it enforces such master to operate in mode 4.

In the general case, unlike in our example above where each task is bound to a master, masters and slaves can be used by several tasks. However, the same principles drawn in this example rule the allocation of modes to task master and slaves.

4.5.3 Providing the same functionality as AHB

As shown in Section 4.4.1, only the functionality provided by locked transactions is affected when using the restricted version of AHB. In *AHRB*, we keep the same principle to deal with locked transactions, however, the timing restrictions now depend on the master and slave modes. This means that any locked transaction has to fit these restrictions, i.e., a master contribution to a transfer must never exceed its operation mode limit (the same applies for slaves). The best solution to provide the same functionality as locked transactions, is to implement the functionality of *atomic operations* in the slave, as shown in Section 4.4.1. This occurs because in order to provide tighter WCET estimates, *AHRB* may have more severe timing restrictions that make more difficult to perform locked transactions within the allowed maximum transaction time.

4.5.4 AHRB architecture

In addition to the AHB architecture shown in Figure 4.1, *AHRB* needs some extra hardware support. Every master and slave requires knowing the mode it is allowed to operate. The arbiter keeps the information about the mode of every master and the slave mode associated to each master. This information can be changed by software (in supervisor mode), in order to allow the scheduler to change master and slaves modes so it can use different WCET estimates for any given task. This can be done through special purpose registers, which is a common mechanism in

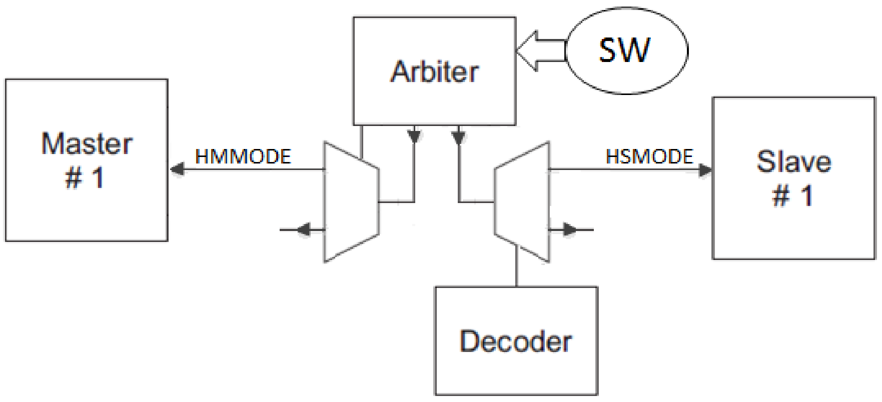


Figure 4.4: AHRB extended architecture with HMMODE and HSMODE signals.

current processor architectures.

We introduce two new signals that define the master mode, *HMMODE*, and the slave mode, *HSMODE*. These signals are generated in the arbiter and reach the master or slave component respectively. The extended bus architecture is shown in Figure 4.4.

4.5.5 AHRB principles in other AMBA specifications

AXI is a specification in AMBA3 which only defines interfaces between (1) the master and the slave, (2) the master and the interconnect and (3) the slave and the interconnect, allowing the chip designer to use potentially any interconnect. On the one hand, in order to determine whether time-composable access delays for the interconnect can be achieved, a specific interconnect has to be defined and analyzed. For instance, crossbars are time-composable by design. However, the fact that AXI does not define the timing aspects in the master-slave-interconnect communication provides full freedom to define an AXI-compliant time-composable interconnect (*tcAXI*).

In our view, this offers an excellent opportunity to real-time industry to define a *tcAXI* enjoying a well-defined and standardized interface such as AXI, while adding specific restrictions to make it time-composable. The principles that rule the definition of *tcAXI* should be in line with those we define in this chapter.

Besides AHB and AXI, the AMBA specification defines other interfaces that we qualitatively analyze next in terms of time composability.

AHBLite/APB. APB and AHBLite are intended for a single master, which prevents inter-task interferences and hence the need of time-composable access delays.

ASB. ASB was designed to be the main system bus, like AHB. However, AMBA recommends AHB for all new designs because AHB provides improved features. Anyway, the same approach explained in this chapter can be carried out in ASB.

ACE. ACE adds system-level coherence support to AXI, which does not have any impact on timing.

ATB. ATB is a trace bus for on-chip debug, which is free of real-time constraints.

4.6 Evaluation

In this section we quantitatively compare the time composable AHB proposals, *resAHB* and *AHRB* against the original AMBA AHB in terms of time composability, WCET estimates and average performance.

Simulator Setup. We use the simulator, explained in Section 3.3, to model a multicore architecture with 4 cores connected through a bus to the L2 cache and a I/O controller.

Each core is a pipelined processor core comprising fetch, decode, execute and commit stages. Each core has its own private instruction (IL1) and data (DL1) caches, which is common in current high-performance and real-time embedded processor designs [42, 84]. 16KB 4-way 16-byte-line IL1 and DL1 caches have been considered. The shared second level (L2) cache is 256KB with 8 banks, 8 ways and 16-byte lines. IL1 and DL1 hit and miss latencies are 1 and 2 cycles respectively. L2 hit and miss latencies are 4 and 6 cycles respectively. DL1 is write-through and L2 write-back. All caches use LRU replacement policy. The bus connecting the cores to the L2 and the I/O device is 128-bit wide, which means that a cache line can be transferred in a single-beat transfer and no burst transfers are needed. The I/O controller has 20 cycle latency.

The L2 cache deploys a cache partitioning technique, way partitioning, that deals with inter-task interferences [138] and has been implemented in real chips like the NGMP [42] or the ARM Cortex A9 [15]. For the memory controller we use the low-overhead solution proposed in [139], which upper bounds the effect of inter-task interferences on the requests of a core to the memory controller. Overall, the only source of inter-task interferences that jeopardizes time-composability, potentially affecting WCET estimation, is the AMBA AHB bus.

Benchmarks. We use the EEMBC Autobench suite [143] as reference programs (see Section 3.2.1). In particular we use: *a2time*, *aifftr*, *aifirf*, *aiifft*, *basefp*, *cacheb*, *canrdr*, *idctrn*, *iirflt*, *matrix*, *pntrch*, *puwmod*, *rspeed*, *tblook* and *ttsprk*.

We also develop a set of synthetic kernels that carry out a fixed number of accesses to the bus, either to the L2 or to the I/O device. We vary the percentage of accesses of the benchmark to the L2 and to the I/O device, such that, if a kernel makes $X\%$ of access to the I/O, the remaining $100\% - X\%$ of the accesses go to the L2.

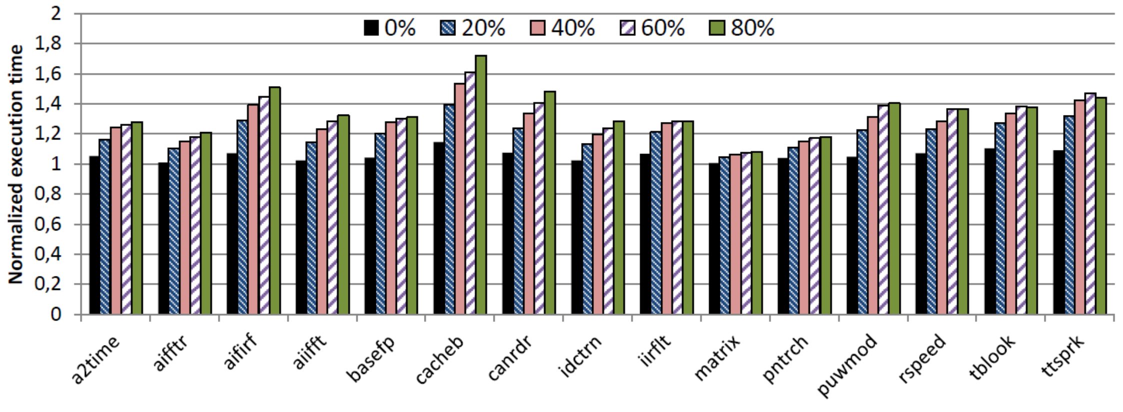
4.6.1 Achieving Time Composability

As MPSoCs integrate an increasing number of IP blocks coming from different suppliers and those IPs may be subject to different safety and security levels, enabling *mixed-criticality functionalities* to be run on the same MPSoC requires limiting the impact that one IP component can create on the timing behavior of the others. Leaving this to the IP provider, especially to those subject to less-restrictive criticality levels, may jeopardize the whole system time composability property. AHRB is precisely designed to provide trustworthiness in terms of timing behavior by design (construction), so that any misbehaving IP component does not affect the time composability of the other components. Hence, we evaluate our proposal in a setup in which time composability can be broken, as it is shown to happen with AHB.

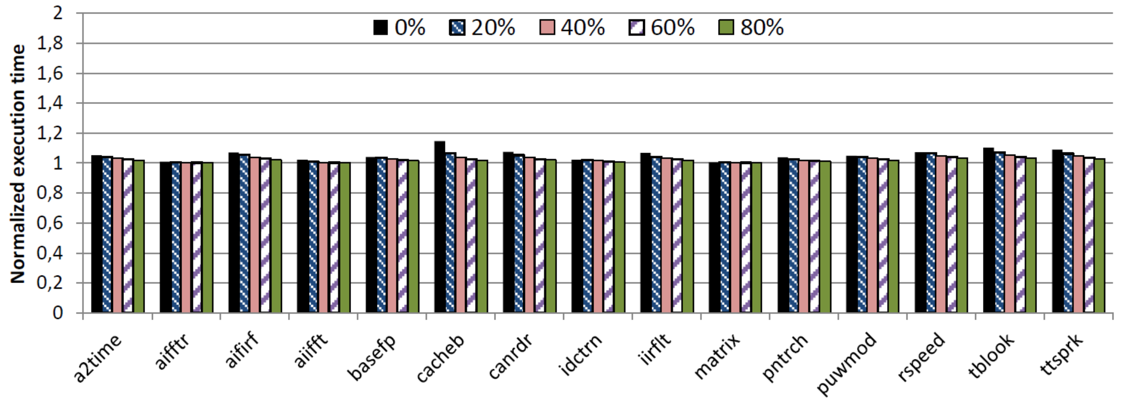
In our experimental setup, the I/O controller has 20-cycles latency and, under AHB, it benefits from unrestricted timing by keeping the bus busy for those 20 cycles when needed. In the case of the time composable *AHRB*, we configure the slaves in *slave mode 4* (L2 hit latency is 4) and masters in *master mode 1* (only 1 beat is required to transfer a cache line), so that all L2 hit requests are served with only one bus transaction. The I/O component has to relinquish the bus on each request because the bus is configured in slave mode 4, which means that a component can only insert 4 wait state cycles, which is less than the latency of the I/O component (20 cycles). Hence, the I/O component issues a split transaction on every access, and so every request goes through two arbitrations.

In order to understand the potential contention that a transaction can suffer in the bus, we run one EEMBC benchmark in one core while the other three cores run 3 copies of the synthetic kernel that continuously accesses memory and the I/O controller, so that they affect significantly the EEMBC benchmark execution time. Different fractions of memory and I/O accesses are considered for this synthetic kernel. Figure 4.5 shows the normalized execution time of each EEMBC benchmark when running against 3 copies of the synthetic kernel with respect to its execution time in isolation (running alone in the MPSoC). Results are shown for both the regular AHB and the time-composable *AHRB* scenarios. Note that the execution times in isolation are *the same* under both setups since cache hits are handled in one transaction while misses require two transactions using split transactions. The I/O device is not accessed by EEMBC benchmarks which only access the L2 cache.

Bars show the results as we increase the percentage of I/O requests sent by the synthetic kernel. Figure 4.5a shows the results for the standard AMBA AHB. As



(a) Normal AHB



(b) Time composable AHRB

Figure 4.5: Execution times of EEMBC benchmarks against 3 synthetic kernels that continuously access memory and the I/O controller for AHB and AHRB. The percentage indicates the amount of accesses to the I/O controller.

shown, bus contention significantly affects the execution time of EEMBC benchmarks (6% to 35% in average). This occurs because every I/O access keeps the bus busy for 20 cycles and as the percentage of I/O accesses grows, this saturates the bus, thus delaying L2 cache accesses for EEMBC benchmarks due to bus contention. Note that for *cacheb*, which makes the most intensive use of the bus across all EEMBC benchmarks, the impact of the synthetic kernels is high (up to 72%) when the percentage of I/O accesses grows. Eventually, a slave component could be in place with unspecified or arbitrarily long latency. In such case no WCET estimate could be provided for any program, or such estimate would be so high that it would be of no use.

Figure 4.5b shows the results for the time-composable *AHRB*. We observe that bus contention becomes negligible (1% to 5% in average), since I/O accesses become split transactions, thus letting L2 cache accesses of the EEMBC benchmarks to proceed. As the fraction of I/O accesses of the 3 copies of the synthetic kernel increases, their *bus access frequency decreases*. This occurs because for this kernel, the accesses to L2 frequently hit so they have shorter latency than I/O accesses. As a result, with low L2 access rate, and hence with high I/O access rate, the kernel access less frequently to the bus. Hence, in the presence of I/O intensive kernels, the EEMBC benchmark is granted access to the bus quickly and competes with fewer requests in memory in case of a L2 miss, thus executing faster. This effect is particularly noticeable for *cacheb* and *canrdr* that have slightly higher execution time when the kernels execute low percentage of I/O operations.

Our results show that the potential effect that tasks can suffer due to inter-task conflicts can be reduced using *AHRB*. *AHRB* also provides time-composable upper bounds by construction, without any requirements on the components at hardware or software level (apart from being compliant with the *AHRB* specification), which significantly simplifies WCET estimation. This allows using components that otherwise would degrade significantly WCET estimates or simply would not allow to obtain such estimates. The main limitation of the standard AMBA AHB is that WCET estimates depend on the actual behavior of components in place. If such timing behavior changes, for example due to a firmware update that changes the I/O component latency to 25 cycles, or simply other applications make a different use of the component triggering higher latencies, the timing analysis of all tasks running on the system is invalidated, even if those tasks do not ever use such component. By putting the timing restrictions on the bus interface instead of on the components, we avoid this issue because the timing upper bounds are set by the bus interface itself and not by the IP component.

4.6.2 Average performance

With time composable *AHRB*, I/O transactions are split into two transactions. This, on the one hand, increases the latency of each transaction as it has to pay two arbitration rounds. On the other hand, however, each arbitration round is much shorter since the bus cannot be locked for long time. Our results show that the benefits of shorter arbitration rounds largely offset the extra latency due to the fact that I/O transactions have to pay two arbitration rounds.

Figure 4.5 show that EEMBC benchmarks have higher average performance, i.e. shorter execution time, under *AHRB* than under AHB. To complement the average performance study, we measured the performance of the synthetic kernel under all percentages of I/O and L2 operations (i.e. 0%, 20%, 40%, 60% and 80%). Figure 4.6 shows the average throughput improvement of *AHRB* over AMBA AHB,

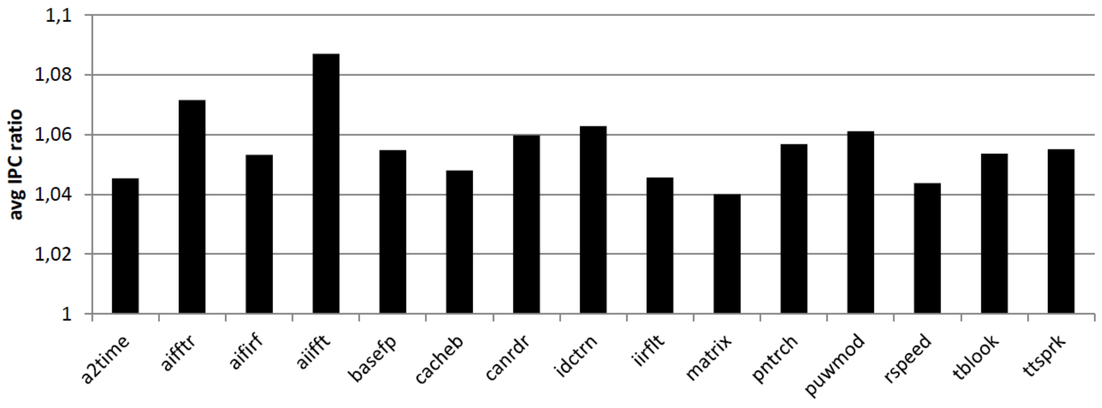


Figure 4.6: Average IPC improvement of *AHRB* over AHB, for the 3 copies of the synthetic kernel when running in a 4-workload setup with each EEMBC.

i.e., the average of IPC_{AHRB}/IPC_{AHB} (IPC stands for instructions per cycle) for all 3 kernels, when they run with each EEMBC under all I/O-L2 percentages. Results confirm the benefits of short arbitration periods over several arbitration phases, hence making AHRB to provide higher average performance than AMBA AHB. These improvements range from 4% to 9%.

4.6.3 WCET estimates

Regular AMBA AHB does not allow deriving time-composable WCET estimates. Instead, they can only be derived with *resAHB* or *AHRB*. For both, *resAHB* and *AHRB*, we have to assume that every access to the bus experiences the maximum arbitration latency [138] and also every memory access experiences the longest *request inter-task delay* [140] (i.e., the maximum possible delay due to requests from the other cores), since they are the only sources of *inter-task interferences* in our platform. For the bus, we take 148 bus cycles as the maximum arbitration latency for *resAHB* as explained in Section 4.4.2 and 19 for *AHRB*, applying Equation 4.2 with master modes being 1 and slave modes 4 (also shown in Table 4.2).

Figure 4.7 shows the normalized WCET estimates of each EEMBC benchmark with respect to its WCET when running in isolation for both *resAHB* and *AHRB*. It can be clearly seen that *AHRB* leads to much tighter WCET estimates than *resAHB*, in average 3.5x tighter. This occurs because the arbitration latency bound is high for *resAHB*, whereas such bound is much tighter for *AHRB* since we fit the bound to the master and slave characteristics with the *master and slave modes*. Although *resAHB* allows deriving WCET estimates, those estimates are so high (above 4x the ones in isolation) that it would be better to use just one core in the platform to schedule all tasks. Instead, *AHRB* provides WCET estimates largely

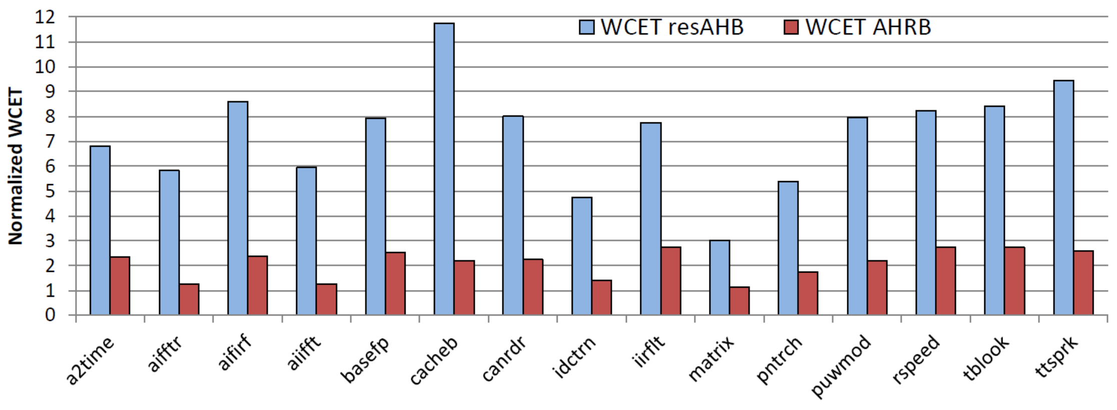


Figure 4.7: WCET estimates comparison for *resAHB* and *AHRB*.

below 4x, thus providing higher guaranteed performance when exploiting the 4 cores in the 4-core platform than when using just one core.

4.6.4 AHRB: Master & Slave modes

In order to show the effect of the master and slave modes on WCET, we compute WCET estimates for EEMBC under *AHRB* with different modes. In particular, we show results for slave modes 2 and 4. With slave mode 4 (equal to the L2 hit latency) we cover each hit request with one transaction. Instead for the slave mode 2, we need to issue split transactions and pay two arbitrations for every L2 hit. Hence, in slave mode 2, L2 accesses suffer lower bus contention due to the shorter transactions in the other cores, but suffer higher arbitration delay in case of a hit due to the split transaction. Therefore, depending on the L2 access frequency and L2 hit rate, the net effect in WCET estimates will vary.

The maximum arbitration time also depends on the other masters' mode. The lower the other masters' mode is, the smaller the arbitration time is. In this evaluation we assume that the master that runs the EEMBC is in master mode 1 and it runs with three other masters, either in mode 1 or mode 4, which correspond to masters that need only 1 transfer per access (e.g., cache line size matches bus width) and masters that need 4 transfers per access (e.g., cache line size is 4 times larger than bus width so a 4-beat burst is needed)[5]. In this case it is clear that the lower the addition of the transfer sizes of the other masters, the tighter the WCET estimate we will be able to derive. The possible combinations and the associated maximum arbitration time computed with Equation 4.2 are shown in Table 4.2.

Figure 4.8 shows the results of two slave modes (2 and 4) for the EEMBC

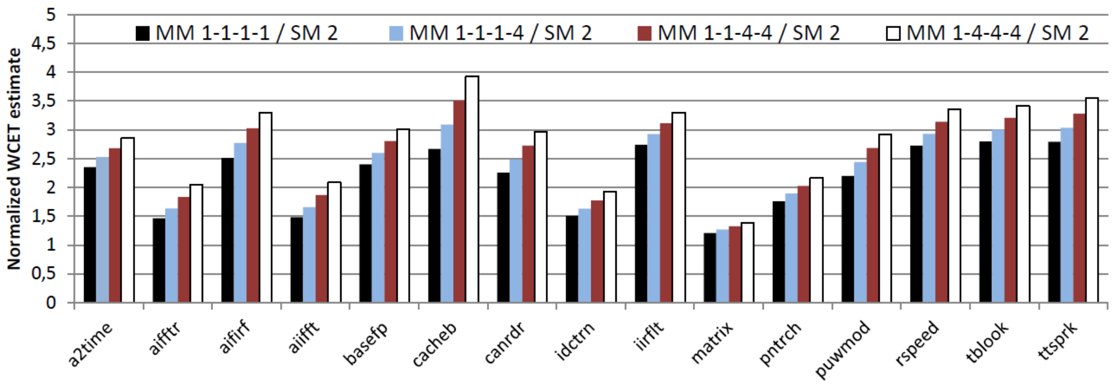
[5]Note that a high master mode might be useful to enable efficient (large) DMA transfers performed by a DMA master.

benchmarks, varying the master modes, shown as MM x-x-x-x in Figure 4.8. The first master, which is used by the EEMBC benchmark, is always in mode 1. We observe that WCET estimates depend on the rest of the masters modes. The higher the other master modes, the larger the WCET estimate is. We also observe that there are benchmarks that are barely affected by the modes, like *matrix*, because of the reduced use of the bus, and others are, however, significantly affected like *cacheb*. Finally, our results show that under *sm* 2, WCET estimates increase by 6% to 19% on average for the different master setups, because L2 hits, which are very frequent for many benchmarks, need 2 transactions to be completed, because the slave (cache) needs to perform split transactions. For instance, we observe that *cacheb* is severely affected when lowering the slave mode to 2 because it makes an intensive use of the L2 cache. For *matrix* the effect is negligible because it barely accesses L2 cache.

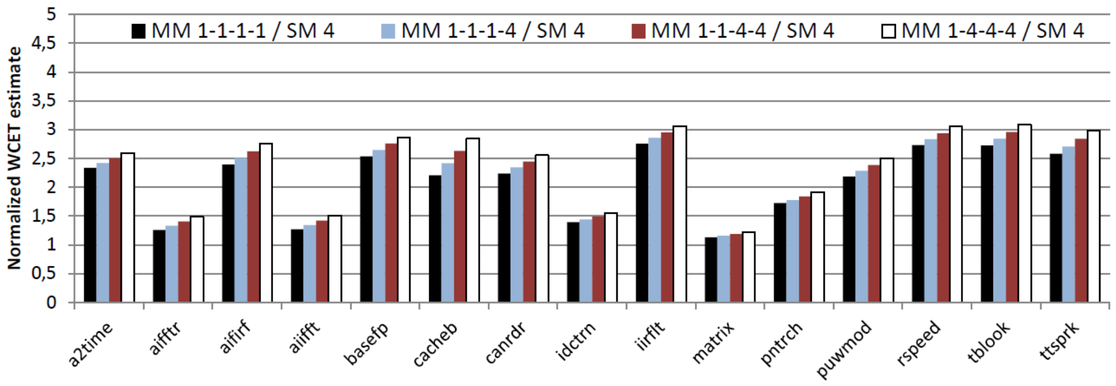
4.7 Summary

MPSoCs have become a must in critical real-time embedded systems (CRTES) since they deliver high performance needed for increasingly computational intensive applications. Integrating different IP components in those processors requires a standard specification for the communication bus, and AMBA AHB has been proven to be a suitable interface in embedded systems. Unfortunately, the need for incremental qualification poses requirements regarding time composability in the AMBA AHB.

In this chapter we thoroughly review AHB features identifying those that make AHB fail to provide time composability and those that, although time-composable, lead to non-tight WCET estimates. Then, we propose a time-composable AHB (*resAHB*) specification, which enables computing application's WCET, though with large overestimations. Finally, we introduce a new bus specification based on AHB, *AHRB*, which achieves both, time composability and tight WCET estimates. Our experiments show that WCET estimates can be derived on top of *resAHB* and *AHRB*, and *AHRB* improves WCET estimates by 3.5x w.r.t. *resAHB* with negligible average performance degradation over single-core execution.



(a) Slave Mode 2



(b) Slave Mode 4

Figure 4.8: WCET estimates for *AHRB* with different master modes (1 and 4) and slave modes (2 and 4).

Chapter 5

Bus Arbitration Policies

5.1 Introduction

The contention among several cores attempting to access the bus simultaneously, generates inter-task interferences in the bus that are handled by the arbitration policy. The choice of the policy affects the whole system because the time a bus request takes to be completed, depends on the amount of time that the request waits to be granted access by the arbitration policy, which intrinsically depends on the other running tasks (or system's workload). In order to be able to provide WCET estimates, the effect of inter-task interferences has to be known or, at least, upper bounded for every possible workload (ideally this bound is independent of the workload) and introduce as little pessimism in the WCET estimate as possible. As seen in Chapter 4, if that upper-bound depends on the set of running tasks, the WCET analysis has to consider all possible interactions within every workload, thus challenging time composability, which significantly complicates the analysis and integration of the system. Thus, using an appropriate arbitration policy will help (1) simplifying WCET analysis for multicore systems and (2) providing tight WCET estimates.

To our knowledge, two main policies have been proposed to deal with *inter-task interferences on on-chip buses* that satisfy these requirements: Time-Division Multiple Access (TDMA) and Interference-Aware Bus Arbiter (IABA) [138].

TDMA applies time sharing between the requests of the different contenders. Time is divided into *windows*, in each of which a contender is assigned a *slot*. When a request of a given contender becomes ready in that contender's slot, the request gets immediate access to the bus. If the request becomes ready out of the contender's slot, it waits until the contender's next slot. IABA, instead of time-sharing the access to the bus, allows tasks to contend for the bus using a given access policy such as round-robin, and bounds the delay that a request can suffer due to inter-task interferences under that access policy. This delay is taken into account when computing WCET estimates for each task, that covers the maximum effect that inter-task interferences can have on that task, thus providing reliable WCET estimates.

The main contributions of this chapter are the following: (1) We develop analytical models of the contention delay that each of those techniques introduce, which potentially affects the WCET of applications. (2) While both techniques have been widely analyzed in the past [151, 101, 138, 186], no study compares them. In this chapter we cover that gap by providing the first comparison between TDMA and IABA. Our study provides means to choose between TDMA and IABA in early stages of the hardware design, which is attractive to chip vendors given that buses are widely used nowadays in CRTES, and if both policies are available in the processor, our study guides software designers to choose the policy that better fits their needs.

For comparison purposes, we use several key metrics in the design of CRTES: WCET estimates using a commercial WCET analysis tool, time composability, average performance, prioritization capabilities and amount of changes required to the WCET analysis tool. Our results show that IABA represents a better choice for real-time multicores: IABA requires no changes in the WCET analysis tool, provides higher average performance and better WCET estimates in those scenarios in which the exact cycle when requests access the bus cannot be determined by the WCET analysis tool. TDMA, instead, slightly outperforms IABA only if the exact cycle in which each request accesses the bus can be determined, which is hard – if at all feasible – to obtain in general.

Given that several real processors deployed by real-time industries use buses as the main communication channel [84, 42, 54], we believe that our study provides valuable information for real-time system designers and chip vendors mainly in the early design stages to choose which policy better fits their needs (more details in Section 5.5).

The rest of this chapter is organized as follows: Section 5.2 describes both TDMA and IABA, which are later compared in Section 5.4. Section 5.5 describes how industry can use the results of this work. Finally, Section 5.6 presents the main conclusions of this study.

5.2 On-chip Bus Access Policies

When more than one hard real-time task run simultaneously on the processor, it may happen that two or more requests from different tasks attempt to access the bus at the same time. In this case, the arbitration policy decides which task is granted access to the bus and which one has to wait. Hence, the task that is granted access to the bus potentially delays other tasks. Under some arbitration policies, there may not be an upper bound to the time one task can delay the others to access the bus. For instance, let us assume two tasks, τ_1 and τ_2, so that τ_1 has higher priority than τ_2 and both tasks attempt to access the bus simultaneously. In this situation, τ_2 is stalled until τ_1 finishes. However, if before the first request from τ_1 finishes, another request from τ_1 becomes ready, τ_2 will also wait for the second request to finish as well. Thus, the upper bound that τ_2 suffers due to inter-task interferences depends on τ_1. Even though this upper bound can be computed knowing τ_1 sequence of accesses, it can be too long/pessimistic to be useful. Moreover, it breaks time composability, the property according to which the WCET estimate for a task τ_i can be computed in isolation and it is not affected by the other tasks that may run concurrently with τ_i (see Section 5.4.4).

The effect of inter-task interferences, which directly depends on the arbitration policy and indirectly on the set of running tasks (workload), has to be known or at least upper-bounded for every request in order to be able to provide safe WCET estimates. If the upper-bound of that effect depends on the other tasks running concurrently (as it is the case with the priority arbitration policy), time composability is broken (see Section 5.4.4). Hence, WCET analysis becomes significantly more complex – if at all attainable – because it must consider all the possible interactions within every possible workload.

TDMA and IABA arbitration policies, allow to upper-bound the effect of inter-task interferences regardless of the workload, for every request of a given task and provide tight WCET estimates. Both policies simplify the WCET analysis and allow using existing WCET analysis tools for single-core processors to analyze the timing of multicores.

We base our study in a multicore architecture in which each core has private instruction and data caches, which is the common practice in current high-performance and real-time embedded processor designs [42, 84]. A bus connects the cores to the shared L2 cache. Hence, a task (core) sends a request to the bus on (1) every L1 data cache load miss, (2) L1 instruction cache miss and (3) store operation since we model a write-through data cache. Memory operations that miss in L2 cache are sent to the memory controller that is connected to the L2 cache. When a core is ready to send a request to the bus at the beginning of a given cycle n, it sends a signal to the bus arbiter and if the core is granted access, it accesses the bus in the next cycle $n + 1$. Instead, if the core is not granted access, its request has to wait. The delay a request suffers to get access to the bus is called *Bus Inter-task Delay* (BID), being the total time (t_i) of a bus request i to reach the appropriate destination, $t_i = BID_i + r_i$, where r_i is the actual time that the request owns the bus, or request latency, which depends on the amount of data to transfer and the bus characteristics (e.g., width, latency). We will focus

	C0				C1				C2				C3			
cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BID	4	3	2	1	0	0	0	13	12	11	10	9	8	7	6	5
Case	A				B			C	D							

Figure 5.1: BID figures for a request of core 1 arriving in different cycles under the TDMA setup: $w = 16$, $N_c = 4$, $s = 4$ and $r = 2$

on BID, since r_i is independent of inter-task interferences, so we assume that r_i is known, or at least, upper-bounded, as it would be in the single-core case.

5.2.1 TDMA

TDMA has traditionally been used to arbitrate communications in distributed embedded systems (like the Time-Triggered Architecture [107]). Recently, several studies propose TDMA to be used for communications on MPSoCs. In particular in [151, 101] TDMA is used as the bus arbitration policy in a bus-connected multicore processor.

TDMA splits time into windows of size w cycles. Every window is divided into slots of size s and each contender to the bus (processor cores in our case) is assigned a slot. During a given slot, only its owner can send requests. If that contender has no requests to send, the bus will remain idle for that slot, even if there are requests from other contenders trying to access the bus (non-work-conserving). Slots can have fixed or variable length. Several approaches have been proposed to optimize TDMA slot size when TDMA is used in distributed systems [170] or to connect multiple cores [151] using variable slot length. In this chapter, TDMA uses equally-sized slots and the TDMA time window configuration chosen is used for all applications, not to affect systems time composability. We elaborate more on this point in Section 5.4.4.

Figure 5.1 shows a scenario with $N_c = 4$ cores, each having a TDMA slot of length $s = 4$ processor cycles and a window length $w = N_c \times s = 16$. A task on core 1 can only access the bus in the second slot ($n \in [4, 7]$). If the task sends a request when its slot has just elapsed, it has to wait for the rest of other core's slots to finish, i.e., $(N_c - 1) \times s$ cycles, even if there is no other request pending. If each request has a latency of $r = 2$ cycles (obviously, it is always required that the request fits inside the slot, i.e., $s \geq r$) and the request of a task arrives in the last $r - 1$ cycles of the task's slot, it cannot be granted access to the bus since it would span into the slot of the next core. Indeed, this is the worst situation in terms of BID, which is given by $BID_{worst}^{TDMA} = (N_c - 1)s + (r - 1)$, i.e., 13 in the scenario drawn in Figure 5.1. BID_{worst}^{TDMA} could be taken as a safe upper bound of the inter-task interferences, though it would be often pessimistic (i.e.

Table 5.1: Arrival times of a bus request

Case	Expression
A	$i < c$
B	$i = c$ and $(s - rc) \geq r$
C	$i = c$ and $(s - rc) < r$
D	$i > c$

overestimated). In reality, the exact effect of inter-task interferences that a given request from a task suffers, depends on the cycle in which that bus request arrives with respect to that task's TDMA slot. Note that in both cases (exact and worst case), the computation of the effect of inter-task interferences, does not depend on the other contenders, which allows to derive WCET estimates in isolation for each task, regardless of the workload.

Let's assume that a bus request from thread/core c (where $0 \leq c \leq N_c - 1$), arrives at the absolute cycle n, measured in processor cycles. The slot $i \in [0, N_c{-}1]$, in which the request arrives is given by $i = \lfloor n/s \rfloor \bmod N_c$. n translates into relative cycle $rc \in [0, s-1]$ within slot i as $rc = n \bmod s$. If i matches the core id, c then the request has arrived in its core slot. There are four different scenarios for the computation of the effect of inter-task interferences depending on the relation between the arrival slot i, the sending core c, and whether the request fits in the current slot or not. Table 5.1 shows these four cases (A-D). Cases A and D correspond to issuing the request before and after its own slot respectively; in case B the request is issued in its slot and has enough cycles left to proceed, and in case C the request is issued in the correct slot but without enough cycles left. Only in case B the waiting time is 0 because the request arrives in its corresponding slot and has enough time to be sent. The time the request has to wait, in cycles, to get access to the bus depends on the relative cycle, rc, in which the request arrives, so BID_{exact}^{TDMA} is:

$$BID_{exact}^{TDMA} \begin{cases} (c - i - 1) \cdot s + (s - rc) & \text{case A} \\ 0 & \text{case B} \\ (N_c - i + c - 1) \cdot s + (s - rc) & \text{cases C,D} \end{cases} \tag{5.1}$$

Equation 5.1 matches BID values in Figure 5.1, in the scenario explained before. Figure 5.2 presents a graphical view of the different TDMA BID values. We observe that BID_{exact}^{TDMA} has sawtooth behavior. Also it can be seen that $BID_{worst}^{TDMA} \geq BID_{exact}^{TDMA}$ for any given cycle, which shows the pessimism when obtaining WCET estimates using BID_{worst}^{TDMA} instead of using BID_{exact}^{TDMA}.

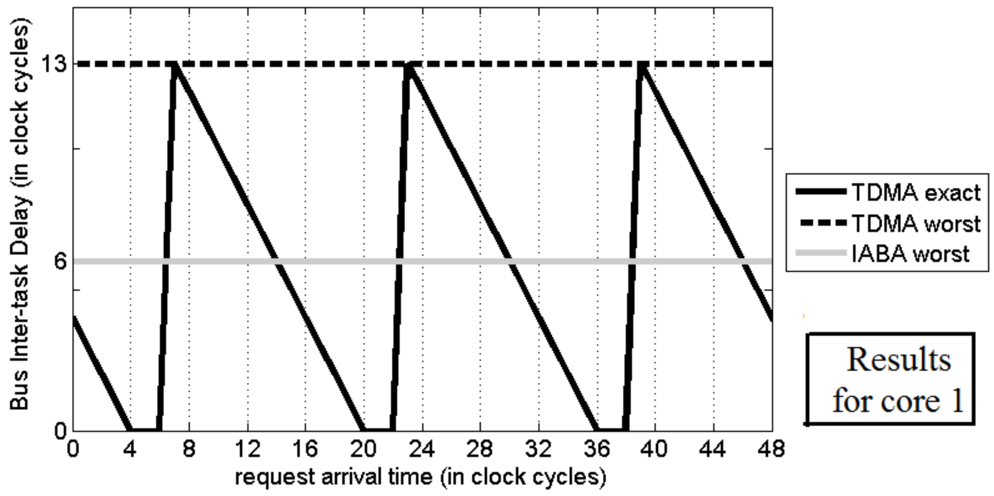


Figure 5.2: TDMA and IABA results with $w = 16$, $s = 4$, $N_c = 4$ and $r = 2$.

5.2.2 IABA

IABA is devised to work on top of an existing arbitration policy. In this chapter, we focus on round-robin because it is a commonly used and simple policy, but the same analysis can be applied to any arbitration policy that allows to upper bound the effect of inter-task interferences. Conceptually, in round-robin all contenders have different priorities, and those priorities are rotated across contenders on every arbitration. Therefore, one core will have the lowest priority in a particular arbitration, the second lowest in the next arbitration, the third lowest in the next arbitration, and so on and so forth until it gets the highest priority after $N_c - 1$ arbitrations. Then, in the next arbitration it gets the lowest priority again. In this way, if one contender has no pending requests, the next contender will be served, being the bus idle only when there are no requests (*work-conserving*). Under round-robin, the maximum delay a request can suffer due to inter-task interferences (contention) occurs when all the other contenders are trying to access the bus and have higher priority. This maximum delay is called BID^{IABA}_{worst} (which is called Upper-Bound Delay (UBD) in [138]). BID^{IABA}_{worst} is bounded by the maximum number of contenders that can send a request at the same time. The latter is, in turn, bounded by the number of cores (N_c). Overall, BID^{IABA}_{worst} is computed as follows, where r is the request latency:

$$BID^{IABA}_{worst} = (N_c - 1) \cdot r \tag{5.2}$$

IABA always assumes BID^{IABA}_{worst} for every access to the bus, in order to be able to perform the WCET analysis of any task in isolation, regardless of the workload (in [47], they reduce the value of BID^{IABA}_{worst} based on the possible workloads, however this breaks the time composability property). This assumption introduces pessimism in the WCET estimate but significantly simplifies the analysis.

Let's assume the same configuration used in Figure 5.1 with N_c=4 and r=2. Further assume that we use round-robin policy. Under this setup, the maximum inter-task delay a request can suffer is 6 cycles, i.e. $BID^{IABA}_{worst} = 6$, which is the delay that corresponds to the accesses of the other 3 cores. BID^{IABA}_{worst} is also shown in Figure 5.2 for this setup. As it can be observed, BID^{IABA}_{worst} is constant regardless of the cycle in which the request arrives, since IABA assumes the worst case for each request. Hence, with IABA the maximum delay that a request will suffer due to bus interferences depends on the number of tasks that are going to be executed simultaneously in the multicore processor, which is bounded by the number of cores N_c, thus eliminating the dependence on the arrival cycle of requests.

5.3 Analytical comparison of TDMA and IABA

We note that in order to minimize the worst-case value, the best choice for TDMA is to use the smallest possible slot size, which is $s = r$, the minimum slot size. In that case $BID^{IABA}_{worst} = (N_c - 1) \cdot r$ and $BID^{TDMA}_{worst} = (N_c - 1) \cdot r + (r - 1)$, so $BID^{IABA}_{worst} \leq BID^{TDMA}_{worst}$, which means that TDMA is never better than IABA.

If for a given application, the exact bus cycle for each access to memory is known, we can use BID^{TDMA}_{exact} instead of the worst-case value. However, BID^{TDMA}_{exact} values can *only* be determined after *all* hardware is designed, when the application binaries can be generated and analyzed on the actual platform (including hardware and system software). Further, any source of uncertainty in the analysis of any component has an impact on BID^{TDMA}_{exact} accuracy.

Chip vendors decide which arbitration policy to implement, typically without knowing the applications that will run on top. In order to help a chip vendor to choose between TDMA and IABA, we can use an expected BID instead of the exact value for TDMA, by assuming that bus requests distribute uniformly over the TDMA window. Intuitively this should be the case, as one does not expect any particular distribution in time at which requests are ready with respect to the window. We verified this assumption on our processor setup, explained in Section III over all the EEMBC benchmarks. The difference between the maximum number of accesses done in any cycle of the window and the minimum number of cycles done in any other cycle (i.e. $\frac{min-max}{min}$) is only: 3.5%, 2.4%, 1.8%, 0.7% and 0.6% respectively for windows of size 128, 64, 32, 16 and 8 cycles. Figures 5.3 show the distribution of accesses per cycle under different TDMA window sizes.

To calculate the expected value of the BID with TDMA, we note that, going back to Figure 5.2, TDMA windows of $N_c \times s$ cycles are comprised of two intervals in terms of BID: The first one of $s-r+1$ cycles with BID value 0, which corresponds

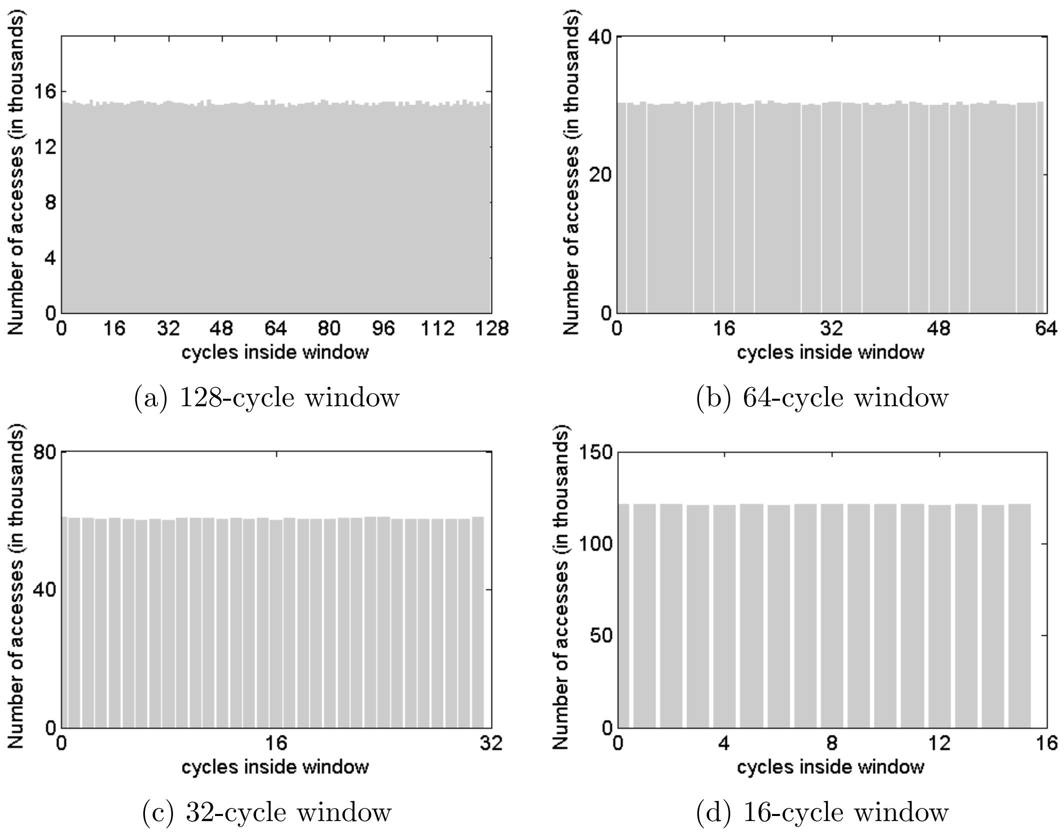
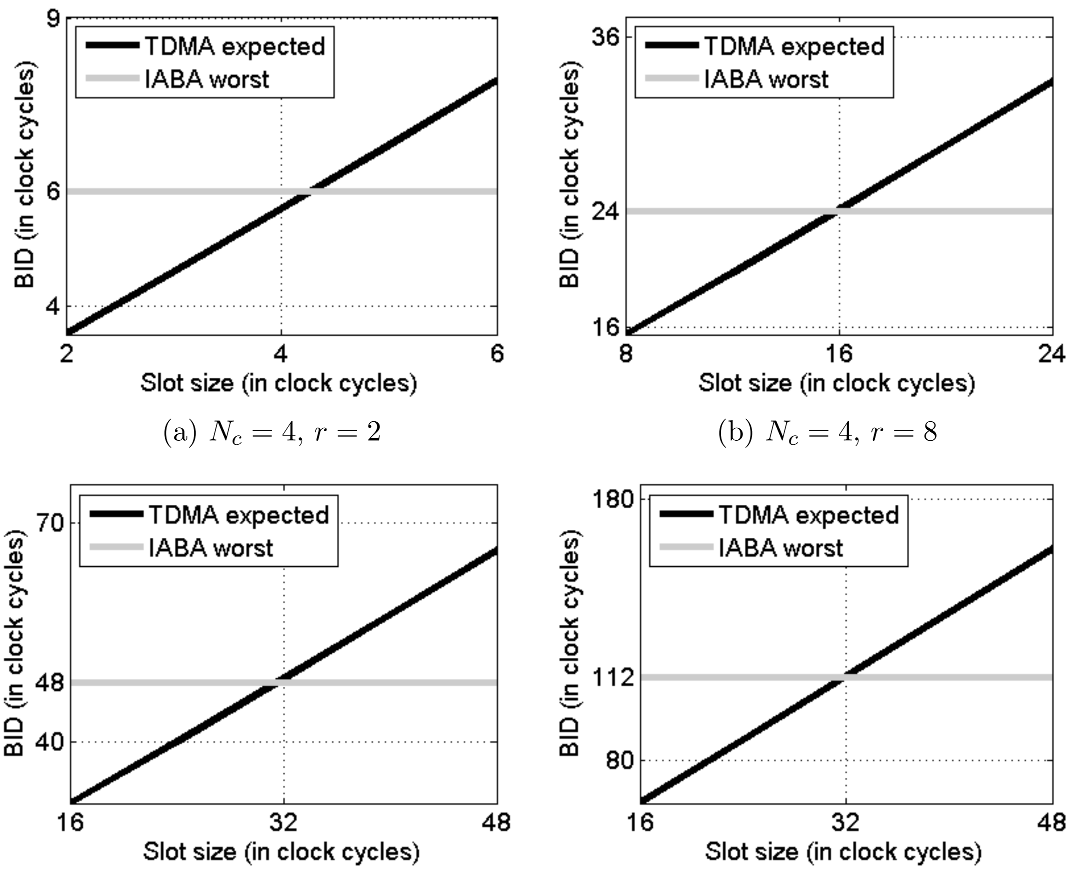


Figure 5.3: Memory accesses histogram inside a TDMA for several window sizes

to the cycles in which the request is in its slot. The second interval, containing the rest of the cycles, forms a linear decay that goes from the maximum BID value, i.e., $(N_c - 1)s + (r - 1)$ down to 1. The average expected BID for both intervals can be expressed as shown in Equation 5.3. In the first addend BID equals 0 for $s - r + 1$ cycles and the second addend corresponds to the linear decay, both addends divided by the number of cycles of the window, $N_c \cdot s$:

$$BID^{TDMA}_{expected} = \frac{0 \times (s - r + 1) + \sum_{i=1}^{(N_c-1)\cdot s+(r-1)} i}{N_c \cdot s} \tag{5.3}$$

Overall, Equation 5.3 allows us to compare IABA and TDMA for any setup including the core count accessing the bus and the bus latency. As an illustrative example, Figures 5.4(a)-(d) show different values for TDMA and IABA. Interestingly, in all cases the cross-point between $BID^{TDMA}_{expected}$ and BID^{IABA}_{worst} occurs when the slot size s is equal to $2 \cdot r$, i.e., TDMA offers better worst-case performance



(a) $N_c = 4,\ r = 2$ (b) $N_c = 4,\ r = 8$

(c) $N_c = 4,\ r = 16$ (d) $N_c = 8,\ r = 16$

Figure 5.4: BID for TDMA and IABA with $N_c = 4, 8$ and $r = 2, 8, 16$

than IABA when the TDMA slot size $s < 2 \cdot r$, as we show in our experiments later.

5.4 Empirical comparison of TDMA and IABA

In this section we qualitatively and quantitatively compare TDMA and IABA using different metrics: required changes at hardware and software level (i.e. analysis tool), WCET estimates obtained, prioritization capabilities, average performance and time composability.

The quantitative metrics have been obtained using the simulator explained in Section 3.3. We model four and eight cores architectures in which each core has private instruction and data caches, which is common in current high-performance and real-time embedded processor designs [42, 84]. The cache hierarchy comprises first-level 4KB instruction cache and 4KB data cache. The shared L2 cache is 256KB in size with 8 banks and 8-way associativity. The cache line size is 16 bytes in all caches. Hits on the data and instruction caches take 1 cycle and misses 2 cycles. Hits to L2 take 3 cycles and misses 1 extra cycle. The access to main memory is 100 cycles. The bus request latency is 2 cycles.

With the aim of evaluating TDMA and IABA, we make the bus the only resource for which tasks in different cores compete. To that end, we deploy existing solutions that partition the L2 cache. The memory bandwidth is partitioned using the techniques proposed for the memory controller in [139, 7].

We used twelve benchmarks of the EEMBC Autobench suite [143] as reference for the analysis of single-path programs. We used: *a2time*, *aifirf*, *basefp*, *cacheb*, *canrdr*, *idctrn*, *matrix*, *pntrch*, *puwmod*, *rspeed*, *tblook* and *ttsprk*.

5.4.1 WCET analysability: feasibility and required changes at hardware level and in the WCET analysis tool

Next we show the main changes required by TDMA and IABA when using Static and Measurement-based timing analysis.

Static Deterministic Timing Analysis (SDTA) SDTA techniques rely on (1) the construction of a cycle-accurate model of the system and (2) determining a trustworthy upper-bound on the WCET. At the hardware level, it is necessary for every resource to have a bounded access latency that is provided as input to the model. The same principle applies to the bus: On the one hand, the time it takes a request to access the bus must be boundable, even in the presence of inter-task interferences. On the other hand, the tighter this bound is, the tighter the obtained WCET estimates. If the bus uses IABA, no change is required in the SDTA tool: Basically, the access latency of each request to the bus is augmented with the BID_{worst}^{IABA}. IABA requires changes only in the configuration of the processor parameters given to the tool but not in the tool itself.

For TDMA, using BID_{worst}^{TDMA} as bound is pessimistic since for any given cycle $BID_{worst}^{TDMA} \geq BID_{exact}^{TDMA}$. If the exact cycle in which each request accesses the bus could be known by the SDTA tool, we could use for each request its BID_{exact}^{TDMA}, thus eliminating the pessimism on the estimate. However, imposing that the STA tool provides the exact cycle in which each bus request occurs is complex [101] or even unfeasible [165]. In reality, for each request, rather than an exact cycle, it can be estimated a time interval [101] in which it can access the bus. This inaccuracy comes from the fact that the structure of the program includes branches, loops, etc., which complicate the estimation of exact cycles, since the time at which one request accesses the bus may depend on the path followed to get to the basic block in which the request is. Note that as soon as the estimated time interval for a request to access the bus is longer than the window, the SDTA tool has to use BID_{worst}^{TDMA} as the bound.

Measurement-Based Deterministic Timing Analysis (MBDTA) MBDTA techniques rely on extensive testing performed on the real system, recording the so-called high watermark execution time, i.e. the longest observed execution time. The high watermark observed during this testing phase is multiplied by an engineering margin. The outcome is used as WCET estimate for the task during deployment time.

Both TDMA and IABA require hardware changes in order to enable the use of MBDTA tools: With IABA, during the testing phase the program under study is run in isolation. Every time a request to the bus is ready it is artificially delayed by the architecture by BID_{worst}^{IABA} cycles, which requires changes in the bus arbiter [138]. From traces obtained during this execution in isolation, a WCET estimate for the task is obtained. At deployment time, the hardware is instructed not to introduce any artificial delay. The key point of this solution is that the artificial delay introduced during testing upper bounds the inter-task interferences that the task can suffer during deployment time [138].

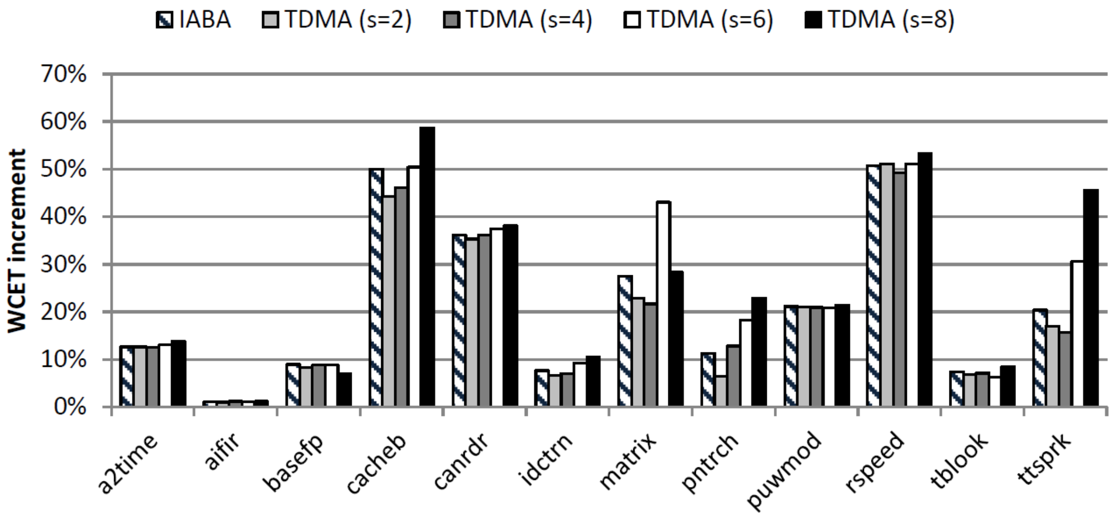
For TDMA, it is critical that every request gets access to the bus exactly in the same cycle during both testing and deployment time, otherwise losing trustworthiness on the results. One possible solution is to synchronize the start cycle of a task with a fixed point of the TDMA window in each execution, by means of some hardware support.

5.4.2 Worst Case Performance

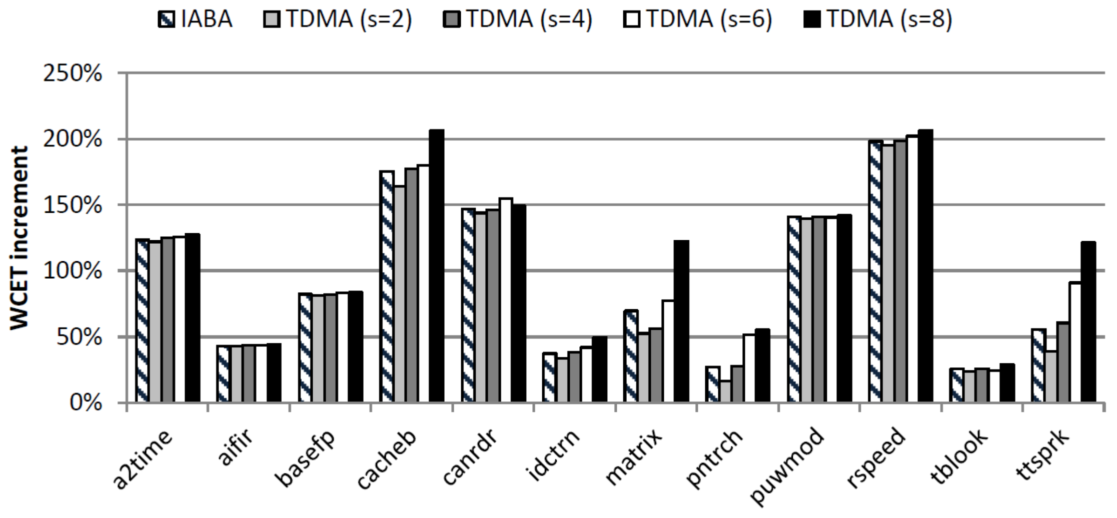
In this chapter we use RapiTime [146], a commercial measurement-based WCET analysis tool. RapiTime uses path analysis techniques to build up a precise model of the overall code structure and determine which combinations of subpaths form complete and feasible paths through the code. RapiTime combines the measurement and control flow analysis information to compute measurement based WCET estimates in a way that captures the execution time variation on individual paths.

For TDMA, we assume two different scenarios: a scenario (1) in which the exact cycle of every bus access is known by the SDTA tool and the scenario (2) in which the exact cycle is unknown and we always have to assume the worst case to give safe bounds. Note that, these two scenarios do not affect the IABA case, which always assumes the worst case, but make a big difference in the case of TDMA.

Case (1) Exact bus access cycle known: Figure 5.5a shows the WCET degradation (increment) obtained for each EEMBC under a 4-core configuration



(a) $N_c = 4$ cores



(b) $N_c = 8$ cores

Figure 5.5: WCET increment when the bus access cycles are known for every request

using IABA and different TDMA setups. As a reference case, we use the WCET estimate obtained when the benchmark runs in isolation in the processor so $BID = 0$. We observe that some benchmarks are barely affected by the bus policy, like *a2time*, *aifir*, *basefp*, *idctrn*, *puwmod* or *tblook* due to their low memory traffic (most of them fit in the L1 caches).

For the rest of the benchmarks, which show some WCET variability depending on the slot size, selecting the best TDMA slot size is tricky. A short slot size makes the overall window size smaller, so regardless of the cycle in which a requests is ready, it has to wait less to get access to the bus. On the other hand, long slot sizes favor bursty requests, i.e., several requests one after the other, since they can be processed consecutively.

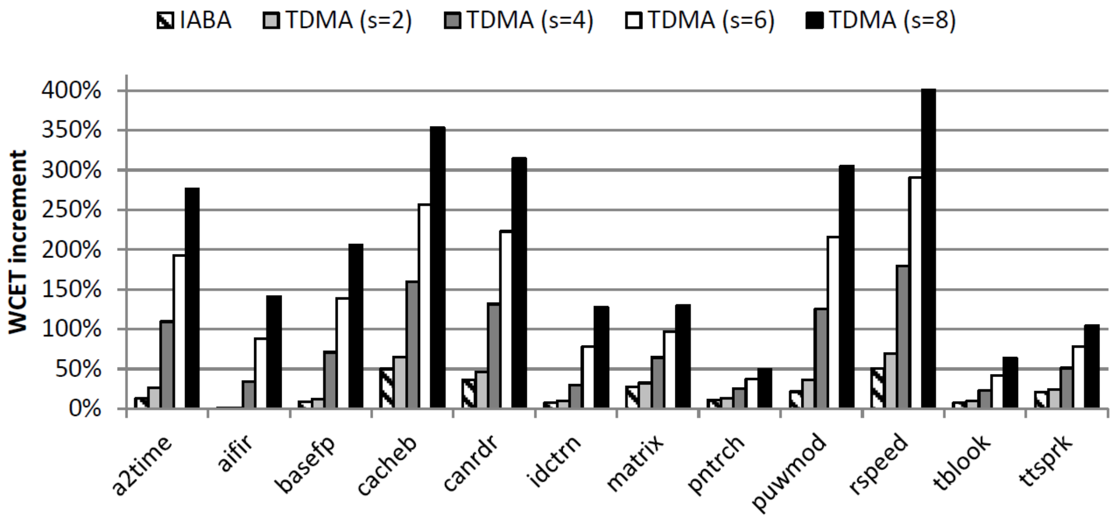
In general, TDMA performs better with smaller slot sizes because the access latencies are smaller. More specifically, when slot sizes are such that $s < 2r$ (as we have seen in Section 5.3), which corresponds to $s = 2$ and 4 in our experiment, the WCET improvement is around 5% for memory-intensive benchmarks.

An interesting case is the *matrix* benchmark for which with $s = 6$ its WCET is 15% worse than when $s = 8$, $s = 2$ and $s = 4$. This high increment can be explained by the bursty access pattern generated by the benchmark: if more than one request fits into the slot, when we have a burst of requests the second and subsequent requests can access the bus immediately after the first one finishes. As a result, bursty requests take advantage of larger slot size. However, if the second and subsequent requests do not fit into the remaining cycles of the slot, they have to wait till the end of current slot and for other core slots, 3 in this case. The burst pattern generated by *matrix* fits when the slot size is $s = 8$, but not for $s = 6$. This behavior is magnified by the repetitive nature of the benchmark, exacerbating the WCET degradation.

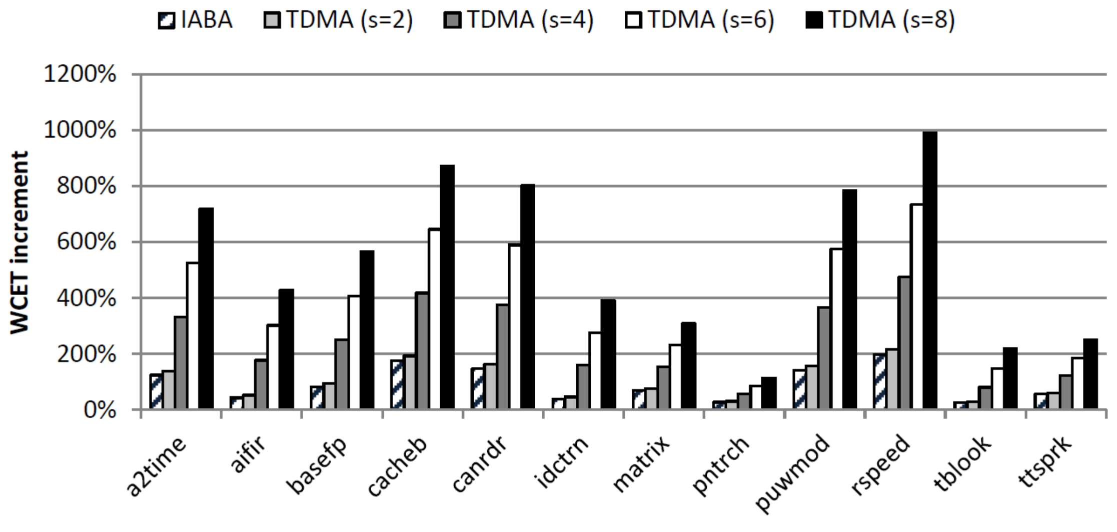
We also observe that IABA is close to the best TDMA setup despite the access time of all requests is known, which is useful for TDMA but not for IABA. On average TDMA leads to WCET estimates around 2% lower than IABA, being *cacheb* the benchmark leading to the highest difference (5.7%).

For the 8-core setup, shown in Figure 5.5b, we observe the same trend. The main difference with respect to the 4-core setup is that the WCET variations are greater because of the higher number of bus contenders. We also observe that there is no benefit of that burst-like behavior mentioned before because the beneficial effect of augmenting the slot size is masked by the higher latencies due to the longer window size (now for the same slot size, the window size is doubled in comparison with 4-cores). We also observe the same trend as for 4-cores that for slot sizes $s < 2r$ TDMA performs better than IABA. More precisely up to 6% and 17% on average.

Case (2) Exact bus access cycle unknown: If the cycle in which requests arrive to the bus cannot be determined, for instance because the SDTA tool cannot be changed or because it cannot be determined with enough accuracy to provide any benefit, we have to use $BID_{TDMA}^{worst} = (N_c - 1) \cdot s + (r - 1)$ in the case of TDMA for SDTA. In general, the smaller the value of s is, the better the result. In the best case, $s = r$ and BID_{TDMA}^{worst} is $(N_c - 1) \cdot r + (r - 1)$. For our 4-core



(a) $N_c = 4$ cores



(b) $N_c = 8$ cores

Figure 5.6: WCET increment when the exact bus access cycle is not known.

setup $BID^{worst}_{TDMA} = 3r + (r-1) = 4r - 1$. With IABA, every request waits for all the other contenders to access the bus, i.e., $3r$. Figure 5.6a shows the WCET increment for the 4-core scenario. In general, we observe that IABA is better than TDMA in all cases (7.6% on average), being the improvement in the case of *puwmod*, *cacheb*, *canrdr*, *rspeed* and *a2time* higher than 10%.

For the 8-core setup, the same trend is observed. Figure 5.6b shows that

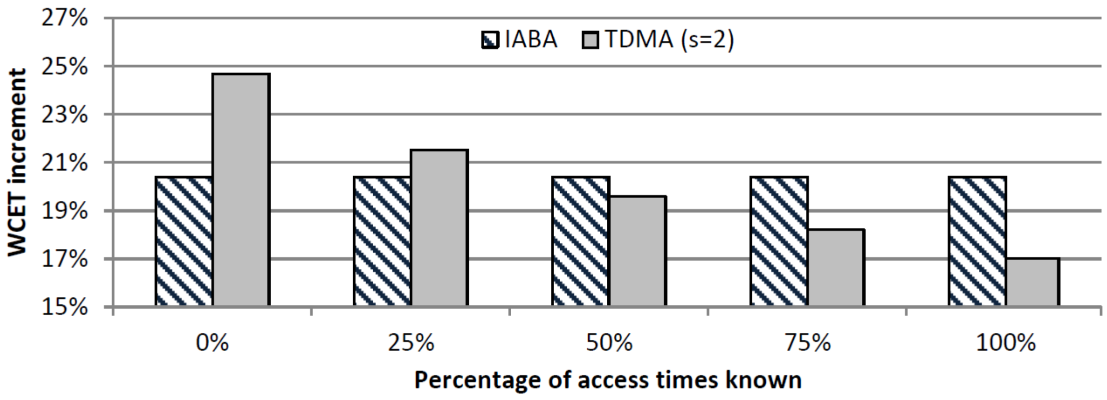


Figure 5.7: WCET increment for *ttsprk* with different percentage of information.

IABA outperforms TDMA by up to 18% and 10% on average. Also for TDMA, as expected, the smaller the slot size the better since a larger slot size means also increasing the worst-case BID.

To complete our study, Figure 5.7 shows the WCET increment for the *ttsprk* benchmark as the percentage of known bus access times varies. This will be the case if the WCET analysis tool can not provide the exact bus access cycle for all the request but only for a percentage of the total number. The 0% case corresponds to the always worst-case scenario (Figure 5.6) and the 100% one to the exact cycle known scenario (Figure 5.5). We observe that the WCET increment for TDMA is inversely proportional to the percentage of access times known. We observe that at least 50% of the exact access times are needed to obtain some gains with TDMA. We observed similar trends when analysing the other EEBMC benchmarks.

5.4.3 Time-bounded Prioritization

From Figure 5.5, we observe that some tasks are not affected by the duration of BID while others are significantly affected. This knowledge can be used to reduce the WCET estimate for those tasks more affected by BID (e.g. *cacheb*) by granting them access to the bus more often, or prioritizing them[1]. We notice that in this context, prioritization does not mean to flush or stop in-flight requests, or to obtain the bus access immediately, it means only to have more resources assigned than other tasks.

With TDMA, prioritization is done by assigning different slot sizes to each task: long slots are assigned to the bus-intensive tasks, leaving the rest of tasks with

[1]This can be done at hardware level identifying each core, which means that we are not actually prioritizing tasks, but rather cores.

shorter slots. For example, we can divide the window in one big slot and several small slots and assign the big slot to a memory-intensive task and the smaller slots to low memory demanding tasks.

With IABA, the resource management can be obtained by using a hierarchical round-robin policy, first proposed in [138] and analyzed in [25]: Tasks are divided into groups and subgroups. For example, if we want to prioritize one task T_1 over other three tasks T_2, T_3 and T_4, we make 2 groups, the first one containing T_1, and the second group containing T_2, T_3 and T_4. Then the arbitration algorithm chooses one of the groups in a round robin fashion. Inside that group tasks are selected also using round robin. The result is that, applying the IABA philosophy, the high priority task, T_1 only has to wait for one round to get access to the bus (i.e. it obtains the bus once every two rounds), and the other three tasks, have to wait for the accesses of the other two tasks in the second group which are interleaved between the corresponding three accesses of the high priority task. Hence, T_2, T_3 and T_4 get access to the bus once every six rounds.

Overall, both IABA and TDMA enable thread prioritization. However, TDMA offers finer granularity than IABA. With IABA tasks can access the bus with a frequency limited by the number of groups and subgroups, and the number of tasks in each group and subgroup. For TDMA we can simply adjust the slot size for each of the tasks, at bus cycle granularity.

5.4.4 Time Composability

In this thesis, we consider time composability in the sense that the WCET estimate for a task τ_i can be computed in isolation and it is not affected by the other tasks that may run concurrently with τ_i. Note that time composability and prioritization are somehow opposed metrics. If we prioritize one task's accesses to the bus, we make this task faster, but the WCET estimates of the other tasks depend on the resources allocated to the prioritized task. It is up to the system designer to define a system cost function to determine which metric is more important, prioritization or time composability.

With TDMA, if we keep the window size fixed, time composability is not affected. For example, let's assume that the case of two tasks, T_1 and T_2, both having a slot size (s) of 5. Further assume that the bus latency of 1 cycle (r). Under this scenario, the worst-case BID is $(10 - 5) = 5$ cycles, no matter what is the schedule of the rest of the tasks. However, if the window size depends on the other tasks' slot sizes, for example, T_2 increases its slot size to 6, which makes a window of $(6 + 5) = 11$, then the worst-case BID for T_1 is $(11 - 5) = 6$ which affects its WCET.

In the case of IABA, if we keep the groups and number of tasks within the groups fixed, time composability is not affected. Otherwise, for example, if we

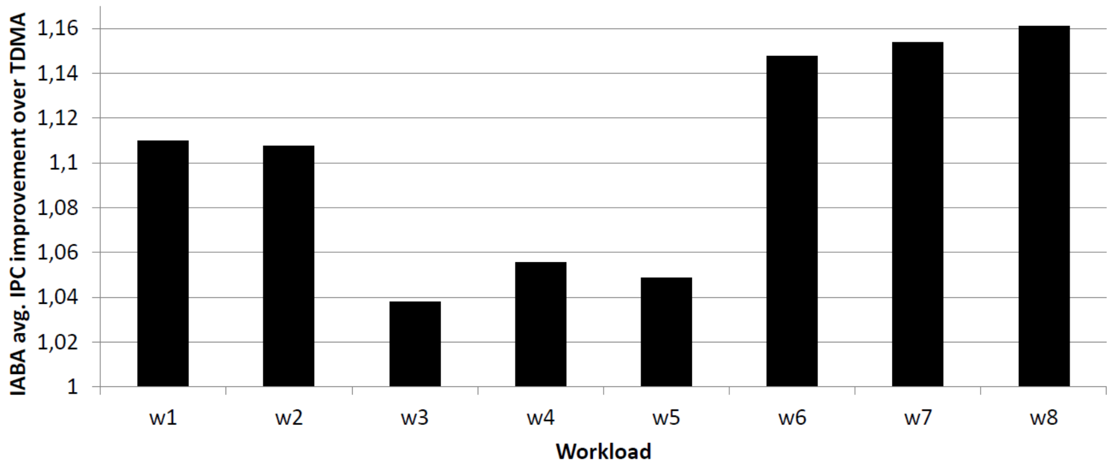


Figure 5.8: Average performance improvement of IABA over TDMA

change one task with one priority for another one with different priority, then the WCET estimates for the other tasks change, thus losing the time composability.

5.4.5 Average Performance

In order to measure the average performance of IABA and TDMA, we run eight 4-thread randomly generated workloads under IABA and TDMA, comparing the performance of each benchmark in terms of IPC (*Instructions Per Cycle*) under both policies. Note that with TDMA, both at analysis and deployment time tasks are not allowed to use the bus out of their slots. With IABA, instead, at analysis time, each bus access of a task is artificially delayed by BID_{IABA}^{worst} cycles. At deployment time, when several tasks run simultaneously, no artificial delay is introduced. The only delay tasks suffer accessing the bus are due to actual inter-task interferences.

Figure 5.8 shows the average performance improvement of IABA over TDMA in terms of the average IPC improvement of all benchmarks in a workload. IABA provides better performance in all workloads than TDMA because it is a work-conserving policy. The improvement ranges from 4% to 16% with an average of 10%.

5.4.6 Discussion

Table 5.2 summarizes the different properties analyzed in this chapter for TDMA and IABA. In this table, we use the following symbols: ++ (very high), + (high), − (low), and −− (very low).

Table 5.2: Summary of the properties of TDMA and IABA

	WCET tightness	Avg perf.	Time composability	Prioritization	HW/SW changes
IABA	++	++	++	+	+
TDMA	+	−−	++	++	-

WCET tightness: Whether the exact cycle of every bus request is known or not, plays a key role in TDMA tightness, while IABA is insensitive to it. Although knowing the cycle in which requests are generated has been identified as difficult – or even impossible – to be obtained by a SDTA tool, we have used it as the best possible scenario for TDMA. Under this scenario, TDMA slightly outperforms IABA for small slot sizes, $s < 2r$, obtaining about 5% lower WCET estimates for 4 cores, for the most memory-intensive benchmarks. We have also seen that, in order to obtain some gains with TDMA, at least 50% of the bus request arrival times must be known. In the more feasible case that exact bus arrival cycles are unknown, IABA largely outperforms TDMA.

Average performance: Since IABA is a work-conserving policy, it achieves significantly higher performance than TDMA.

Time composability: Both techniques are equally time composable if no prioritization is performed across tasks.

Prioritization: In terms of resource management, TDMA offers better (finer) management granularity than IABA, because TDMA allows adjusting the slot of each contender at cycle granularity to match the application's requirements. Still IABA can use *grouping*, which imposes a coarser granularity for prioritization than that of TDMA.

HW/SW changes: While both techniques require hardware support, IABA does not require changes on the static timing analysis tools. If those tools have to provide bus access time intervals for TDMA, they must be deeply changed.

5.5 Benefits for industry

Since several processors use buses as the main communication channel, the study carried out in this chapter provides valuable information for real-time system designers. In particular, this chapter helps hardware and software designers in early design steps to choose which policy best fits their needs.

For hardware designers, at the time the chip is being designed, the information about the target applications may not be accessible. It can also be the case that the chip targets different real-time markets where applications have different profiles in the use of the shared resources including the bus. Our analysis shows that IABA is the best choice because it provides better results and puts much fewer requirements on the information needed from the software designer to provide tight WCET estimates.

If the application environment is known and if software designers can influence the chip design, software designers can decide whether TDMA or IABA is better. In particular our study will help them determining whether the WCET estimate reduction obtained with TDMA (when the exact cycle in which each bus access occurs) pays off the effort of determining those exact cycles or if, instead, IABA is accurate enough. The BID (obtained with our analysis) of the arbitration policy chosen can be provided to the early-stage timing analysis tools, usually called Code-Level Timing Analysis (CLTA) tools, to determine the WCET tightness with TDMA and IABA.

5.6 Summary

Bounding the effect of inter-task interferences is of paramount importance to provide meaningful WCET estimates in multicore processors for CRTES. In this chapter, we have evaluated and compared the two most used bus arbitration policies, TDMA and IABA, intended to deal with inter-task interferences in On-Chip buses. Concretely, we have seen that both policies can provide WCET analyzability and time composability, since WCET estimates for applications can be computed in isolation regardless of the workload executed concurrently. IABA presents worse prioritization capabilities and better WCET and average execution time with little burden for the user.

Chapter 6

Dual-Criticality Memory Controller

6.1 Introduction

Multicore mixed-criticality systems [169] can consolidate onto the same hardware applications with different criticality levels in terms of safety (and security). While in other domains safety standards define multiple safety integrity levels (e.g. DAL in avionics and ASIL in automotive), in the space domain it is well accepted that on-board systems will comprise two criticality levels [142]. One level covers *control* applications, which require real-time execution and are designed to meet requirements in the worst case. Control applications usually have low memory footprint, in the order of kilobytes, so that if they are provided some cache space, they incur low number of memory accesses. The second covers *payload* applications that are high-performance driven, usually with data footprints in the order of megabytes, and some (soft) real time requirements.

Contribution: Memory bandwidth, which is arbitrated by the memory controller, is one of the shared resources with the highest impact on systems' average and guaranteed performance [139, 183]. In this chapter we tackle the challenge of handling inter-task interferences, a.k.a contention, in the memory controller in multicore systems for the space domain. To that end we propose a Dual-Criticality memory controller *DCmc*, which provides real-time guarantees for *control* applications and high-performance for *payload* ones. Instead of deploying a single policy to schedule requests of different types, which inevitably ends up trading off some performance for real-time guarantees, *DCmc* virtually divides memory banks into real-time and high-performance banks that are managed differently by the memory controller, deploying a different scheduling in each case. For the real-time banks, *DCmc* deploys round-robin scheduler across the different requestors to provide predictable and tight bounds on the memory latency. For high-performance banks *DCmc* deploys a First-Ready First-Come First-Served (FR-FCFS) [150] scheduler, similar to the one in COTS high-performance processors [102], to exploit locality of accesses and improve the memory throughput. Further, *DCmc* prioritizes real-time requests over high-performance ones, providing high degree of isolation for critical applications running in dual-criticality workloads. *DCmc* includes support to enable the Operating System (OS) to manage the separation of banks dynamically at run-time. This provides flexibility to distribute banks according to tasks' needs in each system instantiation.

Evaluation: We perform a detailed analysis of the timing behavior of *DCmc* and compare it with the state of the art memory controllers both analytically and quantitatively.

Our results show that for the real-time tasks, tight WCET estimates can be derived with *DCmc*, regardless of the co-running payload tasks, effectively isolating them from payload tasks running on the system. These WCET estimates are significantly tighter than those obtained with FR-FCFS [102]. Further, *DCmc* allows payload tasks to exploit the memory locality, observing a performance degradation of less than 1% due to the execution of real-time tasks.

Applicability in other domains: *DCmc* applies to other dual-criticality real-time domains. There are plenty of systems in cars, planes, etc. related to infotainment or devised for driver/pilot assistance that are not critical but require high performance, so they can be consolidated onto the same multicore as other critical real-time functions and so, our solution is completely valid in those domains. For instance, we completed the *DCmc* evaluation by using EEMBC benchmarks as representatives of control real-time applications. Obtained results in terms of real-time guarantees and high performance follow the same trend as for the space case study.

The rest of this chapter is organized as follows: Section 6.2 presents an analysis of the worst-case memory latency under different memory controller setups. In Section 6.3, the *DCmc* is proposed and in Section 6.4 it is evaluated. Section 6.6 presents the summary of this chapter.

6.2 Analysis of Memory Access Latency

Once a request arrives at the memory controller, its latency, τ, can be divided into intrinsic latency and request interference delay. The former accounts for the time it takes the request to be processed once it is granted access, τ_{req}. The latter accounts for the impact of contention, Δ:

$$\tau \quad = \quad \tau_{req} + \Delta \tag{6.1}$$

There are three main design choices that affect a request's latency (see Section 2.2.2): (1) The *row-buffer policy*, (2) the *memory mapping scheme* and (3) the memory request *scheduling policy*. As a necessary step towards understanding *DCmc*, this section presents the impact that each memory controller design choice has on determining the upper bound latency of a memory request, required in real-time domains. All references to the publications used in this section are in the related work Section 6.5.

6.2.1 Row Buffer Policy

In the absence of interference, the request latency, τ_{req}, depends on the row buffer policy chosen. It is $\tau_{close-req}$ for close-page and $\tau_{open-req}$ for open-page. They are defined as shown in Equation 6.5 and 6.6.

On the event of an access to a non-open row (see Equation 6.3), we need to activate the row first, with t_{RCD} latency. Once the row is open, the request latency covers the column access, t_{CAS} or t_{CWD}, and transferring the data, t_{BURST}, which coincides with the latency of a row-hit. A row-hit, see Equation 6.2, happens when the requested data is on the open row. Finally, for a row-miss (Equation 6.4), which happens when a different row is open in the row buffer, the row is first precharged, with t_{RP} latency, before being activated. These latencies can be expressed as:

$$\tau_{hit-row} = max(t_{CAS}, t_{CWD}) + t_{BURST} \tag{6.2}$$

$$\tau_{closed-row} = t_{RCD} + \tau_{hit-row} \tag{6.3}$$

$$\tau_{miss-row} = t_{RP} + \tau_{closed-row} \tag{6.4}$$

$$\tau_{close-req} = \tau_{closed-row} \tag{6.5}$$

$$\tau_{open-req} = \begin{cases} \tau_{hit-row} & \text{if row-hit} \\ \tau_{closed-row} & \text{if closed-row} \\ \tau_{miss-row} & \text{if row-miss} \end{cases} \tag{6.6}$$

Once we have the intrinsic request latency, we derive the interference (delay) that other requests can generate under the different row buffer policies.

Definition 1 Inter-request worst-case interference under close-page, Δ_{close}. *Under close-page row-buffer policy, the worst-case interference that a request suffers from another request, Δ_{close}, corresponds to the case in which both requests target the same bank.*

Under close-page, a request consists of the sequence of commands ACT, CAS or CWD, and PRE. Assuming that requests are served consecutively, Δ_{close} is determined by the ACT-to-ACT time between the ACT commands of the two requests and is defined in Equation 6.7. The ACT-to-ACT time can be limited by the row-cycle constraint, t_{RC}, which defines the interval between ACT to the same bank. In the case of read requests, the time between CAS and PRE commands has to satisfy the read-to-precharge constraint, t_{RTP}, and the minimum t_{BURST} to be able to send the data before precharging. For writes, the write recovery time, t_{WR}, needs to be satisfied before precharging. Annex I in [95] provides a graphical description of the ACT-to-ACT latency depending on the type (read or write) of the previous request.

$$\Delta_{close} = max(\Delta_{write}, \Delta_{read}) \tag{6.7}$$
$$\Delta_{write} = max(t_{RCD} + t_{CWD} + t_{BURST} + t_{WR} + t_{RP}, t_{RC}) \tag{6.8}$$
$$\Delta_{read} = max(t_{RCD} + max(t_{RTP}, t_{CAS} + t_{BURST}) + t_{RP}, t_{RC}) \tag{6.9}$$

Definition 2 Inter-request worst-case interference under open-page, Δ_{open}. *The highest (worst) interference that a request suffers from another request in an open-page scheme, Δ_{open}, manifests when both are row-misses and the latter hits a different row in the same bank as the former.*

Δ_{open} maximizes when all accesses are row-misses. In that scenario, every memory request accesses a row different to the one active in the row buffer, making each request to send PRE, ACT and CAS/CWD commands. The interference corresponds to the PRE-to-PRE time, which has the same timing constraints as the ACT-to-ACT time under close-page. This occurs because the sequence of commands is essentially the same for close-page and open-page when all accesses are row-misses. Thus, close-page and open-page policies have the same worst-case interference [140], i.e. $\Delta_{open} = \Delta_{close}$.

6.2.2 Memory Mapping Scheme (MMS)

The MMS defines the banks accessed by a request based on its memory address, hence impacting the conflicts that requests have in the access to memory banks. A bank conflict happens when a request waits for another one that is accessing the same bank. The interference that the former suffers, called *intra-bank interference*, manifests when several requests share a bank. When those requests do not share a bank, they can still have bus conflicts when accessing the memory command and data buses. In that case, the interference is called *inter-bank interference*.

In real-time designs, the MMS is selected to reduce the conflicts among requests, and so reduce request's interference in the access to memory. To do so, one common choice is the use of a private bank scheme in which each core has exclusive access to certain banks, effectively removing intra-bank interferences across tasks [148].

Definition 3 Inter-request worst-case interference under private bank, $\Delta_{private}$. *Under private bank, the worst-case interference that a request from a given task suffers from a request of a different task, $\Delta_{private}$, is the inter-bank interference on the access to the command and data buses.*

Under private bank, assuming that each request comprises the commands PRE, ACT, and CAS/CWD(R/W), commands from different banks are scheduled independently. The inter-bank interference that such a request suffers can be split into the interference that each of those commands suffers independently when accessing the command and data buses [102]:

$$\Delta_{private} = \Delta_{PRE} + \Delta_{R/W} + \Delta_{ACT} \tag{6.10}$$

A PRE command can only be interfered by other commands using the command bus, which is given by the time between commands, t_{CMD}. A CAS/CWD command is delayed in the worst-case by another column command sent to another bank, which corresponds to the write-to-read, t_{WTR}, and read-to-write, t_{RTRS}, timing constraints. Annex I in [95] provides a graphical description of the $\Delta_{R/W}$ latency. The ACT command is interfered in the worst-case by other ACT commands, due to ACT-to-ACT timing constraints. The time between two ACTs to different banks is limited by t_{RRD}, and a maximum of four ACTs can be issued during the t_{FAW} time-frame, to restrict the peak current. In the last case, the worst-case interference happens when the other command is an ACT command that is the fourth consecutive ACT so that t_{FAW} does not allow the actual ACT to be scheduled:

$$\begin{aligned}
\Delta_{PRE} &= t_{CMD} && (6.11)\\
\Delta_{R/W} &= max(t_{CWD} + t_{BURST} + t_{WTR}, t_{CAS} + t_{BURST} + \\
&\quad t_{RTRS} - t_{CWD} && (6.12)\\
\Delta_{ACT} &= max(t_{RRD}, t_{FAW} - 3t_{RRD}) && (6.13)
\end{aligned}$$

However, the use of the private bank scheme in shared memory models makes more difficult the communication among cores. Moreover, with private banks the memory is partitioned regardless of the specific application requirements, so applications with very small memory footprint allocate one bank, resulting in a waste of memory. Finally, the private bank scheme has scalability problems due to the limiting number of memory banks (up to eight in case of DDR3 memories). This can be partially mitigated by using more memory ranks, which allows to have more banks [117].

Under interleaved bank scheme [139, 7], data are mapped across all memory banks so that every request accesses all banks simultaneously in a pipelined fashion exploiting bank-level parallelism, and hence removing bus conflicts, or inter-bank interference, among memory requests. For example, in a four-bank interleaved access, four pipelined memory accesses to different banks will be sent per memory request. This scheme is not optimal when the length of a memory request is smaller than the bandwidth of all memory banks because all banks are accessed anyway. We discuss this point in detail in the next section.

Definition 4 Inter-request worst-case interference under interleaved bank, $\Delta_{interleaved}$. *Given a request, the worst-case interference that another request generates on the former in an interleaved bank scheme, $\Delta_{interleaved}$, is given by the intra-bank interference between command sequences accessing all banks.*

In the case of an interleaved bank scheme, every request consists of the same precomputed sequence of commands that access all N_{banks} banks. The interference that a request generates on another one is given by the time from one sequence of commands to the next one, i.e., the ACT-to-ACT time from requests going to the same bank, which matches Equation 6.7. We need to consider also the timing constraints on the data bus for the different bank access that each request does, t_{BURST}, and the limitation of ACT commands that is imposed by t_{RRD}. For illustration purposes, Equations 6.14 and 6.15 show a simplified version of the exact interference, without considering t_{FAW}, read-to-write and write-to-read effects. The exact interference can be found in [6, 139]:

$$\Delta_{interleaved} = max(\Delta_{ACTB} N_{banks}, \Delta_{close}) \tag{6.14}$$
$$\Delta_{ACTB} = max(t_{RRD}, t_{BURST}) \tag{6.15}$$

Definition 5 Inter-request worst-case interference under shared bank, Δ_{shared}. *The worst-case interference that a request suffers from another one in a shared bank scheme is the intra-bank interference generated by the latter if both share the same bank, or the inter-bank interference generated by the latter if it goes to a different bank.*

Equation 6.16 defines Δ_{shared}, where Δ_{inter}, the inter-bank interference, is equivalent to the interference of private bank and Δ_{intra}, the intra-bank interference, is equivalent to the interference of interleaved bank, considering only one bank access.

$$\Delta_{shared} = max(\Delta_{intra}, \Delta_{inter}) \tag{6.16}$$
$$\Delta_{intra} = \Delta_{interleaved}(N_{banks} = 1) \tag{6.17}$$
$$\Delta_{inter} = \Delta_{private} \tag{6.18}$$

Note, that Δ_{shared} is the interference caused by only one request. In the case when several requests can be scheduled concurrently, for instance, requests going to different banks, the worst-case interference includes both, intra-bank and inter-bank interference, as we show in Section 6.3.2. In that case, both terms are not independent between them and depend on the memory scheduler.

6.2.3 Memory Scheduler

The memory scheduler selects the next request to access the memory. It is probably one of the most important components of the memory controller, and the one making the difference between real-time and high-performance designs.

In multi-core real-time designs, the scheduler is designed to bound the impact of interferences across requestors, by using predictable arbitration policies, e.g. round-robin [140].

Definition 6 Inter-requestor worst-case interference under round-robin scheduler, Δ^{rr}. *The worst-case interference that a request from a requestor i, r_i, may suffer from the requests of other requestors under round-robin memory scheduler, Δ^{rr}, corresponds to the case in which all requestors get a request ready in the same cycle and r_i gets the lowest priority.*

Assuming N_{req} requestors under round-robin arbitration, a request has to wait in the worst-case for $N_{req} - 1$ requests, one for each of the other requestors. The effect that each of these other requests has on the former is given by the inter-request interference ($\Delta_{private}$, $\Delta_{interleaved}$ or Δ_{shared}). For instance, with a private scheme, the worst-case inter-request interference is given in Equation 6.10. For shared bank and interleaved bank, Δ^{rr} is computed by changing $\Delta_{private}$ by Δ_{shared} and $\Delta_{interleaved}$ respectively.

$$\Delta^{rr}_{private} = (N_{req} - 1) \cdot \Delta_{private} \tag{6.19}$$

If instead of a real-time amenable scheduler policy, such as round robin, a high-performance scheduler policy is used, such as FR-FCFS, bounds can still be derived on the memory latency [102], though these are less tight. It is required a *reordering term* to be introduced in Equation 6.20, which in order to be able to derive meaningful bounds on latency, has to be limited by hardware. In fact, this is the case for the COTS processor analyzed in [102]. That limit is also useful to prevent memory performance attacks [130].

$$\Delta_{FR-FCFS} = \Delta_{reordering} + \Delta'_{intra} + \Delta'_{inter} \tag{6.20}$$

Further, with FR-FCFS the prioritization of ready requests when scheduling between banks can have unbounded latency, due to an unbounded Δ'_{inter}, as shown in [183]. For instance, whenever there are two requestors sending write requests continuously, a read request can have unbounded delay due to the write-to-read constraint. If the write-to-read constraint is bigger than the write-to-write constraint, writes will be ready before the read, thus being sent before the read and delaying every time the read with the write-to-read, t_{WTR} constraint. In [183], authors change the FR-FCFS policy with a FIFO policy to remove this effect.

FR-FCFS is a clear exponent of opposing metrics between time-predictability and high-performance, since it improves performance [150] on the average-case, however, the effects that it introduces on the interference bounds affect the predictability of the memory controller in the worst-case.

6.3 Dual-Criticality memory controller (DCmc)

High-performance and real-time behavior are somehow opposing goals, since the latter requires reserving hardware resources (either temporally or spatially), which negatively impacts the former. In our space system, these two objectives are structured hierarchically. *DCmc* focuses on providing time predictability to real-time tasks for their correct operation. Once real-time guarantees are satisfied, available resources are used to maximize the average performance of high-performance tasks. *DCmc* design pursues the same two hierarchical goals: providing predictable and tight memory access-latency bounds for real-time tasks and maximizing the average memory performance of high-performance tasks.

DCmc is driven by two design principles DP. DP_1, reducing the interference that high-performance (payload) tasks introduce on the real-time (control) tasks. And DP_2, during those periods in which no real-time memory request is processed, maximize the throughput of high-performance memory requests.

DP_1 is achieved by virtually dividing the banks into those serving requests from/to real-time tasks and the rest which serve high-performance tasks' requests. Bank separation between both application types reduces the interference that high-performance applications introduce on real-time ones. However, bank separation affects Bank Level Parallelism. Interferences across application types are also reduced by prioritizing real-time banks over high-performance ones, such that only if a high-performance request is in-flight by when a real-time request arrives, the latter is delayed by the former. Other than in this case, high-performance requests execute transparently, i.e. without delaying, real-time ones.

DP_2 is achieved by having a memory controller structure in which, during those periods where no real-time requests are processed, high-performance requests can proceed as fast as in high-performance memory controllers, e.g. FR-FCFS.

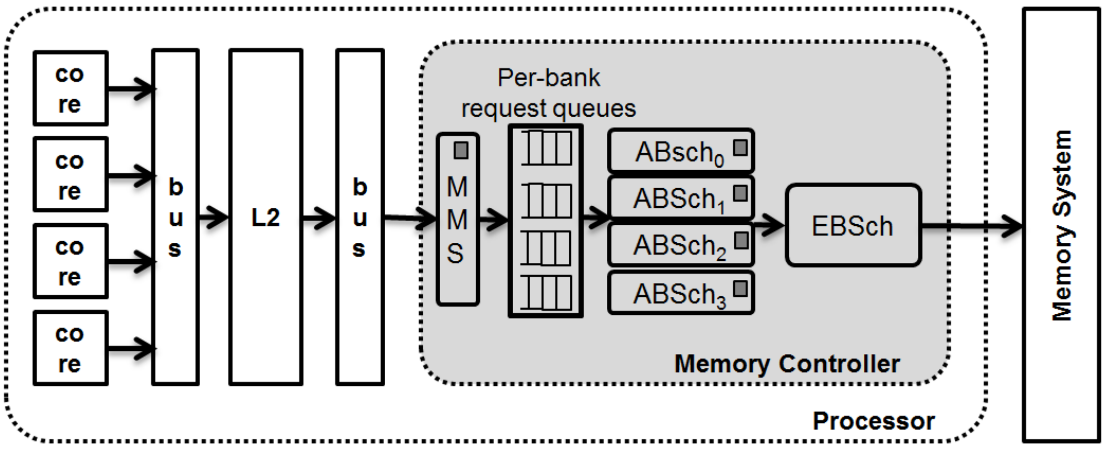


Figure 6.1: *DCmc* architecture. *ABSch$_i$* stands for Intra-Bank Scheduler for bank i. *EBS* stands for Inter-Bank Scheduler. Grey squares show the blocks that communicates with the OS.

Hence, instead of arbitrating high-performance and real-time requests with a single scheduling policy, which would trade both metrics, *DCmc* makes them going to different banks and hence allowing real-time and high-performance requests to be scheduled differently.

DCmc architecture is shown on Figure 6.1. The MMS allocates each request into different per-bank request queues. Each bank is defined as real-time or high-performance. The *intra-bank* scheduler (ABsch) determines the particular commands and their schedule for each bank. The *inter-bank* scheduler (EBsch) grants access to the memory bus to one intra-bank scheduler at a time. Instead of having a fix separation of banks among real-time and high performance, which intrinsically incurs in inefficiencies in real-time guarantees and high performance, *DCmc* enables the operating system to configure bank separation at runtime, i.e. to configure each bank as real-time or high-performance. As a result, on each system instantiation the operating system may distribute banks in a different manner to increase efficiency.

6.3.1 *DCmc*: MMS, RBP, and Scheduler

Memory Mapping Scheme. As we have seen on Section 6.2, we can choose between interleaved, private and shared bank schemes. Equations 6.10 and 6.14 show that private and interleaved bank schemes can reduce the interferences, suffering only from inter-bank or intra-bank interferences respectively. Despite its advantages, an interleaved scheme is not compatible with the MMS required to separate the real-time banks from the high-performance ones, since a given request would not be able to access all banks, as required in an interleaved scheme. Also, the interleaved bank scheme is not suitable when the data needed by a request, usually a last-level cache line, requires only one access to a bank, as it happens in most systems [102, 42]. In particular, for our system, the NGMP multicore processor, generates requests of 32 bytes [42] and the DDR2 system used provides 32 bytes per bank. This makes more efficient to access only one bank at a time, rather than accessing all of them as it would happen with an interleaved scheme.

The hardware-based private scheme lacks scalability due to the reduced number of banks and also has problems with memory usage and enabling the communication among cores as stated on Section 6.2.

For *DCmc* we use a shared bank scheme which is more flexible. With *DCmc*, as opposed to what was shown in Equation 6.16, a request suffers both, inter-bank and intra-bank interferences, since it competes in the intra-bank scheduler and in the inter-bank scheduler. To be able to enjoy the flexibility of shared bank and the reduced interference of private bank, *DCmc* uses software bank partitioning [124] by exposing the MMS to the OS. This makes the OS aware of the address-to-bank mapping, which can allocate tasks into a given bank [187, 102] with the help of an MMU. For instance, in a system with four memory banks, addresses starting with 0x00, 0x01, 0x10 and 0x11 go to banks 0, 1, 2 and 3 respectively. The OS will map any real-time application data and code into real-time banks so that they enjoy predictable latencies. If a single task is assigned to a given real-time bank, it enjoys the benefits of a private bank scheme. Alternatively, if several real-time tasks are assigned to a given bank, they have a shared-bank scheme.

Intra-bank schedulers. The intra-bank scheduler selects the order in which the requests targeting a given bank are prioritized.

For real-time banks we use a policy that allows deriving bounds on the interference that requests generate on each other. In particular we use round-robin as in [139], since it is implemented very efficiently on hardware, it has predictable and composable bounds, as shown on Equation 6.19, with small hardware support [138] and also is work-conserving, providing better average case performance than other predictable policies like TDMA [93].

In order to exploit both, Bank Level Parallelism (BLP) and Row-Buffer Locality (RBL) in high-performance banks, *DCmc* schedules requests per bank, so that it can effectively exploit BLP, and prioritize requests that target open rows, i.e. row hits. Our choice is to use the FR-FCFS [150, 102] as scheduling policy, which chooses first row hits (First Ready) and then in arrival order (First Come First Served), that is also used in nowadays COTS processors [102]. This allows high-performance tasks to benefit from open-page policies, that in turn benefit from locality of accesses.

Inter-bank scheduler. The inter-bank scheduler is round-robin, which is applied across the commands selected by the intra-bank scheduler, having real-time banks priority over high-performance ones. Round-robin is applied to ready commands in the case of ACT and PRE commands, but for CAS/CWD commands it will be applied for all commands, whether they are ready or not, with the scheduler waiting in the latter case (instead of issuing the next command in case it was ready). This prevents very high latencies as explained in Section 6.2 and in [183] for FR-FCFS policy.

Row buffer policy. *DCmc* uses open-page in both, real-time and high-performance banks. For high-performance banks, open-page is required to exploit RBL. For real-time banks, we have seen in Section 6.2 that in the worst-case open-page and close-page policies are equivalent in terms of interference (see Equation 6.7). Open-page has predictable latencies and also enables to exploit RBL when using a private bank scheme. Under a shared bank scheme, for real-time banks, all accesses need to be assumed as row misses, since a requestor is not able to know which accesses perform the rest of requestors sharing the bank.

6.3.2 Memory-access latency analysis under *DCmc*

The ultimate purpose of the real-time banks is to enable deriving WCET estimates in the presence of contention in the access to memory. This requires being able to derive upper-bounds on the interference that do not depend on the rest of requestors (tasks), especially the high-performance ones.

Under *DCmc*, Equation 6.1 is redefined as shown in Equation 6.21. The latency of a request, τ^{DCmc}, is divided into the intrinsic request latency, τ_{req}^{DCmc}, and the interference delay, Δ^{DCmc}. The interference delay is further divided into the interference generated by the real-time banks, Δ^{rt}, and the one generated by the high-performance ones, Δ^{hp}. The real-time interference can be further split into *inter-bank interference*, Δ_{inter}^{rr}, which manifests in the inter-bank round-robin (rr) scheduler; and *intra-bank interference*, Δ_{intra}^{rr}, in the intra-bank round-robin scheduler:

$$\begin{aligned}\tau^{DCmc} &= \tau_{req}^{DCmc} + \Delta^{DCmc} = \tau_{req}^{DCmc} + \Delta^{rt} + \Delta^{hp} = \\ &\tau_{req}^{DCmc} + (\Delta_{inter}^{rr} + \Delta_{intra}^{rr}) + \Delta^{hp} \end{aligned} \tag{6.21}$$

The different latencies of a request under an open-page policy are shown in Equations 6.2, 6.3 and 6.4. In case other requestors use the same bank or when the analysis tool is unable to analyze the state of the row-buffer, all accesses have to be considered row-misses. If we denote by N_R the number of requestors in the bank:

$$\tau_{req}^{DCmc} = \begin{cases} \tau_{hit-row} & N_R = 1 \text{ and row-hit} \\ \tau_{closed-row} & N_R = 1 \text{ and closed-row} \\ \tau_{miss-row} & \text{otherwise} \end{cases} \tag{6.22}$$

Following a similar analysis like the one in [102] we upper bound the interference that any memory request can have in *DCmc*. The *inter-bank* interference affects the commands sent by the request. In the worst-case, and access consists of ACT, CAS/CWD (R/W) and PRE commands, which under round-robin scheduler, generate a worst-case interference when all the other real-time banks accessing the memory have higher priority. If we have N_B real-time banks, the interference will be $N_B - 1$ times the interference suffered by each command independently, equivalent to the interference derived in Equation 6.19:

$$\Delta_{inter}^{rr} = (N_B - 1) \cdot (\Delta_{ACT} + \Delta_{R/W} + \Delta_{PRE}) \tag{6.23}$$

The value of Δ_{ACT}, $\Delta_{R/W}$ and Δ_{PRE} is the same as the ones derived for private bank in Equations 6.11, 6.12 and 6.13.

The *intra-bank* interference is caused by the intra-bank scheduler, that in our case is round-robin. If we have N_R requestors in the bank, for a round-robin scheduler, the worst-case corresponds to waiting for all the rest of requestors, i.e., $N_R - 1$. The worst-case request time that other requests might take is a row-miss also considering the possible inter-bank interference. We need to take into account the row-cycle time, t_{RC}, which is the time between ACT commands to different rows in the same bank, which is also affected by the Δ_{ACT} and Δ_{PRE} inter-bank interference. That is:

$$\begin{aligned} \Delta_{intra}^{rr} &= (N_R - 1)\Delta_{lid-req} && (6.24) \\ \Delta_{lid-req} &= max((N_B - 1)\Delta_{ACT} + (N_B - 1)\Delta_{PRE} + \\ &\quad t_{RC}, \Delta_{inter}^{rr} + \tau_{miss-row}) && (6.25) \end{aligned}$$

With *DCmc*, high-performance banks incur low interference on real-time banks, however, a high-performance request can still cause some interference. The worst-latency, Δ^{hp}, appears when the high-performance memory request is issued just one cycle before the real-time request arrives. If $N_B < N_{banks}$, so there are high-performance banks, this may happen. Δ^{hp} causes the same inter-bank bank interference as a real-time request but removing t_{CMD} cycles for each command:

$$\begin{aligned} \Delta^{hp} &= Q(\Delta_{ACT} + \Delta_{PRE} + \Delta_{R/W} - 3t_{CMD}) && (6.26) \\ Q &= \begin{cases} 1 & N_B < N_{banks} \\ 0 & \text{otherwise} \end{cases} && (6.27) \end{aligned}$$

It is clear that the only input parameters of τ^{DCmc} are the number of real-time banks, N_B, and the number of requestors sharing the same bank, N_R, which both are known by the OS at the moment of scheduling the task. It is important to notice that the request latency bound does not depend on the specific behavior of

Table 6.1: Worst-case access latencies for a DDR2-667 device (memory cycles).

		NR			
		1	**2**	**3**	**4**
NB	**1**	27	50	73	96
	2	40	70	100	130
	3	53	96	139	182
	4	56	112	168	224

the rest of contenders, thus enabling deriving time-composable WCET estimates. In Table 6.1 we derive worst-case latencies for different scenarios of N_B and N_R, assuming that all accesses are row-misses.

6.4 Evaluation

In this section, we provide quantitative evidence of the real-time and high performance properties of *DCmc*.

We use the validated simulator presented in Section 3.3 that models the NGMP. The shared second level (L2) cache is split among cores, each receiving one way of the L2, so that inter-task contention only happens on the memory controller. DL1 is write-through and all caches use LRU replacement policy. The bus connecting the cores to the memory controller uses a round-robin arbitration scheme.We model a 2-GB one-rank DDR2-667 [104] with 4 banks, burst of 4 transfers and a 64-bit bus, which provides 32 bytes per access, i.e., a cache line.

To derive WCET estimates we use the upper bound delay for the bus, as presented in [138], and the worst-case latency derived in Section 6.3 for memory accesses. The memory access latency analysis can be applied either directly with static timing analysis techniques or by means of a worst-case mode [138] in case of measurement based techniques. Since the L2 is split among cores, it does not have any contention.

Space Applications. For the space case study we use a real payload and control applications. As payload we use the OBDP benchmark. As control application we use the AOCS benchmark. Both were presented in Section 3.2.2

Automotive Benchmarks. For the evaluation of *DCmc* in another application domains we use the EEMBC Autobench suite [143] (see Section 3.2.1).

6.4.1 Intra-bank scheduler

DCmc uses a different scheduler per bank type. For the high-performance banks we FR-FCFS, which is deployed in current high-performance architectures due to

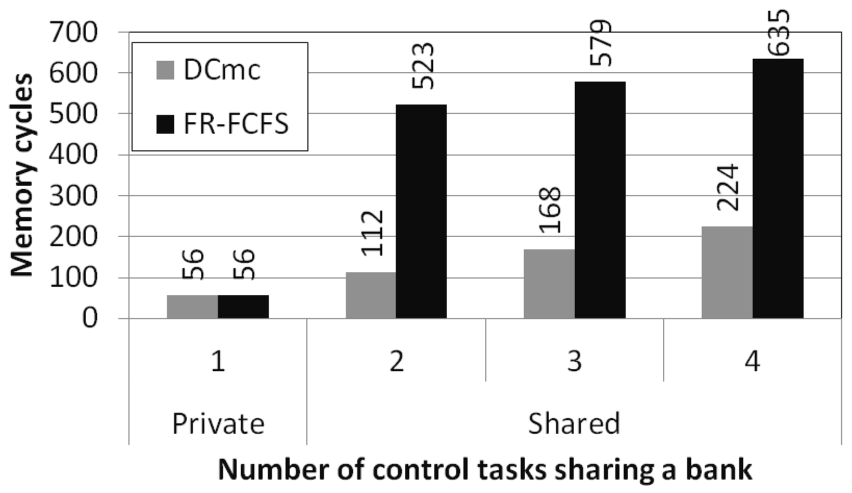


Figure 6.2: Worst-case memory latency, in memory cycles, for tasks under private bank and sharing bank schemes under different control-task count.

its high average performance benefits [150, 102].

For the real-time banks we evaluate the benefits, in terms of inter-task interference memory access bounds, of having a round-robin scheduler w.r.t. the COTS FR-FCFS controller analyzed in [102]. This evaluation is carried out under private and shared bank schemes. In this experiment, we consider a pure hard-real time system in which all tasks are real-time (i.e. there are no high-performance tasks).

The left set of bars in Figure 6.2 shows that the effect of the scheduling under private bank is the same for FR-FCFS and round-robin. However, when real-time tasks share banks, FR-FCFS produces high overestimation due to the reordering effect that can potentially introduce. In our experiment, we assume that the reordering effect is limited to 12, as shown in [102]. The round-robin scheduler reduces the effect of interferences by 3.6x on average in the scenarios with 2, 3 and 4 control tasks sharing the same banks w.r.t. FR-FCFS, making it much more convenient for real-time tasks.

6.4.2 Mixed-criticality in the Space domain

One of the main blocks in *DCmc* is the second-level scheduling which arbitrates the requests from/to the banks. The second-level scheduler prioritizes real-time banks over high-performance ones. The idea behind these priorities is removing as much as possible interference from high-performance banks over real-time banks, while enabling high-performance requests to go full-speed when there is no real-time request to be processed.

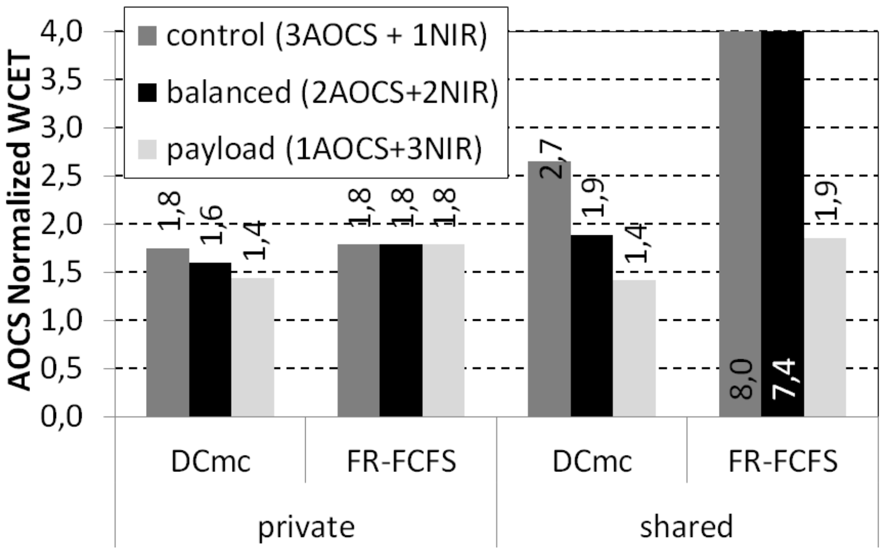


Figure 6.3: Normalized WCET under different mixed-criticality workloads under private and shared banks

We consider three different setups (workloads) for a mixed-criticality system each with a varying number of real-time and high-performance tasks. We assume a partitioned system with one real-time partition and one payload partition. Hypervisors such as Xtratum, which has been ported to LEON4 [127], have been shown to provide time and space partitioning for the space domain [22] similarly to Integrated Modular Avionics (IMA) in the avionics domain. In each setup the number of cores assigned to each partition varies from 1 to 3. We have the *control setup* with three AOCS control applications and one NIR payload application; the *balanced setup* with two AOCS and two NIR; and the *payload setup* with one AOCS and three NIR.

Figure 6.3 shows the WCET for one AOCS task in each mixed-criticality workload type (when there are several copies of AOCS we observed that all copies present exactly the same behavior in terms of WCET). Note that all WCET values are normalized w.r.t. the WCET estimate computed in isolation, i.e. assuming that only one task runs at a time.

We observe that for private bank, the WCET estimate for the control task (AOCS) under FR-FCFS is insensitive to the number of real-time (control) tasks. This is not the case of *DCmc* since it uses round-robin among control tasks, so that WCET estimates increase w.r.t the number of control tasks. However, in all cases the WCET estimates with *DCmc* improve those obtained with FR-FCFS. On the shared bank scheme, *DCmc* is much more competitive, enabling tighter WCET estimates, than FR-FCFS, which lead to very high WCET estimates when the

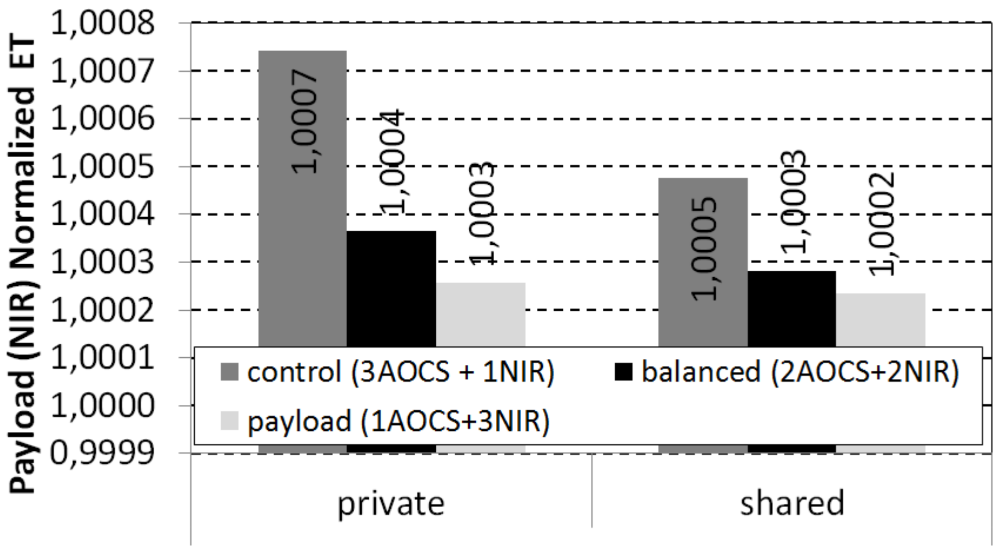


Figure 6.4: Execution time for the payload benchmark under *DCmc* running along with real-time tasks with private and shared banks

number of real-time tasks is above one. For instance, for the balanced workload, *DCmc* leads to WCET estimates $7.4/1.9 = 3.9x$ tighter than FR-FCFS.

A problem that *DCmc* may face is negatively impacting the average performance of payload applications. This occurs because real-time memory requests are prioritized over high-performance ones. Interestingly, the performance that the high-performance tasks can achieve depends on the resources left by the real-time applications. Hence, the performance of payload tasks depends on the workload considered. Figure 6.4 shows the slowdown NIR experiences under each scenario of increasing number of control tasks, w.r.t. its execution time in isolation. Each payload application has a dedicated bank, which minimizes the interference between several payload applications running at the same time. The control applications either have a dedicated bank in the private scheme or share a bank in the shared scheme. We observe that *DCmc* slightly affects payload task performance, with a small increment when the number of control applications increases. We have observed that the L2 cache efficiently filters most of the load and store operations issued by the control application, which in this case enables the payload application to almost fully enjoy the memory system.

6.4.3 Automotive benchmarks

In the next set of experiments, we use EEMBCs Autobench as control applications and run them against our payload application (NIR). In particular, for each work-

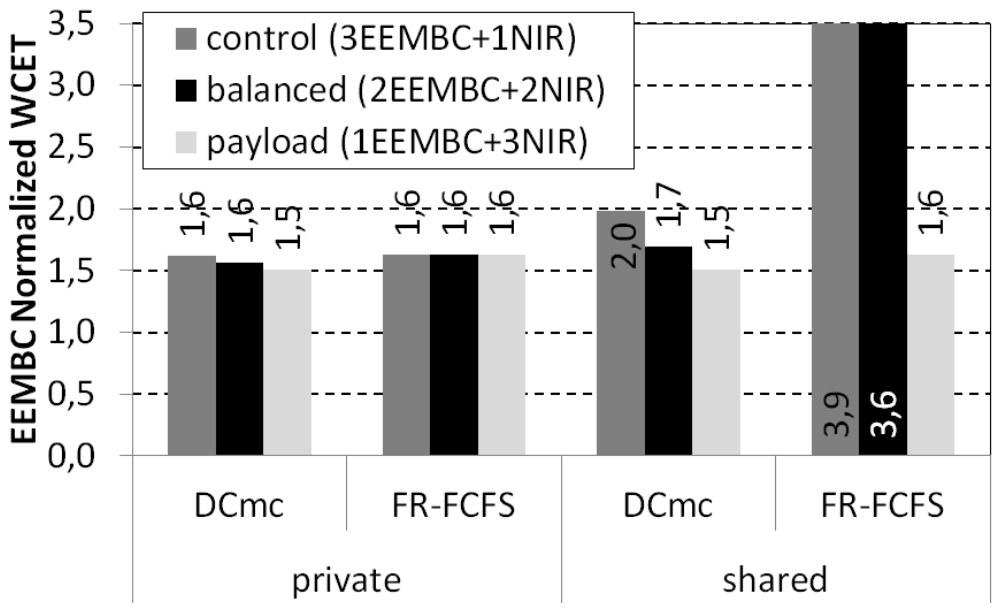


Figure 6.5: Normalized WCET under different mixed-criticality workloads under private and shared banks. EEMBC Autobench.

load type, we generate 10 workloads from randomly selected EEMBC Autobench applications in each of them.

We derive the WCET increment suffered by EEMBC Autobench benchmarks w.r.t. their WCET estimate computed in isolation, i.e. assuming that only one task runs at a time. Figure 6.5 shows the average increment across all executed benchmarks. We observe that, although the particular WCET estimates change with respect to Figure 6.3, the trends closely follow those observed with the space control application (AOCS): *DCmc* improving FR-FCFS mainly for the shared bank approach with the latter leading to high WCET overestimation.

The impact of EEMBC Autobench benchmarks on the payload application is roughly the same as the one presented in Figure 6.4 for AOCS. Payload performance depends on the resources left by control applications. We analyze this trade-off by running real-time applications in a demanding situation and measuring the performance degradation. For that purpose, we run the EEMBC applications against high-memory usage synthetic kernels as payload. The number of Memory accesses per Kilo Instruction (MpKI) of these kernels varies from 150 to 500. We measure that the performance degradation of the synthetic kernels is up to 2.5% for the 500 MpKI case and 1.8% on average.

6.5 Related Work

In this section, we discuss other existing memory controller designs for high-performance and real-time.

Real-time memory controllers were discussed in Section 2.2.2 and their most relevant characteristics are summarized in Table 6.2. High-Performance memory controllers [150, 100, 133] usually implement open-page policy to exploit row-buffer locality and so provide high memory bandwidth, i.e. once a row is open multiple requests to the same row can be performed, maximizing the memory bandwidth. In these systems, the objective of the MMS is to increase bank level parallelism and exploit row buffer locality in order to increase memory bandwidth. To that end, high-performance designs implement shared bank schemes in which blocks of sequential addresses are map into the same row together with open-page policy, and also these blocks are mapped into different banks, which allows requests from different or the same contender to access simultaneously different banks, increasing memory bandwidth. The use of open-page policy becomes fundamental in order to allow multiple memory requests to access the same row, exploiting spatial locality. The main objective of the memory scheduler is to maintain all banks occupied in order to improve the overall memory bandwidth. This is the case of the FR-FCFS [150, 100], in which the intra-bank scheduler prioritizes requests with the row already open, and the inter-bank scheduler prioritizes ready DRAM commands. Another interesting work is [133] (PAR-BS) in which authors improve the FR-FCFS algorithm using request-batching to provide fairness and freedom of starvation and also use a scheduling mechanism aware of the thread parallelism that tries to maintain the bank-level parallelism and row-buffer locality when threads are interfering across them.

Table 6.2 summarizes the row-buffer policy, MMS and memory scheduler used by the related works on memory controllers. We observe that real-time systems obtain predictable latencies by using scheduling techniques that allow to minimize the effect of interferences. On the other hand, high-performance systems try to maximize the Row Buffer Locality and the Bank Level Parallelism using complex scheduling techniques to increase bandwidth.

For dual-criticality systems, the memory controller has to provide predictability for real-time applications, i.e., bounded latencies, and bandwidth for high-performance applications. As we have seen, both goals can be opposing, especially regarding memory scheduling policies. Any solution based on a single policy that tries to cover both will end up in a trade-off between predictability and bandwidth. *DCmc* aims at bringing the best of both worlds (real-time memory controllers and high-performance memory controllers) by using different policies for real-time and high-performance applications, so that interference that the latter generates on the former is reduced, thus obtaining tight bounds on the memory latency, and

Table 6.2: List of memory controllers. RT stands for real time and HP for high performance.

Type	Ref	RBP	MMS	Scheduler
RT	[7]	Close-page	Intrlvd	CCSP
	[75]	Close-page	Intrlvd	Reconfig. TDMA
	[74]	Cons. open-page	Intrlvd	TDMA
	[139]	Close-page	Intrlvd	Round-Robin
	[148]	Close-page	Private	TDMA
	[183, 117]	Open-page	Private	FIFO
	[102]	Open-page	Private	FR-FCFS
HP	[150]	Open-page	Shared	FR-FCFS
	[133]	Open-page	Shared	PAR-BS

high bandwidth for applications that do not require bounds on the latency.

6.6 Summary

In the space domain, on-board software will comprise two criticality levels: control applications, which require real-time execution and are designed to meet requirements in the worst-case; and payload applications that are high-performance driven. These characteristics are also expected in many systems in other domains such as, for instance, automotive and avionics. Consolidating applications of both types on the same multicore hardware is of paramount importance to reduce Size, Weight and Power. Contention in the use of the memory bandwidth has a large impact on applications' execution time and WCET estimates, reducing the benefits of using multicores in the space domain. *DCmc* mitigates this effect by dividing memory banks into real-time and high-performance, providing a different request scheduler policy to each bank type. *DCmc* provides tight bounds for memory access latency of control applications regardless of the load that payload ones put on the memory controller. *DCmc* prevents real-time applications to impact high-performance ones when the former have no memory requests. Our analysis with a space case study shows that *DCmc* achieves both goals, worst-case memory access latency bounds up to 3.9x smaller than with FR-FCFS with minimum impact on the average performance of payload tasks.

Chapter 7

Contention-Free Memory System

7.1 Introduction

In high-performance domains, memory systems are usually organized into pluggable memory modules which require little space on the motherboard, ease expansion and allow replacing memory modules in case of failure. The main timing characteristic of memory modules is that they are divided into several – up to 8 in current designs – ranks that can be exploited to control contention in memory [117]. However, pluggable memory modules are not common in embedded real-time systems. For instance, in the space domain, having pluggable elements would negatively affect reliability due to the harsh conditions in which these systems are often deployed – e.g. with high vibrations and acceleration. Further, in many real-time systems, including the space domain, it is infeasible to extend or replace failing memory modules once the system is deployed, which diminishes the need for pluggable modules. As a result, memory is usually organized into different memory chips that are soldered to the board. This memory organization typically implements a single rank.

Motivation: Our proposal builds on the observation that, when memory is shared (sliced) in time, the impact of memory contention on WCET estimates is high. This occurs because in a mixed-criticality environment where tasks are likely developed by different partners and under different criticality levels, although a task enjoys exclusive access to the memory bus, when the bus is relinquished and assigned to a new task, the latter can make no assumption on the state left in memory by the previous task. For instance, the new task does not know whether the previous access was a write or a read, and since write-to-read and read-to-write timing constraints are greater than read-to-read and write-to-write, the new task has to assume that the previous task made an operation on the opposite direction (i.e., the previous task carried out a read and the new one a write or vice versa).

As a result, the new task makes – for each request – worst-case time allowances to account for the memory state left by the last task using it. This maintains time composability though increases the allocations made in the new task's WCET estimate.

Contribution: Instead of allowing that each task accesses all memory devices simultaneously to enjoy a wide data bus and time-sharing the memory system, which leads to pessimistic WCET estimates, our proposal uses narrower data buses (private to each task) each providing access to different memory devices, while the address bus is shared. With our space-sharing approach we heavily reduce the memory contention among tasks by dividing the available memory data bus to provide independent memory data buses accessing different memory chips that do not interfere each other. Dividing the data bus requires no changes on the memory device but only few extra processor pins. This approach requires more memory transfers to complete a memory transaction which reduces available per-core bandwidth when compared with wider data buses. However, this penalty is reduced since the locality of the memory row-buffers is completely exploited because the extra transfers needed are always sequentially ordered. The net result is a small reduction on average performance that pays-off for having a heavily reduced memory contention, leading to tighter WCET estimates. Tight WCET estimates are key in critical real-time systems, since they are directly related to hardware provisioning made to ensure that time critical tasks finish before their deadline.

The main properties of our memory organization are as follows. (i) It significantly reduces the impact of memory-contention on WCET estimates, which are less affected by high core counts and processor frequency in comparison to time-sharing approaches; (ii) it uses standard memory technology, i.e. commercial off-the-shelf (COTS) memory chips, deployed in other domains; (iii) it maintains time composability allowing to use single-core timing analysis tools; and (iv) it provides flexibility for mixed criticalities such that the memory can be easily configured to satisfy different real-time and performance needs.

We assess the benefits of our proposal and compare it with state of the art memory controllers both analytically and quantitatively in the context of a space case study with real space applications on a simulator validated against real boards. We focus on an on-board-soldered memory setup where chips are organized as a single rank and are accessed in parallel through a single channel. Our proposal reduces contention by 35% and 51% with respect to private and shared memory schemes respectively, having a 2% average performance penalty with respect to other memory controllers.

The rest of this chapter is organized as follows: Section 7.2 presents the motivation of our work. In Section 7.3, PDSC is proposed. Section 7.4 presents the evaluation. Sections 7.5 and 7.6 explain the related work and the summary respectively.

7.2 Motivation

Memory modules are usually plugged on the board to reduce the amount of space on the mother board dedicated for memory while easing memory expansion and replacement of failing modules. However, memories in real-time systems, as happens in many space, avionics or automotive systems, are neither designed to be upgraded nor fixed in case of failure once the system is in operation, so there is no need for interchangeable modules. Also modules and their connectors suffer mechanical stress due to severe vibration/acceleration environments, challenging their suitability for this kind of systems. As a result, in many real-time systems memory is not organized into modules but memory chips are directly soldered on the board. This setup is called *memory-down* [82] and is shown in Figure 7.1a. Furthermore, the use of multiple ranks is challenged in embedded memory-down systems since it requires extra space and weight on the board [88], critical parameters for embedded systems. For these reasons, single-rank memory-down configurations are used in the space domain which is the focus of this chapter.

In the absence of multiple ranks, we observe *bank-conflicts* and *channel-conflicts*. *Bank-conflicts* are caused by the timing constraints between commands going to the same bank, like the act-to-act constraint (t_{RC}), precharge time (t_{RP}) and by row-misses when using an *open-page* policy. *Channel-conflicts* are caused by the timing constraints between commands going to different banks in the same devices, thus, using the same channel. For instance the write-to-read (t_{WTR}) and read-to-read (t_{BURST}) constraints affecting the timing of read/write requests even if they target different banks. When using a shared-bank scheme [140], in which tasks are mapped into the same banks, bank-conflicts dominate over channel-conflicts. When using a private-bank scheme, in which tasks use their own banks, as proposed in [183, 148], bank-conflicts are removed. However, the WCET degradation is still high because of channel-conflicts. Authors in [117] identify the problem of channel-conflicts and propose a memory controller that reduces them by using multiple ranks. However, such an approach based on multiple ranks cannot be applied to many embedded systems using a memory-down setup since they often implement only one memory rank.

An effective way to remove conflicts would be by deploying a multi-channel memory system [73] in which each task uses a private channel with different memory interfaces and devices. In this case, the memory system is neither shared in time nor in space, since each task uses different interfaces and devices at the same time. However, such a solution would require hundreds of processor signals to have separate memory interfaces. For instance, the NGMP processor [42] uses a 625 pins package, out of which around 300 are usable to manage processor I/O whereas the remaining pins are devoted to power/ground, etc. *A complete multi-channel solution would require 544 pins which is completely out of reach.* Hence, for scalability reasons, a unique memory interface (channel) needs to be shared between cores due to the limited amount of processor signals. To reduce the impact of contention on shared memory interfaces, current solutions time-share it among the tasks running in the multicore.

7.2.1 Requirements

Memory-down systems face several requirements to enable their safe use in multicore systems. Although these are presented in a space-centric manner, other domains such as automotive have similar ones, and hence similar solutions apply.

Tight WCET estimates. As seen before, bank and channel conflicts have high impact on WCET estimates, specially bank-conflicts. These conflicts should be avoided in order to derive tight WCET estimates.

Factoring memory contention in WCET estimates. The memory system has to simplify factoring the impact of memory contention in the WCET estimates provided by timing analysis tools without requiring the latter to be changed.

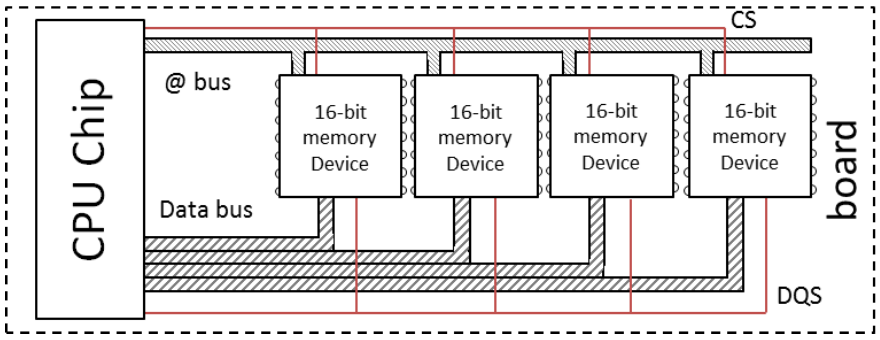
Mixed-criticaility. In the space domain, on-board systems comprise two criticality levels [142], control and payload. Control tasks have real-time constraints and require meeting tight WCET estimates. Meanwhile, payload tasks have soft or no real-time constraints, which makes them high-performance driven. The memory design has to provide *mixed-criticality flexibility*, so that it provides high average performance when no real-time execution is required and guaranteed performance for real-time tasks.

COTS technology. In general the use of COTS technology helps reducing non-recurring costs. In the space domain, while processor design can be changed to accommodate hardware support for real-time systems [40], non-customised (COTS) memory devices are preferred in order to reduce costs.

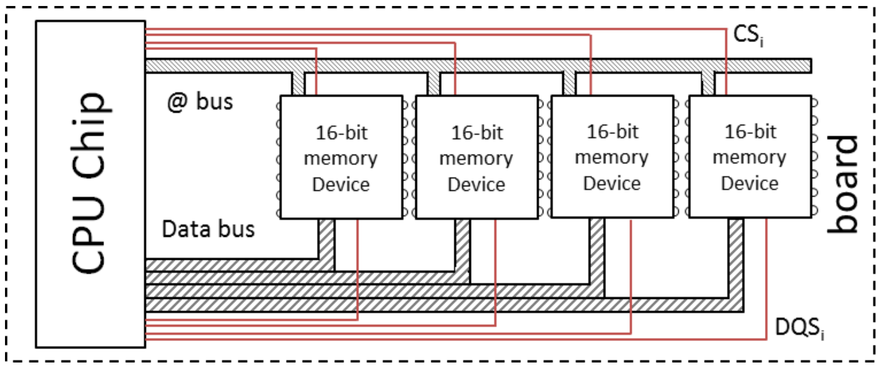
Although these requirements are specific of the space domain, other domains such as automotive and avionics have similar constraints, thus, similar solutions apply to all of them.

7.3 Private-Data bus Shared-Command bus

We propose Private-Data bus Shared-Command bus (PDSC) memory organization, which removes contention between tasks by dividing the available data bus into private independent data buses for each task, that target different memory



(a) Original setup sharing *CS* and *DQS*

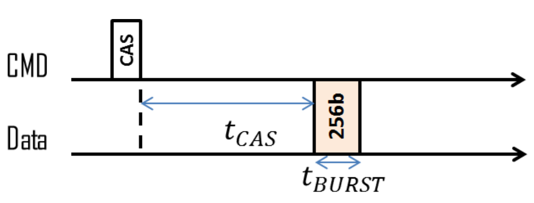


(b) Proposed setup with different *CS* and *DQS*

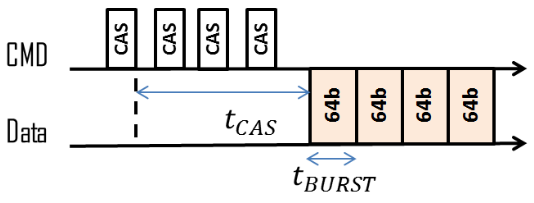
Figure 7.1: Original and proposed 64-bit memory-down system

devices, while sharing the command bus. PDSC removes contention between tasks since they are prevented from sharing the same data bus. The split of the data bus is done in a smart way such that COTS memory chips can be used. In particular, in our reference processor, comprising four cores and a 64-bit memory interface, PDSC divides the 64-bit data bus into four 16-bit buses, one for each core. The command bus is time-multiplexed for the different data buses, which means that the address and command bus is shared between cores, so that contention, or inter-task interferences, only happens on the command bus. Such contention is minimal, since only 1 bus cycle ($t_{CMD} = 1$) is needed to send commands. Hence each memory device can receive one command every four cycles.

Dividing the data bus incurs low overhead in memory-down configurations. The Chip Select (CS) and Data Query Strobe (DQS) signals indicate to a DRAM device when a command and the data are ready respectively. In the original 64-bit bus design, one CS and DQS signal is used for all the 4 devices so that they work simultaneously, as seen in Figure 7.1a. In our 16-bit design with four buses, each



(a) 64-bit data bus



(b) 16-bit data bus

Figure 7.2: Equivalent transfers.

one accessing one memory device, we only require to wire three extra CS_i and DQS_i signals, one for each extra data bus added, as presented in Figure 7.1b, which follow the same physical route as the original CS and DQS signal. These extra signals require few extra processor pins, that in our case incur a small cost since the processor we model, the NGMP [42], has about a dozen pins available. This is in contrast to a complete multichannel solution that in the NGMP would require 544 user-defined pins, hence making it infeasible.

PDSC requires more transfers to complete a memory transaction: A memory transaction brings 256 bits, which corresponds to a last-level cacheline, and requires only one transfer on the 64-bit bus since the memory has a burst of 4, i.e., $4 \times 64 = 256$ bits, as seen in Figure 7.2a. With a 16-bit bus PDSC needs four memory transfers to get the 256 bits, i.e., $4 \times (4 \times 16) = 256$ bits. For instance, with a 64-bit bus we need one transfer to get the 256-bit memory block at `0x00` (bytes `0x00` to `0x1F`). With a 16-bit bus we need four transfers, at `0x00`, `0x08`, `0x10` and `0x18` that bring 64 bits each.

Hence, for every memory transaction, PDSC performs four sequentially ordered transfers, which allows exploiting the memory locality that offers the row-buffer using an open-page policy [183]. The first transfer can be a row-hit or row-miss, depending on the previous transaction, while the following three transfers are always row-hits. Since they are always row-hits, their latency $t_{CAS} = 6$ (memory cycles) overlaps with the first transfer, thus performing those transfers immediately one after the other. Note that we rely on the fact that any cacheline is mapped consecutively on a single memory row. In this case, every extra transfer only has to consider the command bus multiplexing and the transfer time on the bus ($t_{BURST} =$ 2 cycles), since the read or write latency is overlapped between sequential transfers. An example of these accesses is shown in Figure 7.2b for a read.

The hardware cost of PDSC is similar to most tailored memory controllers [140, 183, 7], also requiring multiple queues and a command translator, which converts requests into the corresponding commands and keeps track of the open rows and refresh counters. PDSC deploys a command bus scheduler to share the command bus and a command translator to keep track of the open rows for each data bus.

Despite the use of multiple ranks is not common in memory-down setups, PDSC can also work with multiple ranks. While having ranks reduces channel conflicts [117] there are still conflicts that are caused because of the rank switching time. PDSC would allow to further remove contention, while having multiple ranks on each private data bus, which allows increasing memory bandwidth.

7.3.1 Configuration Options

PDSC can be configured to provide either increased average performance or increased guaranteed performance to respond to the asymmetric needs of mixed-criticality applications. We propose different configurations (modes) for PDSC, that use different addressing schemes and are configurable by software: *RT*, *HP* and *MIX*.

RT uses four independent 16-bit data buses to provide guaranteed performance, intended for real-time tasks. PDSC causes, in this mode, an average performance penalty for dividing the data bus as stated before.

In *HP* the average performance penalty of *RT* is reduced by interleaving the four 16-bit data buses [73]. When the four 16-bit data buses are interleaved, the four accesses needed per transaction are carried out by interleaving one access per data bus, thus using all of them at the same time. In this scenario, average performance is the same as with the original 64-bit bus, except for the time switching of the command bus that introduces a small penalty (2% in our setup).

Under the *MIX* configuration, we can have half of the memory working with independent channels, and the other half working with interleaved channels, so that tasks targeting guaranteed performance can be scheduled with tasks targeting average performance. For that purpose, two of the data buses are accessed as a private data bus each one, and the other two data buses are interleaved. The latter two data buses are accessing the same data and suffer interference.

As explained before, the memory space is divided into four equal segments, one for each data bus in the system. The data bus partitioning is done by software, similarly to the software bank partitioning [124] and requires a Memory Management Unit (MMU). The OS configures the MMU so that each core targets the appropriate memory region: real-time applications target their own data bus. For other applications the OS decides which data bus they can access by configuring the MMU.

The flexibility obtained by interleaving the data buses incurs a small hardware cost due to the complexity introduced in the data path that connects the data bus with the corresponding memory request. Further, PDSC modes can be easily configured at boot time: Each processor instance is used as part of a different system function. Each of those functions may have different high performance and real time requirements that can be accommodated with PDSC through its modes. PDSC could also be configured dynamically though this would cause data movement among memory regions.

7.3.2 Scalability

The presented PDSC memory controller fits the needs of our space case study. Processor designs [99, 137] with higher core counts usually deploy clustered architectures, which organize processor resources into "islands of execution", i.e., subsets of processors. In general, clustered architectures provide better scalability than flat architectures, higher degree of isolation across clusters, which is good for real-time, and reduced complexity. For this reason, when it comes to scaling to larger systems, we consider a clustered processor design.

We divide the available memory interface into several data buses and assign one bus per cluster of cores. Each cluster has a private data bus, so contention occurs between cores within a cluster but not across clusters. Contention between cores within a cluster can be handled by using predictable arbitration policies such as round-robin [140] and reduced by using private-bank schemes [183, 148], since these techniques are orthogonal to our proposal.

It is worth highlighting that PDSC also applies to other real-time domains, besides space, with needs for high memory mixed-criticality performance (e.g. automotive and avionics domains). PDSC, with its flexibility and scalability, would help handling the memory contention issue.

7.4 Evaluation

We use a solid evaluation setup based on a simulator validated against a real NGMP implementation (see Section 3.3). We use real space applications, control and payload [142]. As control application we use the AOCS and as payload we use the OBDP. For the evaluation in other application domains we use the EEMBC Autobench suite [143] (all benchmarks are described in Section 3.2).

We derive WCET estimates using measurements with hardware support to force requests to work on their longest latency for the on-chip bus [138]. The L2 cache has fixed access times, since it is split among cores. Single-core timing analysis tools can be used with *PDSC* since it offers composable memory access latencies. Thus, the memory has the same behavior as in a single-core system.

Worst-Case Performance. We compare PDSC with the most relevant related work: a *shared-bank* approach [140] and a *private-bank* approach [183] (details on these approaches are provided later in Section 7.5). We also consider a multichannel solution in which each core has its own 64-bit memory channel, i.e.,

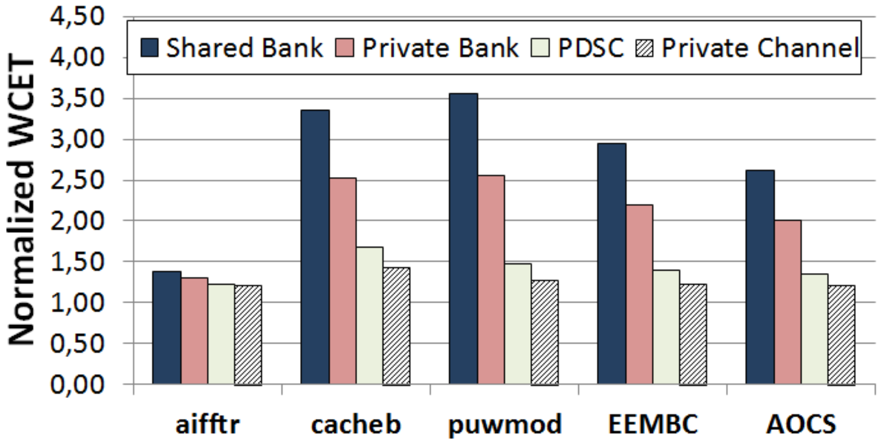


Figure 7.3: Normalized WCET for shared-bank, private-bank, private-channel and PDSC.

private-channel, thus no memory contention (i.e., bank and channel) is observed, only the bus contention is present. Despite such a solution would require hundreds of extra processor pins – posing an unaffordable cost for our reference processor – we use it as an optimal reference design since it removes contention because channels are completely independent[1].

Figure 7.3, shows the WCET normalized with respect to the WCET obtained with the original memory system, i.e., 64-bit data bus, when running in isolation without interference. These WCET are obtained in isolation for each task and take into consideration the worst-case that any workload this task runs in could generate. Results are shown for AOCS kernel, three representative EEMBC with varying cache behavior (`aifftr` with low L2 miss rate, `puwmod` with high L2 miss rate and `cacheb` with a L2 miss rate close to the average) and the average for all EEMBC benchmark suite. We observe that PDSC obtains 33% tighter WCET estimates for AOCS in comparison with a private-bank scheme and a 49% with respect to a shared-bank scheme. In comparison with the private-channel, PDSC is only 10% worse than this ideal solution. Interestingly private-channel WCET is 20% bigger than when the application runs in isolation which is caused by the contention on the on-chip bus. Similar results are obtained for EEMBC on average. The private-bank suffers a slowdown of 110% because, despite banks are private to each task (each one pinned to a different core), tasks suffer contention on the data bus – removed with PDSC.

On average, including all EEMBC benchmarks and AOCS, PDSC leads to 35% tighter WCET estimates than the private-bank scheme and 51% than the shared-

[1]Even a 16-bit multichannel solution would require more than hundred pins due to the per-core 32 bit address and control signals.

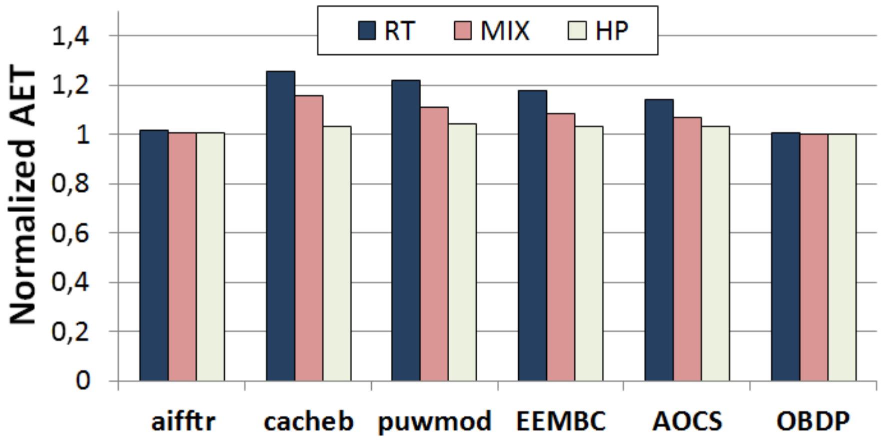


Figure 7.4: Normalized Average Execution Time (AET).

bank. Those WCET estimates are 12% higher than those with the private-channel solution, which is caused by the command bus multiplexing and the extra transfers overhead. These overheads have much less impact than bank and channel conflicts in private- and shared-bank schemes. Overall, PDSC provides tighter WCET estimates than the other approaches and quite close to the multi-channel solution, which is the tightest WCET that can be achieved in this case. Our results, not shown for space contraints, show that the impact of sharing the command bus, as PDCS does, has a negligible impact on WCET, which justifies not implementing it.

Average Performance. PDSC has two operation modes involving payload applications: MIX and HP. We compare these modes with a 64-bit data bus, i.e., the setup leading to the highest average performance. For the *MIX* and *HP* we respectively use 2 and 4 interleaved 16-bit data buses.

Figure 7.4 shows that *MIX* and *HP* have an average performance overhead of 5% and 2% respectively on average (for all EEMBCs, AOCS and OBDP) in comparison to the 64-bit bus solution. The *HP* scheme is equivalent to the 64-bit original memory system except for the time switching of the command bus that leads to a performance loss as low as 2%. For completeness we also included *RT* in Figure 7.4, although it is the mode used with real-time applications in which WCET is the main optimization factor. On average its performance degradation is 11%.

Scalability. We compare different setups in which the memory bus is split into a different number of available buses. Buses are assigned to the different cores according to two possible setups: (1) same number of buses and cores and (2) fewer buses than cores. We do not consider the case of having more buses than cores, since this is unlikely to ocurr in reality. In setup (1), each core owns a private data

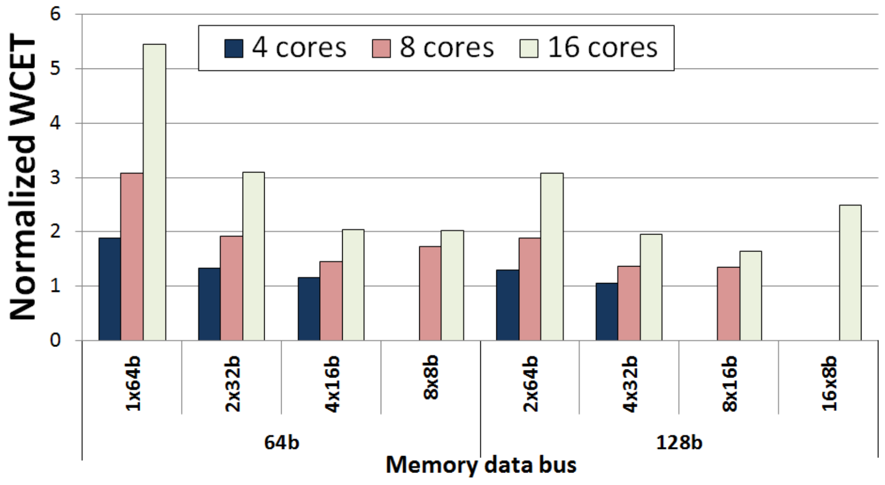


Figure 7.5: PDSC scalability results.

bus, while in setup (2), cores are organized into clusters that use a private data bus. Cores within a cluster use a private-bank scheme that reduces the interferences among them. In this case, the worst-case latency, derived as in Section 6.2, is used for each memory access. This latency can be applied either directly with static timing analysis techniques or measurement-based timing analysis techniques by means of the worst-case mode [138] in case of measurement based techniques.

For this experiment we use a crossbar interconnect, since the bus becomes a bottleneck for 16 cores masking memory impact. Hence, in this experiment, in contrast to previous ones, the increment on WCET estimates is exclusively caused by the memory contention. Figure 7.5 shows the WCET estimates, on average for all benchmarks, for 4, 8 and 16 cores normalized to the single-core WCET obtained with the original memory system, i.e., 64-bit data bus. The X-axis shows two bus setups, 64b and 128b width. The former is split into 2, 4 and 8 buses and the latter into 2, 4, 8 and 16 buses. In general terms, we see that for all core counts (4, 8 and 16) WCET estimates are largely below 4x, 8x and 16x respectively with respect to the single-core WCET. If this were not the case, the WCET results of multicores would have diminishing returns due to the memory contention. Instead we see that with PDSC, WCET shows good scalability for large core counts and wide buses. Looking at per core-count results we find the following:

– For the 4-core setup, the 4-bus configurations, i.e., 4x16b and 4x32b, are the best performing ones, since there is no interference on the data bus. Note that the 4x16b configuration is the one analyzed in detail in the chapter.

– For the 8-core setup, 8x8b buses do not improve the 4x16b configuration. This occurs because the benefit of not having interference on the data bus (since there

is a private data bus per core in the 8x8b configuration) is outweighed by the overhead of the extra transfers required. This does not happen for the 8x16b configuration because fewer extra transfers are required due to the wider data bus.
– For the 16-core case, 8x8b and 8x16b are the best performing ones for 64-bit and 128-bit interfaces respectively. In both cases the contention reduction outweighs the overhead of the extra transfers required. This occurs because of the large impact of contention in the 16-core setup. On the other hand, 16x8b configuration presents diminishing returns, due to the overhead of the extra transfers required.

7.5 Related Work

As shown in Section 2.2.2, several works use a *shared-bank* scheme in which data are mapped across all memory banks. To exploit the bank-level parallelism, access to banks is interleaved so that every request accesses all banks simultaneously in a pipelined fashion, hence removing bus conflicts among memory requests.

Other works shown in Section 2.2.2 use private-bank scheme to reduce contention by providing each core exclusive access to certain banks, effectively removing bank-conflicts. Authors in [117] further reduce write-to-read and read-to-write contention in [183] by switching between several ranks. However, such an approach based on multiple ranks is not applicable in our case, since only one rank is used.

A similar technique to the bank interleaving is used in [73] to interleave memory channels in order to exploit channel-level parallelism besides bank-level parallelism. However, as mentioned in Section 7.2, a multichannel solution is out of reach. Accessing individual DRAM devices in a memory module instead of all devices at the same time, has been proposed to improve energy efficiency with a small impact on system performance [5, 173] or storage efficiency [185]. However, contention is neither discussed nor evaluated in those works.

7.6 Summary

We have proposed a new memory organization, PDSC, for embedded real-time systems. PDSC builds on the observations that 1) in the embedded domain memory is unlikely to be pluggable because of reliability-related issues and due to the impossibility to upgrade/replace memory once the system is deployed; and 2) time-sharing memory has poor scalability with the core count and the processor-to-memory frequency ratio. PDSC uses space-sharing heavily reducing WCET, while providing enough flexibility to handle heterogeneous, i.e. high average and guaranteed performance, requirements of mixed-criticality multicore systems. Our evaluation on a solid space case study proves the benefits of PDSC to produce tight

WCET estimates, 35% tighter than with private-bank schemes and 51% tighter than with shared-bank schemes with a minimum impact on average performance in a 4 core setup. PDSC also presents good scalability with core counts.

Chapter 8

A Probabilistically Analyzable Multicore Processor

8.1 Introduction

Probabilistic Timing Analysis (PTA) [46, 32] has emerged as an alternative to conventional timing analysis. PTA provides WCET estimates with an associated probability of exceedance, called probabilistic WCET (pWCET) estimates (see Section 2.1.2). The main advantage of PTA is that it is less dependent on execution history, allowing to significantly reduce the amount of information required to obtain tight WCET estimates in comparison to other timing analysis approaches. PTA can be applied either in a static (SPTA) [32] or measurement-based (MBPTA) [46] manner. SPTA derives for each instruction a-priori probabilities of its execution time from a model of the system, whereas the second variant, MBPTA, derives those probabilities by collecting observations of end-to-end runs of an application running on the target hardware.

PTA techniques, both SPTA and MBPTA, require that the timing events under consideration, i.e. instruction execution times for SPTA and observed program execution times for MBPTA, have a distinct probability of occurrence and can be modelled with *independent and identically distributed* (i.i.d.) variables. Solutions for single-core architectures [110, 46] show how processor cores such as the Cobham Gaisler LEON3 and LEON4 [40] can be easily adapted to achieve PTA requirements. Unfortunately, to the best of our knowledge, no multicore architecture has been proven to meet PTA requirements. The main stumbling block in proving probabilistic bounds to the execution time of applications in multicores is the deterministic and history-dependent behaviour of shared resources such as bus and memory access policies, which impede memory operations to have a timing behaviour that can be modelled with i.i.d. random variables.

In this chapter, we describe for the first time a complete PTA-compliant multicore processor design. In particular, we propose new low-cost PTA-compliant bus designs that break (1) the deterministic behavior of the bus and (2) the dependence of a given application execution time's on the behavior of co-hosted applications by means of randomized arbitration policies. Analogously, following the same principles we propose a memory controller design with randomized selection of memory requests. The contributions of this chapter are as follows:

- Lottery arbitration bus: Under this bus arbitration [118], on every round the arbiter selects one core to access the bus randomly. This breaks any dependence between applications, but makes that a request may have to potentially wait an infinite number of arbitration rounds to be granted access to the bus.
- Randomized-permutation arbitration bus: We propose this new bus arbitration policy under which every N rounds, where N is the number of bus contenders, the arbiter generates a random permutation for the contenders. That sequence determines when each contender can use the bus. This arbitration provides an upper-bound to the number of rounds a request has to wait and leads to tighter pWCET estimates than the lottery arbitration bus.
- Memory: We identify the two factors that affect the execution time of requests in memory, namely, the *memory issue delay* and the *intrinsic execution time* of a memory request. While the latter can be easily upper-bounded based on the DRAM memory parameters, the former depends on the number of cores sending requests to a given memory controller and the arrival time of requests from all cores. To probabilistically model the effect of the issue delay, we apply both lottery and randomized-permutation arbitrations in the policy that selects from which core the next memory request is to be processed.

Overall, our results prove that our proposed multicore designs (i) fulfil PTA requirements by using proper independence and identical distribution tests, (ii) allow deriving pWCET estimates with *exactly* the same methods and tools as for single-core processors, thus keeping timing verification costs low, and (iii) achieve much higher guaranteed performance than single-core processors. In particular, our sensitivity analysis shows that the hardware block with the largest impact in guaranteed performance for our multicore architecture is the number of cores that contend for a memory controller. If the core count per memory controller does not exceed 4, we can achieve reasonable scalability in guaranteed performance: 2.6x average (up to 4.0x) in an 8-core architecture and 4.9x (up to 7.9x) in a 16-core architecture with respect to a single-core baseline architecture.

The rest of this chapter is organized as follows. Section 8.2 provides background on PTA and the current hardware designs for single-core PTA-compliant architectures. Section 8.3 introduces PTA for multicores. Section 8.4 presents our PTA-compliant bus designs. Similarly, Section 8.5 presents our PTA-compliant memory controller design. Section 8.6 presents the main results of this chapter. Finally, Sections 8.7 and 8.8 present the related work and the summary of this chapter.

8.2 Background

8.2.1 PTA

The probabilistic timing behavior of a program or an instruction (or in fact any component) can be represented with an Execution Time Profile (ETP). An ETP defines the different execution times of a program (or latencies of an instruction or a particular resource) and their associated probabilities. That is, an ETP represents a probability distribution function, see Equation 8.1, where p_i is the probability of the program/instruction/component to take latency l_i, with $\sum_{i=1}^{k} p_i = 1$.

$$ETP = \{\vec{l}, \vec{p}\} = \{\{l_1, l_2, ..., l_k\}, \{p_1, p_2, ..., p_k\}\} \tag{8.1}$$

The *convolution* function is used to combine ETPs, leading to a new ETP in which all possible pairs of execution times from the two ETPs are added and the probabilities multiplied. For example, let $E_1 = \{(2, 101, 200),\ (0.1, 0.4, 0.5)\}$ and $E_2 = \{(2, 101),\ (0.6, 0.4)\}$ be two ETPs. Then their convolution is:

$$E_r = \{\{2+2,\ 2+101,\ 101{+}2,\ 101{+}101,\ 200{+}2,\ 200{+}101\},$$
$$\{0.1{\times}0.6,\ 0.1{\times}0.4,\ 0.4{\times}0.6,\ 0.4{\times}0.4,\ 0.5{\times}0.6,\ 0.5{\times}0.4\}\}.$$

And given that $101 + 101 = 2 + 200$ we collapse duplicate pairs resulting in:

$$E_r = \{(4,\ 103,\ 202,\ 301),\ (0.06,\ 0.28,\ 0.46,\ 0.2)\}.$$

8.2.2 PTA requirements on hardware design

PTA techniques require that the events under analysis, program execution times for MBPTA and instruction latencies for SPTA, can be modelled with *i.i.d.* random variables [32]: two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution function.

The existence of an ETP ensures that each potential execution time of the program (for MBPTA) or instruction (for SPTA) has an actual probability of occurrence, which is a sufficient and necessary condition to achieve the desired i.i.d. execution time behavior [32]. A difference between SPTA and MBPTA, besides the level of abstraction at which ETPs are to be constructed, is that while SPTA requires ETPs for each instruction to be *determined*, MBPTA simply needs those ETPs to *exist*, but not necessarily to be known. How to use those ETPs in the context of SPTA and their meaning in the context of MBPTA is beyond the scope of this thesis. We refer the interest reader to [4] for further details on this topic.

Summary: Overall, a SPTA- and MBPTA-analyzable multicore must provide the following properties:

SPTA: SPTA requires the i.i.d. hypothesis to strictly hold at the instruction level. Hence SPTA requires deriving ETPs for each instruction and that the timing probability distribution captured by the ETP of an instruction j from a task T_k (i_j^{Tk}) is independent of the behavior of any other instruction from any other task i_l^{Tm}. If we use the core design in [110], which has been proven to be PTA-compliant, inter-core resources (e.g. bus and memory controller) have to be designed such that *each potential latency that a memory access may experience must have an associated probability of occurrence, and this probability has to be independent of the accesses generated by other tasks.*

MBPTA: The observed execution times fulfil the i.i.d. properties if observations are independent across different runs and a probability can be attached to each potential execution time. To that end, it is enough if we *make the events that may affect the execution time of a program random events whose probabilities are independent of the execution of any other program.* Hence, taking measurements from a program is equivalent to rolling a dice, with each face having a probability of appearance. Making enough rolls is enough to apply MBPTA, which derives upper-bounds of the execution time distribution by means of *Extreme Value Theory* (EVT) [115, 46]. For MBPTA, it is also enough to show the existence of ETPs for each instruction, since *the existence of the ETPs for each instruction ensures that any different execution time of the program have an associated probability and therefore MBPTA can be applied.*

ETPs, however, cannot be derived with standard (deterministic) processor architectures since events affecting execution time (e.g., bus access policy) on those architectures cannot be attached a probability of occurrence. In a multicore, ETPs cannot be obtained due to the deterministic nature of the bus arbitration policy and the deterministic processing order of requests in the memory controller. This impedes execution times of memory instructions (e.g. load/store) to be modelled with i.i.d. random variables, thus preventing the use of PTA. Hence, the focus of

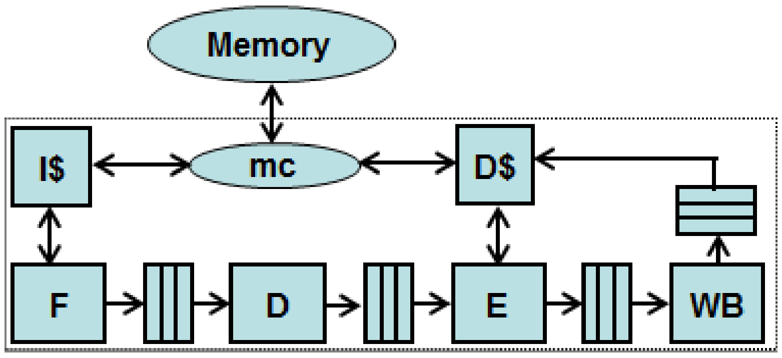


Figure 8.1: Block diagram of the PTA-compliant core architecture used in [110].

this chapter is devising bus and memory controller designs amenable to PTA.

8.2.3 Single-core Probabilistically analysable hardware designs

The basic principle to design probabilistically analyzable hardware is to control the sources of execution time variation (i.e. jitter) [32, 46]. Jitterless resources have a fixed latency, independent of the input request or of the previous history of requests accessing that resource. Jitterless resources (e.g. integer adders) are easy to model: its ETP has a single latency with probability 1. Resources with jitter, or *jittery resources* have a variable latency. Their latency depends on the execution history of the program or on the particular request sent to that resource. Jittery resources have an intrinsically variable impact on the WCET estimate for a given program. Jittery resources are either (i) enforced to always respond on their worst-case latency, so their upper-bounded timing behavior also becomes i.i.d., or (ii) redesigned so that their timing behavior depends on random events.

Following this approach [110] proposes a PTA-compliant single-core pipelined core architecture, see Figure 8.1. The architecture comprises fetch (F), decode (D), execute (E) and write-back (WB) stages. In between all stages there are latches or queues. Additionally, the WB stage comprises a write buffer in which stores are put until they are sent to cache. Loads are processed in program order in the execute stage. The accesses to the instruction and data caches happen in the fetch and execute stages respectively. Note that this pipeline design is similar to LEON3/LEON4 designs given that core operations have a fixed latency.

Data and instruction caches deploy random placement and random replacement policies [110] as needed to satisfy PTA requirements. A fixed-latency memory controller (mc) serves as the bridge between caches and memory.

8.3 PTA in multicore systems

Multicores offer better performance per watt than single-core processors while maintaining a relatively simple processor design. Moreover, multicore processors enable co-hosting applications with different requirements (e.g., high data processing demand or stringent time criticality), reducing the overall size, weight and power costs. Despite these advantages, multicores have not been widely adopted yet in embedded markets with real-time needs due to the difficulties to analyze the complex timing behavior of applications on top of multicores. These difficulties emanate from inter-task interferences when accessing shared resources. Inter-task interferences appear when two or more threads sharing a hardware resource, access it simultaneously. An arbitration mechanism determines which contending task is granted access to the shared resource, which affects the execution time and the WCET of running tasks.

Our objective is to design a multicore processor in which we can probabilistically model inter-task interferences as a way to enable probabilistic timing analysis of multicore processors with the same degree of complexity as in single-core ones. Therefore, our designs must also work with existing PTA tools [46]. Finally, we want to maintain simplicity in our hardware designs.

8.3.1 Clustered multicore architectures

Historically, clustered architectures have been considered in computer architecture. At the processor core level, execution pipelines are split into clusters (e.g. the IBM POWER7 [98]) to decrease hardware cost while efficiently exploiting instruction-level parallelism. At the chip level, processor implementations of many-core architectures (e.g. ARM Cortex A15 MPCore [12]) may group several cores into clusters as a means to reduce implementation costs. Besides, clusters enable voltage and frequency scaling as well as power-gating at the granularity of several cores, since having those mechanisms on a per-core basis has high hardware cost [132, 87].

In both clustered and non-clustered multicores the interconnection network is one of the most critical resources affecting the performance of tasks. In this chapter, we use a baseline clustered architecture as shown in Figure 8.2, where cores are equipped with private data and instruction caches. We evaluate our designs in a wide range of setups in which we vary the number of clusters and the number of memory controllers.

8.3.2 Inter-task interferences

In single-core architectures, the execution time of a task is influenced by (1) the initial processor state when the task starts its execution –which in turn is affected

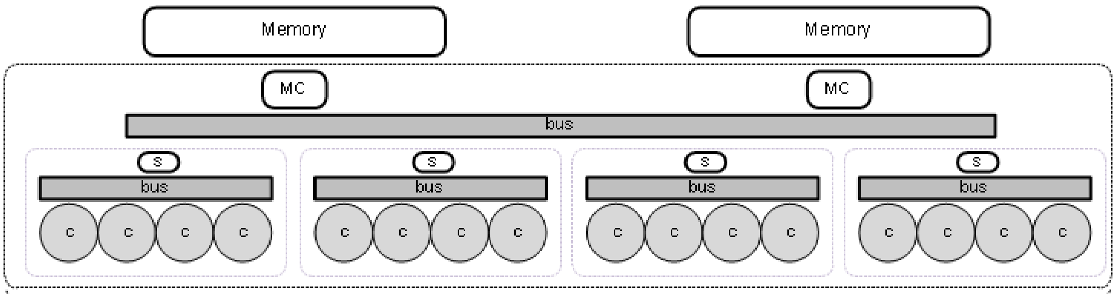


Figure 8.2: Baseline multicore architecture considered in this chapter. 'c' stands for core, 's' for switch and 'mc' for memory controller.

by previously executed tasks–, (2) the Operating System (OS) interferences, (3) the input data of the task and (4) the randomization carried out in some processor resources. The effect of initial conditions can be taken into account by flushing the state of the stateful resources (e.g., caches) prior to the execution of the task. At OS level, solutions have been shown to make the OS-induced jitter probabilistically analyzable [128], hence covering (2). The effect of input data on the execution paths followed by the program or data-dependent instructions, (3) above, is also under control by PTA techniques [32, 46]. Analogously, the jitter introduced by the randomized hardware resources is also taken into account by PTA techniques [32, 46].

In multicore architectures, in addition to all these sources of execution time variability, an additional one arises: inter-task interferences. In single-core architectures, given two instructions i_x and i_y of the same program, where x and y determine the order in which each instruction is fetched into the processor, i_y may have a potential impact on the execution time of i_x only if $y < x$, meaning that i_y executes prior to i_x. In a multicore, when several tasks run in a multicore architecture, the execution time of one instruction i_x^{T1} belonging to task T_1 may be affected by any other instruction i_y^{T2} from a task T_2. If there is precedence ordering in task execution such that T_2 executes after T_1, then the inter-task interferences generated by i_y^{T2} do not affect i_x^{T1}. If there is not precedence execution ordering, T_1 and T_2 can execute in any order. In particular, they can execute concurrently in different cores, so i_y^{T2} may introduce inter-task interferences affecting i_x^{T1}.

It becomes then evident that we cannot take into account the effect that any instruction of any task i_j^{Tk}, may have on any other instruction of any other task i_l^{Tm}. This would simply make the usage of the probabilistic approach intractable. To break this dependence we design our multicore such that we make that *the worst effect that one task can have on the execution of any other task due to inter-task interferences can be probabilistically bounded.* This makes our design *time composable*, meaning that the pWCET estimate obtained for a given task is independent of the tasks that may run concurrently in the processor.

Overall, our multicore platform is designed so that every individual processor instruction can be characterized by a distinct ETP that has no dependence on any instruction of any other task. Next, we present the proposed bus and memory controller designs, which allow deriving an ETP for each instruction independently of any other task.

8.4 On-chip interconnection network

As we have seen, in the processor core architecture proposed in Section 8.2, core operations have a fixed latency except for the access to the instruction cache (IL1). Memory operations, in addition to the execution time variation they have due to their instruction cache access, are also affected by the access to the data cache (DL1). For each access to the DL1 or IL1 we can derive a probability of hit [110]. In general, for a cache with S sets and W ways, given the sequence $< A_i, B_1, ..., B_k, A_j >$, where A_i and A_j correspond to accesses to the same cache line and no B_l (where $1 \leq l \leq k$) accesses the cache line where A_j is, the probability of A_j to hit in the cache can be approximated as [110]:

$$P_{hit_{A_j}}(S,W) = \left(1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{B_l}}}\right) \cdot \left(1 - \left(\frac{S-1}{S}\right)^{k}\right) \tag{8.2}$$

Such hit probability is used to compute the ETP of each cache access as follows where lat_{hit} and lat_{miss} are the cache hit and miss latency respectively:

$$ETP_{cache} = \{\{lat_{hit}, lat_{miss}\}, \{P_{hit_{A_j}}(S,W), 1 - P_{hit_{A_j}}(S,W)\}\} \tag{8.3}$$

On the event of a miss in the instruction or data cache, the missing instruction generates an access to the shared bus hierarchy. Next, we present several bus architectures and their corresponding probabilistic analyzes. The resulting ETP for each bus should be composed with the miss part of the ETP_{cache}. For instance, if the ETP of the bus is $ETP_{bus} = \{\{l_1, l_2, l_3\}, \{p_1, p_2, p_3\}\}$, the resulting ETP of composing cache and bus effects would be:

$$\begin{aligned} ETP_{\substack{cache\\+bus}} = & \{\{lat_{hit}, lat_{miss} + l_1, lat_{miss} + l_2, lat_{miss} + l_3\}, \\ & \{P_{hit}, (1-P_{hit}) \cdot p_1, (1-P_{hit}) \cdot p_2, (1-P_{hit}) \cdot p_3\}\} \end{aligned} \tag{8.4}$$

8.4.1 Lottery bus

We define an *arbitration round* or simply round as the number of processor cycles that a core needs to send any request to the bus. In this approach, on every round the arbiter selects one core to access the bus using a random policy. A similar bus was analyzed in [118]. However, unlike [118] we assume that all bus contenders have always pending requests, although in a given cycle, only a subset of the contenders may have pending requests. This assumption makes our design time composable, and hence independent of the traffic generated by each contender. Otherwise, ETP for memory operations of one task would depend on the traffic generated by other tasks, breaking the desired i.i.d. timing behavior.

In a N core single-bus processor, the probability of a request to be selected in round $k+1$ is given by Equation 8.5, where the first element is the probability of the request not being granted access in the first k rounds and the second element the probability of being selected in the $k+1$ round.

$$p_{lotarb}^{k} = \left(1 - \frac{1}{N}\right)^{k} \cdot \frac{1}{N} \tag{8.5}$$

Blue diamonds in Figure 8.3 show the probability of a bus request not to be granted access after k arbitration rounds for a bus shared by 4 cores. We observe that the larger the number of rounds in which a contender participates, the lower its probability not to be selected. There is a probability the contender not to be granted access after a large number of rounds. However, this probability decreases exponentially and more importantly, it is probabilistically computable.

If arbitration rounds have durations longer than 1 cycle, say L cycles, then a request may become ready in the middle of a round. In that case, the request has to wait until an arbitration round boundary before it can compete to get access to the bus.

Equation 8.6 shows the ETP for a bus access. (1) The first element convolved in the equation is the delay to align the request with the cycle at which the next round starts. A bus access request is initiated on a miss to the data or instruction cache. Given that the event 'cycle in which an access misses in the data or instruction cache' is a random event (and hence so is the cycle in which the access to the bus happens), the probability of a bus request to arrive in a particular cycle can be computed. In particular, every request to the bus may arrive in any cycle of the arbitration round $(0, ..., L-1)$ with a given probability p_{cyci} to arrive in cycle i with $0 \leq i \leq L-1$. Note that for the ETP it does not matter whether p_{cyci} follow any particular distribution as long as it is probabilistic.

(2) The second element convolved is the number of rounds that the request waits. Each round has L cycles and Equation 8.5 is used to compute the probability

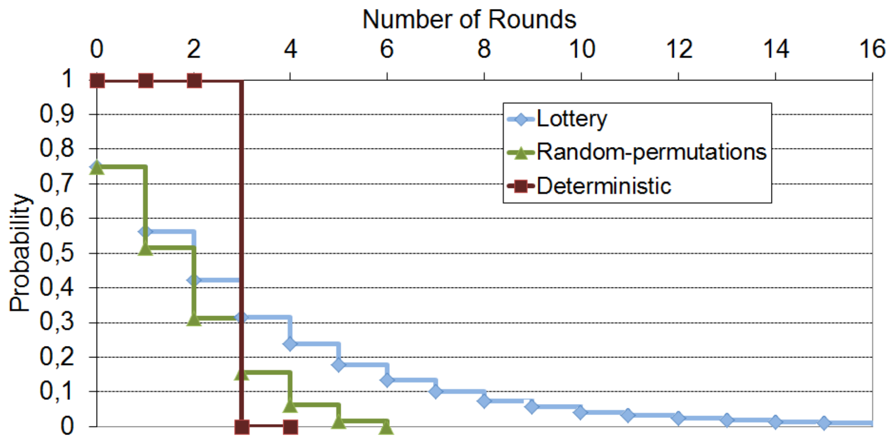


Figure 8.3: Probability of not being granted access as a function of the number of arbitration rounds for a bus with 4 cores.

of a request to be selected in a given round.

(3) Finally, the last element convolved is the actual latency of the bus access request: it takes L cycles with 100% probability.

$$\begin{aligned} ETP_{bus_{lot}} &= \{\{0,1,...,L-1\},\{p_{cyc1},p_{cyc2},...,p_{cycL-1}\}\} \otimes \\ &\quad \{\{0,L,2L,...\},\{p^{0}_{lotarb},p^{1}_{lotarb},p^{2}_{lotarb},...\}\} \otimes \\ &\quad \{\{L\},\{1\}\} \qquad (8.6) \end{aligned}$$

8.4.2 Randomized permutations

An *arbitration window* or simply window refers to N consecutive bus arbitration rounds. Under this arbitration policy, in each window one round is randomly assigned to each of the N contenders. Each round has a duration of L cycles, the maximum bus cycles that any request may take. In this approach, at every window boundary the arbiter generates a random permutation for the N contenders (cores). This sequence determines the order in which contenders can use the bus.

Analogously to the fact that the arrival cycle of a request in a round follows a given probability distribution function, there is a probability defining the round in a window in which requests arrive. This occurs since the accesses to the bus are initiated on random events: miss to the data or instruction caches. As explained before, we do not care which particular distribution function this is as long as it is probabilistic. Without lost of generality and for the purpose of this explanation we assume that the probability function describing the arrival round of each request is uniform, that is, there is a probability $\frac{1}{N}$ a request to arrive in a particular round[1]. We identify the following extreme cases:

- The shortest delay (0 cycles) occurs when a request becomes ready right when the core it belongs to gets its round.
- The worst delay occurs when (i) a request belongs to a core that gets the first round in the current permutation and (ii) the last round in the next permutation, and (iii) the first round of the current permutation has just elapsed. In this case, the request should wait $2N-2$ rounds corresponding to the $N-1$ remaining rounds of the current permutation and the first $N-1$ rounds of the next one.

The probability of a request to wait k rounds to get access to the bus, with $0 \leq k \leq 2N-2$ is as follows:

$$p^k_{perarb} = \frac{max(N-k,0)}{N^2} + \sum_{i=max(1,N-k)}^{min(N-1,2N-k-1)} \frac{i}{N^3} \tag{8.7}$$

The first addend in the equation is the probability of finding the appropriate round in the remaining part of the current permutation, whereas the second part stands for the probability of finding the appropriate round in the next permutation *if and only if* such round was not found in the current permutation. Figure 8.3 shows the probability of a bus request not to be granted access after k arbitration rounds. We observe that after $2N-2$ rounds, the probability not to be selected is 0.

The ETP of the random permutation bus, see Equation 8.8, is obtained as the convolution of three components, as for the lottery bus. The first component in the convolution is the time and associated probability to align the request to the start of the next round; the second one stands for the waiting rounds until the request is granted access, and the final one stands for the actual bus access latency of the request.

$$\begin{aligned} ETP_{bus_{per}} = \; & \{\{0,1,...,L-1\}\{p_{cyc1}, p_{cyc2}, ..., p_{cycL-1}\}\} \otimes \\ & \{\{0, L, ..., k \cdot L, ..., (2N{-}2) \cdot L\}, \\ & \{p^0_{perarb}, p^1_{perarb}, ..., p^k_{perarb}, ..., p^{2N-2}_{perarb}\}\} \otimes \\ & \{\{L\},\{1\}\} \end{aligned} \tag{8.8}$$

[1]With a similar processor setup, our results show that the arrival of bus requests across rounds in the window is uniform (see Figure 5.3 in Chapter 5).

Example of ETP computation. Let us assume a configuration with $L = 2$ cycles and $N = 2$ cores (*c0* an *c1*) and that the probability of a request arriving in any of both rounds is 0.5 ($\frac{1}{L}$). Further assume that the probability of the request to arrive in each of the 2 cycles of the round is also 0.5. In the case the request arrives in the first cycle of the round, the delay to align with a round boundary is 0. If it arrives in the second cycle of the round it takes 1 cycle to align with a round boundary.

The second component of Equation 8.8 corresponds to the number of rounds that the request will have to wait until being granted access to the bus. Every time a new arbitration window is generated (once every 4 cycles) one of the two following window setups is randomly chosen: $ws1 = (c0, c1)$ or $ws2 = (c1, c0)$. That is, *c0* first and then *c1* or *c1* first and then *c0*. For the purpose of this example, we assume that a new request from the core under consideration (e.g., *c0*) may arrive at any point in time, so it will arrive at the beginning of a window (0.5 probability) or when the first round has been elapsed (0.5 probability). In each of the cases, the probability of being under $ws1$ or $ws2$ is 0.5 for each window setup. This leads to the following 4 cases for the arriving request (from *c0*)[2]:

- It arrives at the beginning of window $ws2$. It waits 1 round with 0.25 probability since the probability of arriving at the beginning of a window is 0.5 and the probability of this window being $ws2$ is 0.5, so $0.5 \cdot 0.5 = 0.25$.

- It arrives after one round of window $ws2$. It waits 0 rounds with 0.25 probability.

- It arrives at the beginning of window $ws1$. It waits 0 rounds with 0.25 probability.

- It arrives after one round of window $ws1$. Its round in this window has elapsed and will be granted access in next window with 0.25 probability.
  - The next window is $ws1$. It waits 1 round (from the first window) with 0.125 probability ($0.25 \cdot 0.5$).
  - The next window is $ws2$. It waits 2 rounds (one from the first window and one from the second one) with 0.125 probability.

Overall, the request is delayed during 0 rounds with 0.5 probability, 1 round (2 cycles) with 0.375 probability and 2 rounds (4 cycles) with 0.125 probability.

[2]Note that any probability distribution function (PDF) is acceptable for the events determining (1) the cycle inside a round and (2) the round inside the window in which a request arrives. The only requirement is that those events are truly probabilistic. In this example we have assumed an evenly distributed PDF.

Finally, the request must also pay the fixed latency of the bus itself, L (third component in Equation 8.8). The final ETP is therefore:

$$\begin{aligned} ETP = \{\{0,1\},\{0.5,0.5\}\} \otimes \{\{0,2,4\},\{0.5,0.375,0.125\}\} \otimes \{\{2\},\{1\}\} = \\ \{\{0+0+2, 1+0+2, 0+2+2, 1+2+2, 0+4+2, 1+4+2\}, \\ \{0.5 \cdot 0.5, 0.5 \cdot 0.5, 0.5 \cdot 0.375, 0.5 \cdot 0.375, 0.5 \cdot 0.125, 0.5 \cdot 0.125\}\} = \\ \{\{2,3,4,5,6,7\},\{0.25,0.25,0.1875,0.1875,0.0625,0.0625\}\}. \end{aligned}$$

8.4.3 Deterministic bus

Alternatively to time randomized buses, a deterministic bus deploying a predictable policy, such as round-robin policy could be used. This requires a non work-conserving approach in which a task is assumed always to suffer the worst latency when accessing the bus [138]. In the case of a round-robin policy, the worst scenario implies assuming that whenever a particular core has to access the bus, its round has just elapsed and has to wait for $N-1$ rounds where N stands for the number of cores. This is safe yet pessimistic to be assumed at analysis time. At deployment time such a constraint can be removed since the number of actual rounds that requests will have to wait will never exceed $N-1$. As a result, the ETP still remains as a safe upper-bound at deployment time, but average performance is improved. The ETP for this deterministic bus policy is show in Equation 8.9. It is analogous to that of randomized buses except for the second term, which is now fixed: $N-1$ rounds of L cycles each with 100% probability.

$$\begin{aligned} ETP_{bus_{det}} = & \{\{0,1,...,L-1\},\{p_{cyc1},p_{cyc2},...,p_{cycL-1}\}\} \otimes \\ & \{\{(N-1)\cdot L\},\{1\}\} \otimes \\ & \{\{L\},\{1\}\} \end{aligned} \tag{8.9}$$

Figure 8.3 shows the probability of a bus request not to be granted access after a number of arbitration rounds for this arbitration policy (dark red squares) in a multicore setup with 4 cores. We can see that a particular core is never granted access at rounds 0, 1 and 2, and it is granted access at round 3.

8.4.4 Hierarchical buses

Based on the ETPs derived for simple buses we can derive the ETP for a hierarchical bus network. In our bus setup, on the one hand, we have an intra-cluster bus ("ibus") per cluster. The ibus is accessed by N_{co} contenders (cores) and has an access latency of L_i cycles. Each cluster has a switch that connects the intra-cluster bus to the inter-cluster bus ("ebus"). This simple switch adds a fixed latency to the end-to-end latency of the on-chip interconnection network, S. Finally, the ebus is connected to each of the N_{cl} clusters and has a latency L_e. Usually, $L_e \geq L_i$ as the ebus is longer than the ibus as the ebus connects distant clusters.

These three resources of the hierarchical bus are accessed serially: ibus, switch and ebus. The ETPs of each of those resources can be easily composed as shown in Equation 8.10. Note that the bus ETPs are characterized by parameters L, latency, and N, number of contenders, as described Sections 8.4.1 and 8.4.2.

$$ETP_{hbus} = ETP_{ibus} \otimes ETP_{switch} \otimes ETP_{ebus} \tag{8.10}$$

$$\begin{aligned} ETP_{ibus} &= ETP(L_i, N_{co})_{bus} \\ ETP_{switch} &= \{\{S\},\{1\}\} \\ ETP_{ebus} &= ETP(L_e, N_{cl})_{bus} \end{aligned}$$

8.5 Design of a Probabilistically Analyzable Memory Controller

In this chapter, we focus on DDRx SDRAM off-chip memory system which consists of a *memory controller* and one or more *memory devices* (see Section 2.2.2). The timing behavior of a memory request is characterized by the *Request Inter-Task Delay* or t_{ItD}, and the *Request Execution Time* or t_{RET}. The former stands for the delay the request can suffer – due to interferences with other requests generated by co-running tasks running on different cores – before being granted access to the memory device. The latter stands for the time a memory request takes to be completed, once it cannot suffer interferences from the other threads' requests.

While t_{RET} delay is fixed (jitterless), and hence taking it into account in the ETP for the memory is trivial, t_{ItD} is jittery and depends on:

- Number of scheduling-rounds: The channel to access the memory device is shared across all tasks sharing the memory controller so they may interfere each other. Among the different memory requests in the ready queues, the memory scheduler is in charge of scheduling the next request. The arbitration policy implemented in the memory controller and the structure of the internal queues used to buffer the requests inside the controller determine how many scheduling-rounds a particular request must wait to be granted access to the shared memory.

- Duration of each scheduling-round (*Issue Delay*): The Issue Delay is the time interval between the issue of two consecutive requests, i.e. from the instant a request is issued until the next one can be issued. The Issue Delay depends on the DRAM device used, on the specific timing constraints, on the *Row-Buffer Management policy* and the *Address Mapping scheme* implemented in the memory controller [92].

Our goal is to probabilistically model the *Request Inter-Task Delay* (t_{ItD}), so that by taking it into account during the pWCET analysis of a task, the pWCET estimates are independent of the rest of the co-running tasks, hence enabling time composability.

8.5.1 Probabilistically Modelling Inter-Task Delay

The effect of inter-task interferences on average performance has made that high-performance processors come equipped nowadays with several memory controllers (or multi-channel memory controllers). For instance, the ORACLE UltraSparc T2 features 2 memory controllers to control two memory channel, the IBM POWER7+ up to 8 memory channels and the Freescale P4080 includes 2 memory controllers for 8 cores (4 cores per memory controller).

Our memory controller has several request queues (q queues). In particular, in our design, we assume *one request queue per core*. When we have several memory controllers, a given core accesses only one of the memory controllers. For instance, if we have 16 cores and two memory controllers, each memory controller has 8 request queues ($q = 8$ for each memory controller). Note that the higher the number of cores mapped onto a memory controller, the higher the effect of inter-task interferences on both, pWCET estimates and average performance.

In each request queue, memory requests are kept in order. The memory scheduler uses round robin, lottery and random-permutation policies to select the request queue that is granted access to access memory (i.e. send DRAM commands), as for the bus.

In [139], authors derived an Upper Bound for the Issue Delay, the Longest Issue Delay (t_{LID}), that takes into account the Row-Buffer Management policy and the Address Mapping scheme. Once a memory request is ready, a hardware mechanism controls that the DRAM commands are not sent to memory until t_{LID} cycles have elapsed since the last memory request was granted access. Once a request is granted access to memory, it takes t_{RET} cycles to complete, and the next request will be granted access t_{LID} cycles after the previous request was issued.

Hence, the maximum memory turn-around time and the ETP for a memory request can be expressed as $t_{mem} = f(t_{LID}, policy) + t_{RET}$. The first addend, which upper-bounds t_{ItD}, covers the delay to select the next request from those in the request queues and depends on the policy (i.e. lottery, random permutation or round-robin). The second addend is the intrinsic execution time of the memory request (i.e. the sequence of DRAM commands generated to serve the memory request).

The ETP for memory requests is analogous to that for bus requests, but replacing L by t_{LID} in the first two components and by t_{RET} in the third component. For instance, Equation 8.11 shows the ETP for memory requests when we use the lottery arbitration. It has three components: (1) The first component convoluted is the delay to align the request with the cycle at which the next round starts. In this case rounds last t_{LID} cycles. Again, without loss of generality we assume an even probability distribution function such that there is a probability $\frac{1}{t_{LID}}$ a request to arrive in a given cycle of the round. (2) The second component stands for the number of rounds that the request may have to wait. Each round lasts t_{LID} cycles and Equation 8.5 is used to compute the probability of a request to be selected in a given round. (3) Finally, the last component is the actual latency of the memory access (t_{RET} cycles with 100% probability). For random-permutation and round-robin policies the process is analogous, just changing the part regarding the arbitration policy (e.g., using Equation 8.7 instead of 8.5 for random-permutation).

$$\begin{aligned} ETP_{mem} = & \{\{0, 1, ..., t_{LID} - 1\}, \{\frac{1}{t_{LID}}, \frac{1}{t_{LID}}, ..., \frac{1}{t_{LID}}\}\} \otimes \\ & \{\{0, t_{LID}, 2 \cdot t_{LID}, ...\}, \{p^0_{lotarb}, p^1_{lotarb}, p^2_{lotarb}, ...\}\} \otimes \\ & \{\{t_{RET}\}, \{1\}\} \end{aligned} \tag{8.11}$$

8.5.2 Memory refreshes

Data must be periodically read out and restored to the full voltage level with refresh operations (*REF* commands) in SDRAM memories for data integrity. Every t_{REFI} cycles a refresh command (REF) is automatically sent to all banks, refreshing one row per bank. This operation takes t_{RFC} cycles to be completed. Upper-bounding the number of refresh operations that can occur during the execution of a task (taking its pWCET) implies considering both the pWCET without refreshes ($pWCET_{NOREF}$) and the time devoted to refresh operations ($N_{REF} \cdot t_{RFC}$, where N_{REF} stands for the maximum number of refresh operations occurring during the execution of the task). N_{REF} is given by the minimum number of refreshes, N_i that accomplishes with the following inequation.

$$\left\lceil \frac{pWCET_{NOREF} + N_i \cdot t_{RFC}}{t_{REFI}} \right\rceil \leq N_i \tag{8.12}$$

In order to account for the effect of refreshes in the pWCET estimates, we add the delay suffered due to memory refreshes on top of the estimated pWCET. Thus, the new pWCET is computed as: $pWCET_{TOTAL} = pWCET_{NOREF} + N_{REF} \cdot t_{RFC}$. Note that some refreshes could have occurred when obtaining measurements for MBPTA which adds some pessimism since $pWCET_{NOREF}$ would already include the effect of some refreshes. However, it may be the case that refreshes never impacted measurements because they occurred during idle rounds in memory. Thus, we cannot discount the effect of those potential refresh operations from the pWCET obtained.

8.6 Evaluation

We use a similar simulation framework to the one presented in Section 3.3 to model the probabilistically analysable processor. Cores are as presented in Section 8.2: 4-stage pipelined cores with a memory hierarchy composed of separated instruction and data caches. The size of each cache is 4-KB with 64-byte line size and 4-way associativity. The latency of the fetch stage depends on whether the access hits or misses in the instruction cache: a hit has 1-cycle latency and a miss has variable latency to access to memory. After the decode stage, memory operations access the data cache so they can last 1 cycle or a variable latency to access memory in case of a miss. The remaining operations have a fixed execution latency (e.g. integer additions take 1 cycle).

We use different multicore setups featuring the hierarchical bus architecture presented in Section 8.4, varying the number of cores from 4 to 16, and the number of memory controllers (MC). In this study, we have fixed the number of cores per cluster to 4. However, the trends shown in this section are also observed when changing the number of cores per cluster. We have used the following setups: 4 Cores with 1 MC setup (4C1MC); 8 Cores with 1 MC (8C1MC) and 2 MC (8C2MC); and 16 Cores with 2 MC (16C2MC) and 4 MC (16C4MC).

In this study, we have focused on a close-page, interleaved-bank, DDR2-800E memory setup, because it is the one providing higher flexibility (in terms of address mapping scheme) and lower t_{LID}, although all other designs are also compliant with our PTA multicore processor. We assume a CPU frequency of 800MHz, being a CPU-SDRAM clock ratio of 2. Note that our pWCET estimates provided include memory refresh delays.

We use the EEMBC Autobench benchmarks suite [143] (see Section 3.2.1). We used: *a2time*, *aifftr*, *aifirf*, *aiifft*, *cacheb*, *canrdr*, *iirflt*, *puwmod*, *rspeed*, *tblook* and *ttsprk*.

8.6.1 Actual measurements vs. models

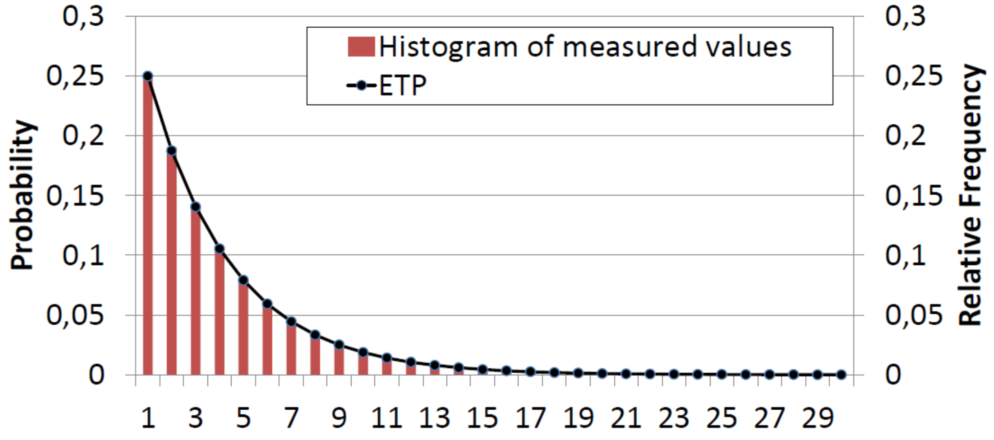
In this section, we show how the analytical ETPs derived for the components match the timing behavior observed for those components. In particular, we focus on the lottery-arbitration bus and the random-permutations arbitration bus.

Figure 8.4 shows the observed frequencies (bars that are measured in the secondary y-axis) in the bus access time and a graphical representation of the ETP (lines that are measured in the primary y-axis) for a configuration with $N = 4$ contenders and $L = 1$ round size. In both cases, lottery and randomized-permutations, the observed behavior matches the analytical model of Equations 8.6 and 8.8 (note that Figure 8.3 shows their CCDF instead of the ETPs as in Figure 8.4). Such match occurs because the arrival time of bus accesses is evenly distributed across cycles in the round and across rounds in the window. This occurs because (1) cache misses generating bus accesses are random events and (2) the latency of the instructions executed between consecutive bus accesses is large enough with respect to the window size of randomized-permutations such that two bus requests from the same core are rarely sent during the same window. In particular, we observed no meaningful deviation with respect to an even distribution and, if such deviation existed, it would have no impact on the approach as stated in Section 8.4.2.

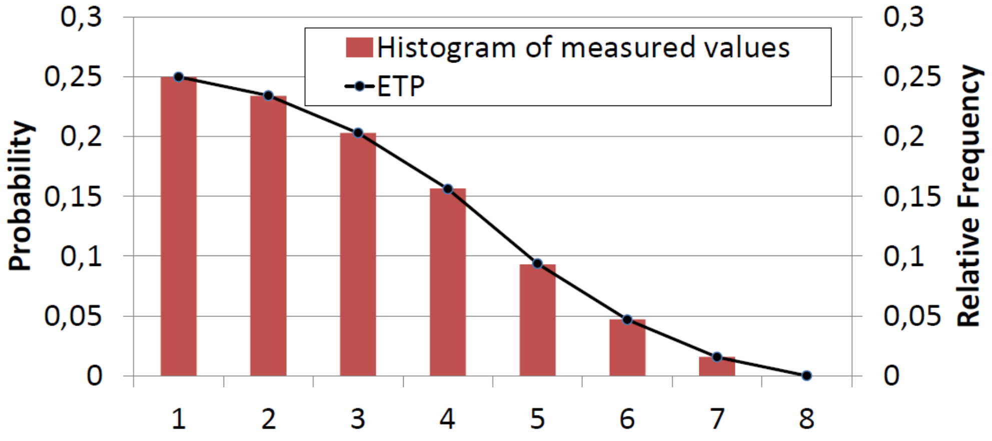
8.6.2 Hardware overhead

Lottery bus: The lottery arbitration simply requires $\lceil \log_2 N \rceil$ bits to select which contender is granted access in each round. Given that the number of contenders, N, is typically a power-of-two, using *exactly* $\log_2 N$ bits produced by a pseudo-random number generator (PRNG) is enough to select the particular contender that is granted access. Note that efficient PRNGs already exist in real processors implementing, for instance, random-replacement policies in cache [13, 38].

Random permutations: Such arbitration can be also implemented with very low cost. A *randperm* register with $N \cdot \log_2 N$ bits is needed. This register keeps the N identifiers of $\log_2 N$ bits for each of the contenders. In order to generate a random permutation, $N - 1$ random bits, called *randbits* generated by a PRNG are needed. To that end we swap the identifiers in the *randperm* register in a hierarchical way based on the values of *randbits*. For instance, if $N = 4$, we need 3 random bits. The first bit in *randbits* determines whether the first and second identifiers in *randperm* are swapped (1) or not (0). The second random bit determines whether the third and fourth identifiers in *randperm* are swapped. The third random bit determines whether the first pair of identifiers is swapped with the second pair. If the current state of *randperm* is ‘00-01-10-11’ (so contenders order is 0, 1, 2, 3) and *randbits* is ‘101’, the new permutation will be ‘10-11-01-00’ (the first and second ids were swapped, and the first and second pair were



(a) lottery bus



(b) randomised-permutation bus

Figure 8.4: Measured execution times vs. ETPs.

also swapped), so the new contenders order will be 2, 3, 1, 0. It can be seen that the probability of a particular contender to reach any particular position in the permutation is exactly $\frac{1}{N}$ regardless of its position in the previous permutation since $\log_2 N$ random bits will determine its new position. It can be also seen that all permutations cannot be generated. For instance, in the example before contenders 2 and 3 cannot be in different halves of the permutation. However, this is irrelevant because any particular contender occupies each position in the permutation with the same probability and the order of contenders in the remaining positions has no effect on the current contender. For instance, in the example before it is irrelevant for contender 3 whether the contender in the first position is 0, 1 or 2.

Benchmarks	**4C1MC**	**8C1MC**	**8C2MC**	**16C4MC**
a2time	0.25/0.99	0.57/0.67	1.80/0.31	0.82/0.61
aifftr	1.13/0.11	0.60/0.55	0.53/0.29	0.13/0.95
aifirf	0.53/0.72	0.93/0.67	0.33/0.93	1.73/0.23
aiifft	1.27/0.33	0.93/0.51	1.40/0.77	0.80/0.90
cacheb	0.87/0.15	1.47/0.82	0.60/0.82	1.07/0.29
canrdr	1.60/0.61	0.73/0.96	1.60/0.90	0.27/0.61
iirflt	1.53/0.77	0.60/0.96	1.20/0.17	0.47/0.29
puwmod	0.60/0.93	0.40/0.51	1.07/0.96	1.13/0.41
rspeed	0.26/0.37	1.47/0.86	1.53/0.19	0.27/0.17
tblook	0.40/0.67	0.80/0.82	0.47/0.86	1.00/0.23
ttsprk	0.53/0.37	0.40/0.29	0.13/0.96	0.20/0.56

Table 8.1: Independence and identical distribution tests results (outcome independence test / outcome i.d test).

8.6.3 Fulfilling the i.i.d properties

Our bus and memory controller designs guarantee that observed execution times fulfil the properties required by MBPTA given that we are able to derive ETPs. We contrast this empirically by analyzing whether execution times of EEMBC benchmarks are independent and identically distributed.

In order to test independence we use the Wald-Wolfowitz independence test [27]. We use a 5% significance level (a typical value for this type of tests), which means that absolute values obtained with this test must be below 1.96 to prove independence. For identical distribution, we use the two-sample Kolmogorov-Smirnov identical distribution test [24]. For 5% significance, the outcome provided by the test should be above the threshold (0.05) to indicate identical distribution.

Table 8.1 shows the results of both tests for all EEMBC benchmarks under several multicore setups implementing random-permutation arbitration, when running each benchmark 1,000 times. Both tests are passed in all cases. The results for the configurations not shown in the table also passed both tests.

8.6.4 MBPTA: EVT projections

In this section, we provide pWCET estimates obtained with the MBPTA method in [46] for EEMBC benchmarks under several multicore setups. In all setups we deploy random-permutation arbitration in buses and the memory controller, since, as it will be shown in Section 8.6.5, it outperforms the other policies. Note that MBPTA has been used so far *only* on top of single-core architectures. Therefore,

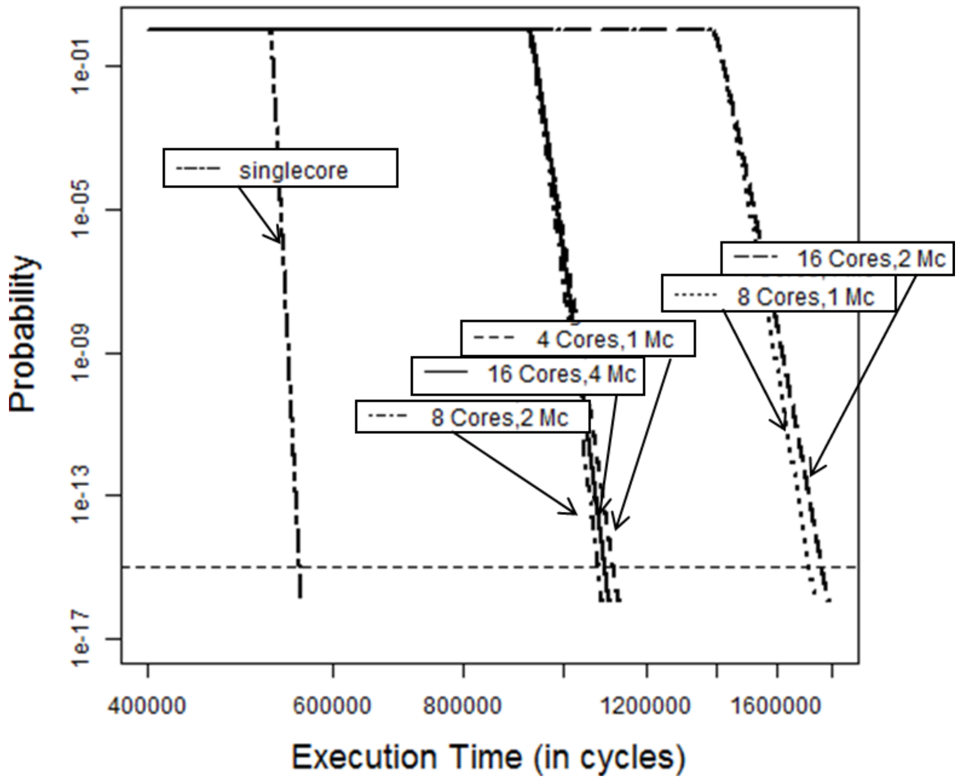


Figure 8.5: pWCET estimates for `canrdr` under different multicore setups.

this chapter provides the first multicore designs (i) amenable for PTA and (ii) analyzable with the same tools as single-core designs.

Following the iterative method of [46] we carried out 1,000 experiments and used Extreme-Value Theory (EVT) to extract pWCET estimates. As an example, Figure 8.5 shows the EVT projections for the `canrdr` benchmark. We consider an exceedance probability of 10^{-15} per run. Our selection of the exceedance probability, i.e. the probability that an instance of a task misses its deadline, is based on the observation that for the aerospace commercial industry at the highest integrity level (DAL-A) the maximum allowed failure rate in a piece of software is 10^{-9} per hour of operation [154]. In current implementations, the highest frequency at which a task can be released is 20 milliseconds (180,000 times per hour) [154]. Hence, the highest allowed failure rate per task activation is 5.56×10^{-15}, which is largely above our exceedance probability.

In Figure 8.5 we observe that the main factor affecting pWCET estimates is the number of cores per memory controller and not the total number of cores. In the baseline setup, with only one core in the architecture, `canrdr` does not suffer any inter-task delay in the memory controller ($t_{LID} = 0$). In the set of pWCET

benchmark	4C1MC	8C1MC	8C2MC	16C2MC	16C4MC
a2time	3.8	8.0	3.4	8.0	3.5
aifftr	3.8	7.6	3.7	7.2	4.0
aifirf	3.1	5.7	2.6	6.8	2.6
aiifft	3.9	7.3	3.9	7.6	3.9
cacheb	4.5	9.0	4.6	9.0	4.2
canrdr	1.9	3.0	1.9	3.1	1.9
iirflt	4.1	8.4	4.7	7.7	4.9
puwmod	1.8	2.8	1.8	2.7	1.8
rspeed	1.9	3.0	1.8	3.0	1.9
tblook	3.3	6.5	3.7	6.6	3.0
ttsprk	2.4	3.9	2.4	3.9	2.4

Table 8.2: pWCET estimate for 10^{-15} for each EEMBC under all setups w.r.t their pWCET estimate in single-core.

projections in the middle we have 4 cores per memory controller, which increases t_{LID}. Note that *our design makes the pWCET estimate for a task* (`canrdr` in this case) *to be independent on the particular tasks running concurrently.* Finally, the pWCET estimates on the right of Figure 8.5, are for the setups with 8 cores sharing each memory controller. Table 8.2 summarises the results for all benchmarks. In particular, it shows pWCET estimates for each EEMBC under the different multicore setups in which we have 4 and 8 cores contending in each memory controller with respect to the pWCET estimate obtained when each benchmark runs in a single-core setup.

For the configurations in which there are 8 cores sharing a memory controller (i.e. 8C1MC and 16C2MC), we observe that for some few benchmarks their pWCET increases more than 8x, meaning that there are diminishing returns in the use of multicores. For the rest of benchmarks we appreciate that the increment in pWCET is much smaller than 8x, hence providing benefits. For the configurations in which there are 4 cores sharing a memory controller (i.e. 4C1MC, 8C2MC and 16C4MC), the benefits are much higher. In some cases, a benchmark suffer an increment in its pWCET of less than 2x in its pWCET estimate (when in a 8-core or 16-core configuration) with respect to its pWCET estimate in single core. As seen on the Freescale P4080, 4 cores per memory controller is a usual choice for embedded processors, and our results confirm this choice for PTA-compliant processors.

8.6.5 Comparison of arbitration policies

In this section, we compare random-permutation, lottery and deterministic (round-robin) arbitration in terms of reduction in pWCET estimates. Section 8.4.2 shows that the ETP derived for the random-permutation arbitration is better than for the lottery and deterministic arbitration, i.e. the area below the random-permutation curve in Figure 8.3 is smaller than for lottery and deterministic arbitrations.

Lottery and deterministic arbitration need $N-1$ rounds on average to grant access to the shared resources, where N stands for the number of contenders. Conversely, random-permutation arbitration needs around $\frac{N}{2}$ rounds. For instance, for 4, 8 and 16 contenders lottery and deterministic arbitration need 3, 7 and 15 rounds on average, whereas random-permutation needs 1.8, 4.2 and 8.8 rounds on average.

This translates into lower pWCET estimates for the random-permutation arbitration. Figure 8.6 confirms that random-permutation improves over the other two policies. We observe that pWCET reductions range between 9% and 16% w.r.t. lottery arbitration and between 5% and 11% w.r.t. deterministic arbitration. pWCET reductions w.r.t. lottery arbitration are higher because its impact on average delay is similar to that of deterministic delay, but lottery arbitration introduces more variability (recall that with lottery arbitration there is a probability contenders not being granted access after a large number of rounds). Hence, although the best case of lottery arbitration is better than that of deterministic arbitration, its worst case is worse than that of deterministic arbitration, thus leading to worse pWCET estimates.

8.6.6 Guaranteed performance scalability analysis

The ratio $\frac{executed\ instructions}{pWCET}$ (where pWCET is expressed in cycles) gives us the *guaranteed performance* or guaranteed Instructions Per Cycle (IPC) that an application can achieve for a given probability threshold.

We built 11 workloads, each consisting of as many copies of the same EEMBC benchmark as cores in the multicore setup under study. Note that for any particular program it is irrelevant which programs are run in the other cores since pWCET estimates always consider that *all* contenders are ready to access the particular shared resource (bus or memory controller) regardless of whether this is the case in reality. Figure 8.7 shows the addition of the guaranteed performance of all benchmarks in each workload. We observe a wide range of different behaviors. Some benchmarks access memory frequently, and hence few copies of them are enough to saturate memory. As a result, adding more benchmarks in the architecture does not improve the guaranteed performance. This is the case, for instance, of `a2time`. Other benchmarks make a moderate use of the memory and hence

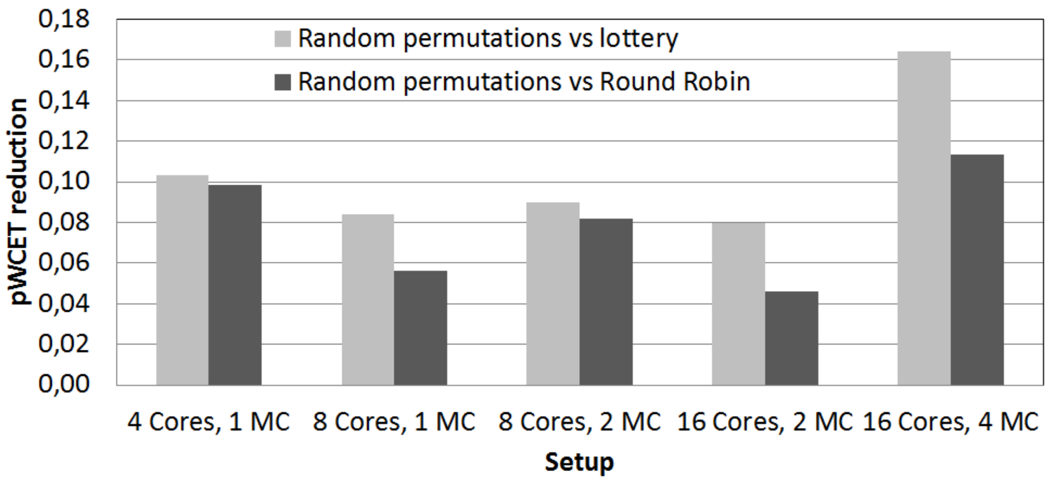


Figure 8.6: pWCET reduction when using randomized permutation arbitration in the buses and in the memory controller with respect to using lottery and deterministic arbitration respectively.

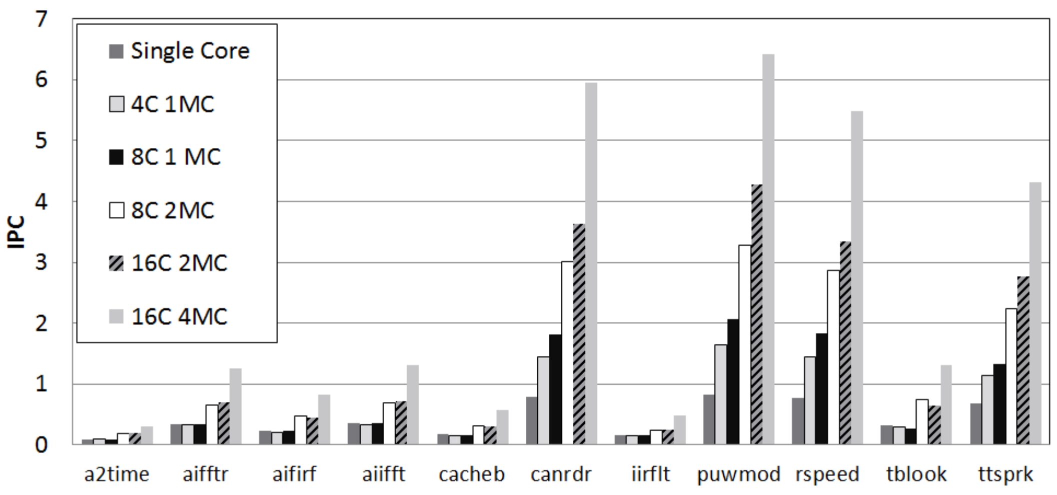


Figure 8.7: Guaranteed IPC (Instructions Per Cycle) scalability

scale much better with the number of cores, e.g. `puwmod`.

We average the increase obtained in guaranteed performance in each multicore setup (with respect to the single-core pWCET) across all the 11 workloads obtaining increases in guaranteed performance of 1.3x, 1.4x, 2.6x, 2.9x and 4.9x for the 4C1MC, 8C1MC, 8C2MC, 16C2MC and 16C4MC setups respectively.

8.7 Related Work

As presented in Section 2.2, there have been several proposals to enable the use of multicores in real-time systems. Several of these techniques deal with the communication bus or with the memory controller. A commonality of all these proposals is that they are meant to work in conjunction with deterministic timing analysis techniques and not with probabilistic timing analysis as it is the focus of this chapter (see Section 2.1).

Regarding probabilistic timing analysis, as seen in Section 2.1.2, several techniques [32, 46] have been proposed to cover PTA's static and measurement-based variants respectively. At hardware level, a PTA-compliant single-core architecture is presented in [110]. However, to the best of our knowledge, no multicore PTA-compliant architecture has been presented so far.

8.8 Summary

PTA enables affordable analysis of complex hardware in safety-critical real-time systems by reducing the amount of information about the hardware and software state required to provide trustworthy WCET estimates. Yet, PTA relies on some properties that existing multicore processors fail to provide. In particular PTA requires that the execution times of the program on the target platform can be modelled with i.i.d. random variables.

As multicores become the *de facto* processor architecture for CRTES given their simple design and ability to co-hosting tasks with different requirements, enabling their use in conjunction with emergent timing analysis techniques, such as PTA, is a must.

We have described for the first time a PTA-compliant multicore design. We propose new low-cost PTA-compliant bus designs that (1) break the deterministic behavior of the bus and (2) the dependence of a given task execution time on the behavior of co-hosted tasks by means of randomized arbitration policies. Analogously, following the same principles, we have proposed a memory controller design with randomized selection of requests. Our results prove that the proposed designs (i) fulfil PTA requirements, (ii) can be analyzed with existing PTA tools for single-core processors and (iii) provide high guaranteed performance.

Chapter 9

Contention-Aware Performance Monitoring Support

9.1 Introduction

As part of the validation and verification process of CRTES functional and temporal requirements need to be assessed to obtain enough evidence about the proper operation of the CRTES. Testing, which is an integral part of the validation and verification process, is intended to find both temporal violations (i.e. a function overruns its assigned time budget) and functional bugs (i.e. a given function does not perform its work). The absence of errors during the testing phase increases the confidence had on the system correct behavior. In this chapter we focus on the temporal validation and verification of CRTES, which requires deriving execution time bounds for software units – also referred to as WCET estimates. The most common method used to obtain those bounds is measurement-based timing analysis [180]. In processor architectures with limited complexity the challenge lies in finding a set of (program) inputs that lead to the WCET. Tools have been developed and qualified to help on this task. For instance, Rapita's Verification Suite [147] can be used to test the achieved execution-path coverage with the user provided inputs. Of course, the level of rigour required varies depending on the criticality defined for the function under analysis (e.g. modified condition/decision coverage is required for DAL A functions under DO-178C [153]).

In recent years CRTES are witnessing an unstoppable transition towards MultiProcessors System-on-Chip (MPSoC) – featuring cache memories and multicores – to respond to the increased performance requirements of CRTES in domains such as avionics, space and automotive. This, in turn, is required for CRTES to cope with more sophisticated value-added software functionalities [178]. MPSoC promised benefits come at the cost of complicating CRTES temporal verification.

In particular the contention between tasks in MPSoCs hardware shared resources has been acknowledged as one of the most complex elements for temporal analysis [3, 135]. This occurs because the load a given task puts on hardware shared resources affects its co-runners and vice versa.

In particular, the estimated bounds for the Worst Contention Delay (WCD) in hardware shared resources are exposed to several inaccuracies that decrease the confidence had on them [63]. In this scenario, confidence can only be regained via proper testing in the validation and verification phase. Whenever a task overruns it is key determining whether the overrun is caused by the contention on the hardware shared resources or it is due to the application intrinsic (in isolation) behavior. In the former case, one would want further information on the resource where the contention delay is taking place and which of its contenders is causing it. This would be very valuable information for validation, verification – acknowledged as time and effort consuming steps – and optimization purposes.

This chapter makes the case for the Actual Contention Delay (ACD) metric and the Contention Cycle Stack (CCS) approach to improve the temporal-related testing of MPSoCs. ACD captures the time tasks spend stalled in a shared resource due to contention with their co-runner tasks. The CCS is a stacked representation of a task's ACD to understand the particular contention time the task spends in each shared resource. For each such resource the CCS also allows determining the contending task causing the contention. The use of CCS brings benefits at different abstraction levels, both during the integration tests and once the system is deployed. This includes (i) determining whether a task overruns due to (unnoticed) systematic hardware failures or software faults; or (ii) the overrun is caused by inaccuracy of the WCD used by the timing analysis tool; and (iii) optimizing energy usage and average performance by scheduling tasks such that ACD is reduced. Overall, our contributions in this chapter can be summarized as follows:

1. We make an in-depth analysis of the Performance Monitoring Counters (PMC) provided in several processor chips targeting high-performance and real-time domains including designs by IBM, Intel, Freescale, ARM and Gaisler. Rather than providing an exhaustive list of PMCs, we focus on the events related with contention that PMCs can track. Our analysis reveals that, despite some of the studied architectures having hundreds of PMCs, they are not meant to track ACD, which prevents deriving the CCS in a cost-effective manner.

2. With focus on the GR740 processor – which implements the latest NGMP – we propose a new set of low-overhead CCS-aware PMCs for the two main shared resources in the GR740: the AHB AMBA bus and the memory controller (note that the L2 cache is partitioned using the hardware support provided by the GR740, so tasks suffer no contention delay in this resource).

3. We evaluate our proposal in two solid setups. First, a highly accurate GR740 simulator (see Section 3.3). And second, the CCS-aware PMCs for the AMBA bus are implemented in a FPGA with the GR740. For both setups, our results show that CCS-aware PMCs effectively capture the contention delay suffered by each task which we present with CCS. We also show that our proposal incurs low hardware cost since it reuses the available PMC infrastructure in the GR740, which is multiplexed for different event counts, adding only 16 new events and its corresponding wiring on the FPGA.

Overall, CCS increases the confidence on the bounds of tasks' WCD, and hence the CCS becomes an instrumental means for the testing of CRTES – arguably the only way to consider increasing dynamic contention in MPSoCs. In case of a task overrun, CCS allows to ascertain whether this is due to an inaccuracy in the model or it is due to other systematic behaviors intrinsic to the task or the hardware. Finally, CCS also enables other scheduling optimizations.

The rest of this chapter is organized as follows: Section 9.2 presents the problem attacked and the expected results. Sections 9.3 presents a taxonomy of the PMC in the architectures studied in this chapter based on the detailed analysis done in Section 9.6. Section 9.4 shows our proposal of CCS aware PMCs. Section 9.5 evaluates our proposal. Section 9.7 presents the most relevant related work. Finally, Section 9.8 presents the summary of this chapter.

9.2 Introduction to and Applicability of The Contention Cycle Stack

During the analysis phase of the system, bounds are derived to the WCD that a task can suffer. At deployment (operation) tasks suffer delays due to contention, which we call actual contention delay (ACD). It is worth nothing that, while several works focus on deriving bounds to WCD, understanding the ACD that tasks suffer has been barely studied in the literature. In this chapter we show the benefits that deriving ACD brings for validation/verification and optimization purposes.

9.2.1 Introduction to the CCS

Figure 9.1 shows a synthetic example of the CCS of a given task[1], *Task1*, derived for a 4-core processor with a shared bus and memory (the L2 is split per core), when *Task1* runs in a workload with *Task2*, *Task3* and *Task4*. Under each CCS

[1]In this chapter, for the sake of simplicity, we consider a correspondence between task and core, i.e., *TaskX* runs in CoreX.

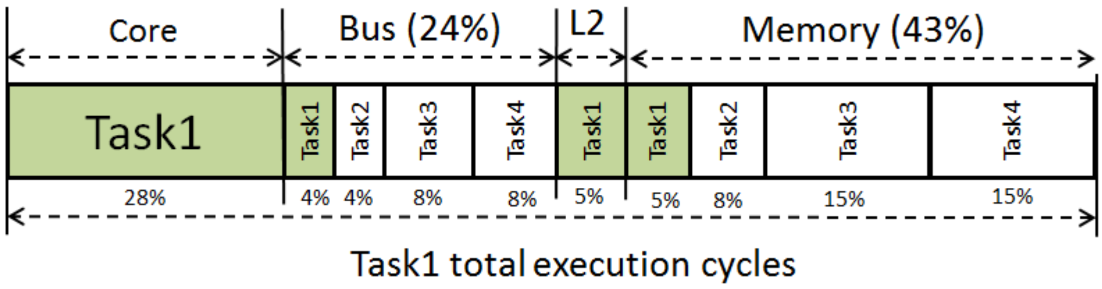


Figure 9.1: Synthetic CCS for a given Task1 on a quad-core. Note that we usually represent the CCS vertically, though for space constraints this figure presents it horizontally.

component we stack[2] all the cycles that *Task1* spends on a given resource. The *core component* covers the time *Task1* spends executing locally in the core (28% in the example).

For the off-core resources, the CCS breaks down per task the time in each component. For instance, the bus component covers the cycles, called *working cycles*, in which the processor is stalled due to the processing of a request in the bus, labelled as *Task1*; and the cycles in which *Task1* is waiting to get access to the bus while Task2, Task3 and Task4 are using it, called *contention cycles*. In the figure, we observe that Task1 is stalled during 24% of its overall execution time due to the bus: 4% of the cycles to handle its own requests, while 4% waiting for Task2, 8% for Task3 and 8% for Task 4. Likewise, the CCS also provides information about Task1's working and contention cycles in the L2 cache and memory controller. The shaded elements in each component represent Task1's working cycles, i.e. its behavior in isolation, while the other elements provide information about the time Task1 spends stalled in each shared resource due to contention caused by each contending task.

9.2.2 Applicability

Temporal Validation and Verification. Deriving bounds to the contention delay in MPSoCs, either by building timing models to feed static timing analysis tools or using measurements, is challenging. In both cases the complexity of MPSoCs affects the confidence one can put on the derived bounds.

For timing models, one could infer the contention that requests accessing a shared resource cause on each other from the reference documentation. However, manuals are becoming multi-thousand documents so extracting timing informa-

[2]Cycles in each component do not occur consecutively during the execution of the task. In the CCS we *stack* those cycles (and show them) consecutively.

tion is an error-prone process. As a matter of example, some parts of the Freescale P4080 specification have 2,000 pages [71] and the Infineon XMC4500 microcontroller documentation has more than 2,500 pages [85]. Even if the chip vendor explicitly provides this information in the manuals, it is the case that MPSoCs documentation can be inaccurate or outdated with respect to the deployed chip implementation [3]. For instance, the FreeScale e500mc core documentation has already reached the third revision with details about non-negligible changes across revisions [72]. Similarly, the documentation of processors such as the ARM Cortex R5 – specifically targeting CRTES – have abundant errata [72, 16] despite being relatively simple. All these difficulties have made that real-time industry and static timing analysis tool providers use measurement-based approaches to derive contention bounds [135].

With the goal of deriving bounds to WCD, several measurement-based techniques exist that use specialized kernels, called resource stressing kernels (*rsk*) [65], which aim at putting high, ideally the highest, load on shared resources. Those *rsk* are used to put a given task under analysis under stressful contention scenarios. When there are several shared resources, it is virtually impossible to design a *rsk* that puts the highest load on all of them simultaneously [144]. Hence, it is hard – if at all possible – to determine from the execution time whether, under a given workload, a task hits the WCD in one shared resource and not in another or whether the WCD was not hit for any of the resources.

The CCS allows determining the ACD a task suffers per resource (and per task) which enables verifying and validating contention bounds by determining how close is the ACD to the theoretical WCD. The CCS can be used in those scenarios to provide evidence about the validity of contention time bounds – increasingly derived with measurements [135, 63] – such that the contention suffered by the task in the workload matches the worst possible latency it can suffer [138]. Also, in case a bound is exceeded, the CCS can detect it and identify the reason behind it. For instance, assuming that Task 1 in Figure 9.1 had an overrun due to Tasks 3 and 4 memory contention, the CCS allows identifying this situation by providing the information that memory contention is 38%, having Tasks 3 and 4 a 15% of impact each. CCS also provides the execution time in isolation just by removing the ACD. For instance, if Tasks 2, 3 and 4 are removed from the CCS in Figure 9.1, what remains is the execution time of Task 1 when no other core is running.

Optimization. The CCS provides valuable information that can be used for optimization purposes. For instance, the CCS allows identifying, in the example in Figure 9.1 that Task 1 spends 43% of its execution time in memory, 30% of which is due to interference from Tasks 3 and 4. In such a case system designers can either reduce memory accesses from Task 1 by improving cache locality of accesses or prevent Tasks 3 and 4 from being scheduled together with Task 1.

Table 9.1: Analysis of whether each processor contains PMC in each of the categories identified in our taxonomy (Detailed description provided in Section 9.6).

	Cycle Count			Event Count				Data Count	Current Events
	Use	Busy	Idle	Use	Busy	Thld	InsT		
IBM	✓	✓	✓	✓	✓	✗	✓	✓	✗
Intel	✓	✓	✓	✓	✓	✓	✓	✗	✓
ARM	✓	✓	✗	✓	✓	✗	✓	✗	✗
Freescale	✓	✓	✗	✓	✓	✓	✓	✗	✗
GR740	✓	✓	✓	✓	✓	✓	✓	✗	✗

Such optimizations bring the advantage of having more slack in the schedule, so that other tasks can also be consolidated onto the same hardware platform, or some energy can be saved.

To obtain the CCS we need detailed information of the processor events affecting its timing, which is obtained through its PMC. In the next section, we analyze those PMC available in some MPSoCs useful for measuring contention.

9.3 Taxonomy of PMCs in Real Multi-Core Processors

We have analyzed in detail the available information about the PMCs of the IBM POWER7, the latest Intel architectures, the ARMv7-A, the Freescale P4080 and the GR740. Our goals are identifying the type of events that can be tracked with those counters and whether this can be used to build the CCS, rather than presenting an extensive list of all counters of the analyzed architectures – that in several cases exceed 500. It is worth noting that, since our focus is on contention interference effects, we analyze the trackable events on hardware shared resources. From this perspective, the events happening inside the computing cores private resources (e.g. pipeline and local caches) are of no interest.

From our analysis we have produced a PMC taxonomy and have derived some conclusions on PMC that hold across all platforms. For readability reasons, we defer a detail explanation of the PMC support in each architecture to Section 9.6 and we focus on this section on the result of the analysis, i.e. the taxonomy which is shown in Table 9.1.

Cycle count. This category covers those counters that measure time (cycles) . This is further broken down into the following. *Use*: number of cycles that a resource is in use; *Busy*: number of cycles that a resource is unavailable and causes a core stall. And *Idle*: number of cycles that a resource remains unused.

Event count category groups those counters that measure the number of times an event happens. This category is further divided into the following. *Use*: number of times a resource has been accessed; *Busy*: number of times a resource has been unavailable because it was busy; *Threshold*: number of times a set threshold has been exceeded; And *Instructions by type*: number of times an instruction of a given type is executed.

Data count covers the counters that measure the amount of data transferred or managed.

Current events groups those counters that provide the current status of the processor.

We find few counters that help understanding to some extent the effect of contention interferences:

POWER7: The PMC PMC-CMPLU-STALL-THRD provides the number of cycles a task is stalled due to inter-task interferences in the reorder buffer (a.k.a. as Global Completion Table). No information on contention in core-external shared resources is provided.

Intel: The MEM-TRANS-RETIRED.LOAD-LATENCY counter can be configured to track the number of times memory load operations latencies exceed a user defined threshold. This allows the user to approximate the worst observed behavior, and to upper bound it. However, such counter neither measures actual contention nor identifies the reasons behind such contention.

GR740: PMCs can be configured in *maximum count mode.* While in this mode, the counter keeps the maximum amount of time the selected event has been asserted. It is also possible in this mode to count the maximum amount of time between two event assertions. Maximum count mode might be useful as a first step to implement CCS PMCs. Using Maximum Count Mode it could be possible, for example, to count the longest burst of bus cycles, or the longest amount of time the bus has been without having a read access. As in the case of Intel and IBM counters, these counters are neither designed to measure actual contention nor to identify the cause of such contention.

In all the studied architectures, PMCs are used to improve average system performance by monitoring software execution, characterizing processor behavior, and/or helping system developers bring up and debug their systems, but their focus is not monitoring contention or interferences across tasks, so they do not help deriving the CCS. We note that some information about contention interference can be derived in controlled scenarios by means of experimentation. For instance, in a first experiment the program under study is run in isolation recording *cycle count* PMC readings. In a subsequent set of experiments the program under study is run again as part of a workload. By subtracting the PMCs in the first run from those in the second run some inter-task interference information can be obtained, though likely it will not be precise on which shared resource and which task caused it. Furthermore, this process is complex due to reproducibility issues and, in some cases, because the system cannot be used to carry out those extra runs. For instance, if a deadline is missed in a fail-safe system (1) it may be difficult – if at all possible – to reproduce the scenario that led to the deadline miss and (2) the system cannot be used to run some experiments because it would jeopardize its availability given that its operation needs to be quickly resumed.

9.4 PMC proposal for obtaining CCS

The CCS provides a representation of the working and contention cycles of each task (τ_i) running in a multicore. Tasks spend some processing cycles at core level[3], p_i, and some others accessing core-external hardware shared resources ($\mathcal{R}$) that cause cores to stall and consume cycles, s_i. For each shared resource $r \in \mathcal{R}$, stall cycles are broken down into working cycles, w_i^r, and contention cycles, c_i^r. The former corresponds to the cycles τ_i spent actually using the resource, as it happens when running in isolation, i.e. without contention. The latter covers the cycles in which τ_i was stalled due to some inter-task (contention) interference activity generated by another contending task τ_j. In our reference architecture, the CCS can be expressed as shown in Equation 9.1:

$$t_i = p_i + s_i = p_i + \sum_{r \in \mathcal{R}} (w_i^r + c_i^r) \tag{9.1}$$

Let $c(\tau_i)$ be the (contending) tasks executing at a given point in time with a given task τ_i. The number of tasks in $c(\tau_i)$ varies from 0 to $Nc-1$, i.e. the number of cores minus one. For τ_i, its CCS for each resource c_i^r, can be further broken down so that it provides information about the contention each of its contenting tasks cause on τ_i, called $c_{i\leftarrow j}^r$.

$$c_i^r = \sum_{\tau_j \in c(\tau_i)} c_{i\leftarrow j}^r \tag{9.2}$$

By combining Equation 9.2 and Equation 9.1, the CCS can be expressed as shown in Equation 9.3:

[3]We include in this category also cycles in which the pipeline is stalled (no instruction can be fetched) due to a local stall, for instance a floating-point operation blocking the processor due to its long latency.

$$t_i = p_i + \sum_{r \in \mathcal{R}} \left(w_i^r + \sum_{\tau_j \in c(\tau_i)} c_{i \leftarrow j}^r \right) \tag{9.3}$$

In this chapter we explore the cost and benefits of the CCS in the Cobham Gaisler GR740.

9.4.1 Cobham Gaisler GR740

The GR740 is equivalent to the NGMP described in Section 3.1. The GR740 implements hardware mechanisms to partition the L2 cache so that each core can only access its assigned cache ways. Hence, there is no contention once a request arrives at the L2 and all cycles spent in the L2 are working cycles. If the request misses in L2, it accesses memory. The memory controller behaves as a FIFO queue, with the request on top accessing memory, i.e., consuming working cycles, and the other requests in the queue waiting, i.e., consuming contention cycles.

9.4.2 CCS for the GR740

The GR740 comprises three main on-chip hardware shared resources, the memory, L2 cache and the bus, so $\mathcal{R} = \{mem, L2, bus\}$. Building the CCS for the GR740 requires deriving the *processing cycles* (p_i) for the core where the task under analysis is, *working cycles* (w_i^r) for each shared resource and *contention cycles* ($c_{i \leftarrow j}^r$) for each contending task in the bus and the memory. The L2 cache is a special case because it is partitioned, which means that contention on the access to it is removed, i.e., $c_{i \leftarrow j}^{L2} = 0$ for all τ_i, τ_j.

9.4.3 Processing and stall cycles

We deploy existing PMCs in the GR740 to derive p_i. In particular, we use the *execution cycles*, t_i, and the *stall cycles*, s_i to compute processing cycles as $p_i = t_i - s_i$. Execution cycles t_i are obtained from existing PMC (processor event *time*:`0x15` [39]). When deriving s_i it is worth noting that the stall cycles are obtained as a combination of three existing PMCs since the cycles that the processor is stalled, s_i, are caused by the i)IL1, ii)DL1 or iii)the write-buffer when they wait for a request to be completed outside of the core. These three events can be directly measured from their respective available PMCs (processor events *ichold*:`0x02`, *dchold*:`0x0A` and *wbhold*:`0x10` [39]) and their addition gives s_i.

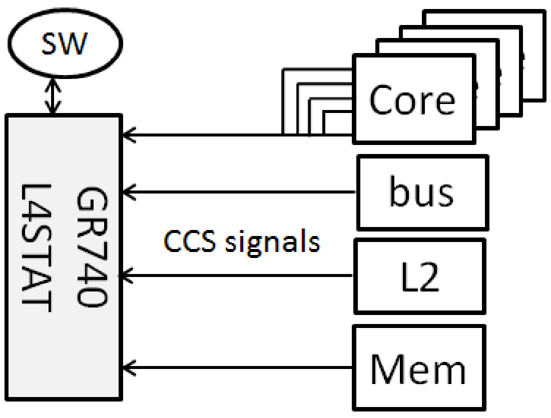


Figure 9.2: CCS module and input/output signals.

9.4.4 Working and contention cycles

Our extension to the GR740 PMC infrastructure consists in adding new low-overhead PMCs that allow accounting for w_i^r and $c_{i \leftarrow j}^r$ for the bus and the memory. L2 working cycles, w_i^{L2}, can be obtained indirectly from available PMCs. Since hit and miss latencies are known, and available in the documentation [39], an estimate of the time spent in the L2 cache can be derived using these latencies and the number of hits and misses, which can be directly measured from their respective PMCs (events l2hits:`0x60` and l2miss:`0x61` [39]). If L2 latencies were not available in the documentation, L2 working cycles could be easily obtained as the remaining execution time cycles, since all the cycles, except L2 cycles, can be accurately classified as either core, bus or memory cycles. Note that all L2 cycles are working cycles, because there is no contention in the L2.

For the bus and the memory, working (and contention) cycles cannot be indirectly obtained from available PMCs. For instance, the GR740 provides PMCs to count the number of bus accesses, however, since the latency per access is not fixed – and in some cases depends on other components [94] – *working* and *contention cycles* cannot be derived.

We propose extending the available processor's statistics unit, called L4STAT in the GR740, with new events needed by the CCS, called CCS module (CCSm). Figure 9.2, sketches the CCSm and how it is connected to other hardware blocks in the GR740. Each shared resource provides two pieces of information to the CCSm on every cycle: the core that is using the resource, which allows tracking the working cycles, and the cores that are waiting for the resource, which allows tracking if a core is interfered and in that case, the actual core using the resource is designated as the interferer.

It is worth noting that certain resources inside the core, such as the write-buffer, can hide the latency of some requests. This makes that even when there are outstanding requests in the bus or in memory, the processor core is not necessarily stalled, since the write-buffer can hide that latency. The CCSm requires identifying the cycles that the processor is stalled to know if it should account cycles in shared resources, either as *working* or *contention* cycles. Otherwise there would be more accounted cycles than cycles spent in reality, thus reducing the accuracy of the CCS.[4] For instance, if the write-buffer hides the latency of a write request, even if that request is using the bus, the core is working, thus consuming processing cycles and those cycles should not be accounted for the bus. For that purpose, the core should provide a *stalled* signal to the CCSm, indicating whether the core is stalled waiting for a request or not. As mentioned at the beginning of this section, the *stalled* signal can be easily obtained as a combination of the three *trigger signals* for the PMCs that measure the cycles that IL1, DL1 or the write-buffer are waiting for a request, which correspond to the stall cycles of the core, as seen before.

Although in theory some events could overlap, and so some cycles could be accounted twice, this occurs seldom in practice. In particular, a core may have two requests in flight simultaneously (one in the bus and another in the memory controller). However, this can only happen if the first request is a write operation – so it does not stall the core – and the second one a read operation. In this case, since the write operation cannot stall the execution, any stall due to contention or access delay is accounted to the second request, since it is the only one that can impact execution time. Still, our results show that the frequency of occurrence of that combination of events is negligible in practice.

9.4.5 Shared AMBA AHB bus

One of the most important hardware shared resources in the GR740 (and in many other MPSoC for real-time systems) is the backbone bus, since it connects the different cores with the memory/cache subsystem (and possibly other devices or subsystems). The In an AMBA bus (as explained in Chapter 4), a single AMBA request from a given task can block other tasks' accesses to the bus for long periods. This reinforces the need to have the CCS for the bus. In Chapter 4 we also describe the high cost that changing the AMBA interface would incur, specially regarding compatibility and development/usage of third-party intellectual-property cores. *Therefore, our PMC support for the AHB AMBA bus must not require any change in its interface.*

Under the AMBA protocol, the arbitration process involves several hardware blocks (the arbiter and one or several masters) and several signals. We focus on two

[4]This also stands for the counters that count L2 hit/miss, counting only when the processor is stalled. Otherwise, original PMCs would account hits/misses for write requests that do not stall the processor.

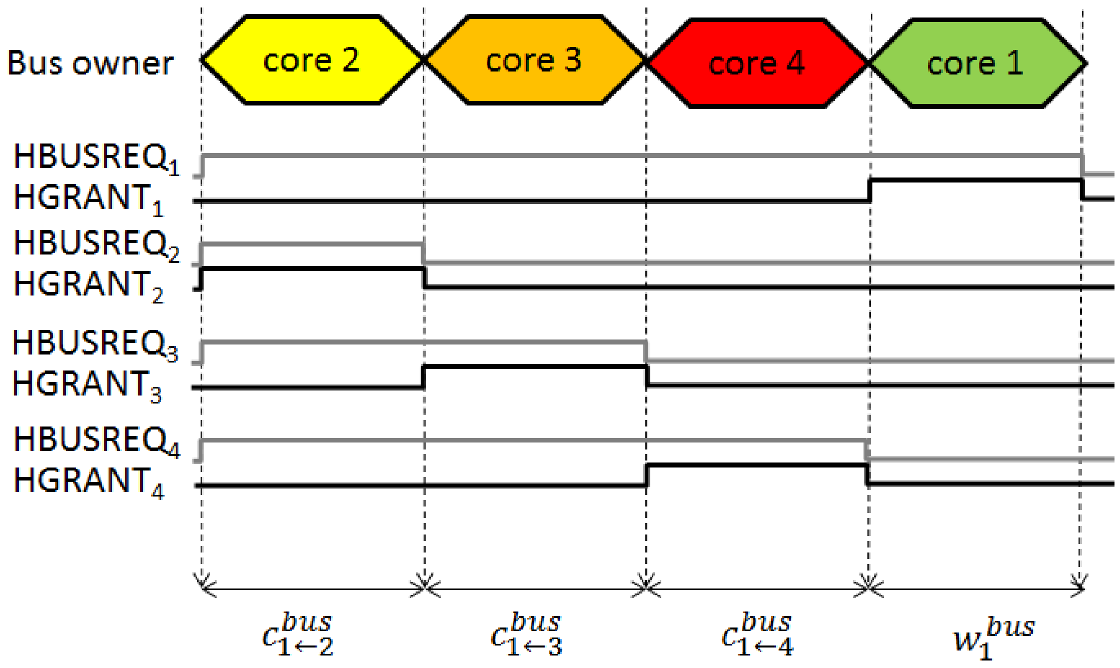


Figure 9.3: Simplified AMBA arbitration process.

of these signals: $\texttt{HBUSREQ}_i$ and $\texttt{HGRANT}_i$. Master m_i asserts $\texttt{HBUSREQ}_i$ to indicate the arbiter that it is requesting the bus. The arbiter asserts $\texttt{HGRANT}_i$ when it grants access to m_i, according to its arbitration policy (not specified by AMBA protocol, though in our case is round-robin [39]). As an illustrative example, Figure 9.3 shows one arbitration process[5] for m_1. Once master (core) 1 is ready to send a request, it asserts the $\texttt{HBUSREQ}_1$ signal. At that point in time, core 2 is using the bus, since $\texttt{HGRANT}_2$ is active. In this example, let us assume that core 3 and core 4 are also waiting for the bus ($\texttt{HBUSREQ}_3$ and $\texttt{HBUSREQ}_4$ are active) and according to the round-robin arbitration policy they both have higher priority than m_1 at this point. The figure shows how the grant is passed from m_2 to m_3 and m_4 respectively. When the arbiter grants access to core 1, it sets the $\texttt{HGRANT}_1$ signal for m_1.

We propose to forward $\texttt{HBUSREQ}_i$ and $\texttt{HGRANT}_j$ signals from the arbiter to the CCSm. By checking these signals the CCSm infers on each cycle which master m_j is using the bus, thus the working cycles w^{bus}_j, and whether another master m_i is waiting for master m_j, i.e. the contention cycles $c^{bus}_{i\leftarrow j}$. Our proposal maintains the same bus interface, since no signals are introduced or modified. We simply forward the existing signals to the CCSm as shown in Figure 9.4. The CCSm has

[5]The timing on the figure is an abstraction of the real AMBA timing, see section 3.11 of [14] for details.

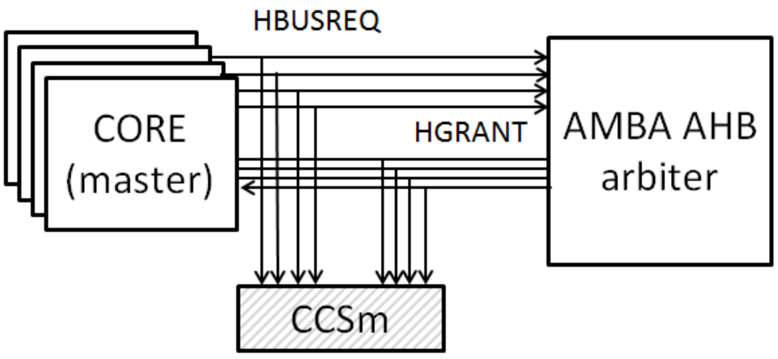


Figure 9.4: AMBA Signals and CCSm.

$N_c \times N_c = 16$ Bus Contention Counters ($bcc_{i,j}$). $bcc_{i,j}$ stores the number of cycles that ($\mathtt{HBUSREQ}_i$) & ($\mathtt{HGRANT}_j$) holds. $bcc_{i,j}$ where $i \neq j$ hold the contention cycles that master i suffers from master j, i.e. $c^{bus}_{i \leftarrow j}$. Counters $bcc_{i,i}$ store the working cycles that master m_i uses to process its requests, i.e, w^{bus}_i. In the bottom part of Figure 9.3 we show how cycles are accounted for core 1.

It is worth noting that the $\mathtt{HBUSREQ}_i$ and $\mathtt{HGRANT}_j$ signals are common to other bus interfaces such as Wishbone (grant and cycle signals), Avalon (waitrequest and read/write signals), VCI (valid and acknowledge signals) or CoreConnect (command and response send and accept signals). For other types of interconnects, such as as AMBA AXI, similar signals are available that allow to monitor the interconnect usage.

Overall, our approach to account CCS does not change the AMBA interface or protocol. Instead we simply snoop the AMBA AHB arbiter signals and with this information we measure the time that a given task waits for the others when it tries to get access to the bus and the time that spends using it, so that all types of accesses are captured [94]. The hardware cost in terms of storage is 16 counters $\times 32$ bits each. That is 64 bytes.

9.4.6 Memory controller

The memory controller comprises a FIFO queue, with one entry per core, the memory bus and a command translator that translates AMBA requests into DRAM commands. When a request from a core c_i arrives at the FIFO queue, if the queue is empty, it is put at the top of the queue and accesses the memory immediately. Otherwise, when other requests are in the queue, it has to wait for them to finish since the memory bus and memory controller only accept one request at a time. The former corresponds to the time the request takes to be processed once it is granted access and it cannot be further delayed by any preceding request. The latter is the time the request waits to get access to the memory controller.

At any given cycle, the core using the memory could be determined if information is provided about the id of the core that is at the top of the FIFO queue, and hence whose request is being processed. The id of the requests in a FIFO queue entry – other than that at the top position – are those cores suffering contention. Hence, the knowledge of the core id for each request in the FIFO queue is needed. In the GR740, the id of the core generating a request is kept in the AMBA AHB as master id. However, once the request accesses the L2 this information is no longer kept.

Intuitively this would require keeping the core id on every L2 cache line, which would incur a significant increase in the L2 cache size. However, in reality the core id is kept in the Miss Status Holding Register (MSHR) of the L2 cache. The process goes as follows. On the event of an access to the L2, the L2 cache determines whether that access is a hit or a miss. In case of a miss, the request is stored in the MSHR with the core id as part of the miss request, to be able to respond to the appropriate master afterwards.

Our proposal propagates the core id from the MSHR to the memory controller FIFO queue. When a request is sent from the L2 to the memory controller it is tagged with the core id. Both the MSHR and the FIFO queue are relatively small with sizes up to 8/16 entries in general. In our case both have 4 entries. Hence our proposal incurs an increase in area of $4 \times \log_2(Nc = 4) = 8$ bits (1 byte).

In terms of logic, each position in the FIFO queue sends a signal to the CCSm with the core id of the request in that position, if any. The core i at the top of the FIFO queue, $\texttt{FIFO}(0) = i$, is the one accessing the memory and the rest of the cores in the FIFO queue, $j \in \texttt{FIFO} | j \neq i$, are those interfered by i. The CCSm considers working cycles in memory for core i, those cycles when the core is at the FIFO's top entry. If there is any other request from another core j in the FIFO queue, the CCSm accounts contention cycles in the memory for them caused by core i.

The CCSm has $N_c \times N_c = 16$ Memory Contention Counters ($mcc_{i,j}$) for the memory controller with an associated size of $16 \times 32/8 = 64$ bytes. Counter $mcc_{i,j}$ stores the number of cycles that ($\texttt{FIFO}(0) = i$) & ($j \in \texttt{FIFO}$) holds. Counters where $i \neq j$ hold the contention cycles that core j suffers from core i. Counters with $i = j$ store the working cycles that core i uses to process its requests.

9.5 Evaluation

We carried out the evaluation of CCS under two different setups, both focusing on the GR740 [39]. First, we model the GR740 on a simulation tool in which we implemented the PMCs presented in Section 9.4. In our second evaluation environment we focus on an FPGA model of the GR740 in which Cobham Gaisler implemented the PMCs for the bus, the CCSm and their signals.

Benchmarks. As reference applications we use the EEMBC Autobench suite [143] (see Section 3.2). In particular we use: *a2time*, *aifftr*, *aifirf*, *aiifft*, *basefp*, *cacheb*, *canrdr*, *idctrn*, *iirflt*, *matrix*, *pntrch*, *puwmod*, *rspeed*, *tblook* and *ttsprk*. We also developed a set of synthetic kernels that inject constant high pressure either on the shared bus or on the shared memory. The Bus-Stressing Kernel, or *bsk*, comprises memory read requests that always miss the L1 and hit the L2, thus maximizing the traffic on the bus. This is done by having 5 memory accesses that access the same set of the L1 cache, thus exceeding its 4 ways. These accesses hit on the L2 cache by targeting different sets on the L2 cache. The Memory-Stressing Kernel, or *msk* comprises memory read requests that always miss on the L1, but in this case they also miss on the L2, following the same procedure of targeting the same set, done for the L1. When our task under analysis runs against three of these kernels, it finds very high contention on the bus or the memory respectively.

For simplicity, in our experiments we run four-task workloads in which the task in core 1 (also referred to as task 1) is the task under analysis (TUA) for which the CCS is derived. The tasks on the other cores are considered contending tasks.

9.5.1 Simulator evaluation

Setup. We model the GR740 [39] running at 200MHz using the validated simulator (see Section 3.3). Each core's private instruction (IL1) and data (DL1) caches are 16KB, 4-way with 32-byte lines. The shared second level (L2) 256KB cache is split among cores, each receiving one way of the L2, so that inter-task contention only happens on the bus and the memory controller. With DRAMsim2 [171] we model a 2-GB one-rank DDR2-667 [104] system.

We implemented the CCSm and its signals, as presented in Section 9.4, including the PMCs required to directly measure the working and contention cycles on the bus and the memory controller. With these modifications we build the CCS directly from the measures obtained from the PMCs in one execution.

Timing Validation. In our first experiment, we use the CCS to cross validate two different methods deriving bounds to the WCD tasks suffer accessing GR740 shared resources. For that purpose we run each EEMBC benchmark under different 4-task workloads and collect the ACD obtained with CCS and compare it with the expected WCD.

WCD bounds with measurements. In this case we run each EEMBC against several copies of the *bsk* or *msk*, which are expected to generate high (potentially the highest) contention in the bus and the memory to the TUA.

Theoretical WCD bounds. In this case, WCD bounds are predicted by multiplying the number of accesses the benchmarks does to the bus (and the memory)

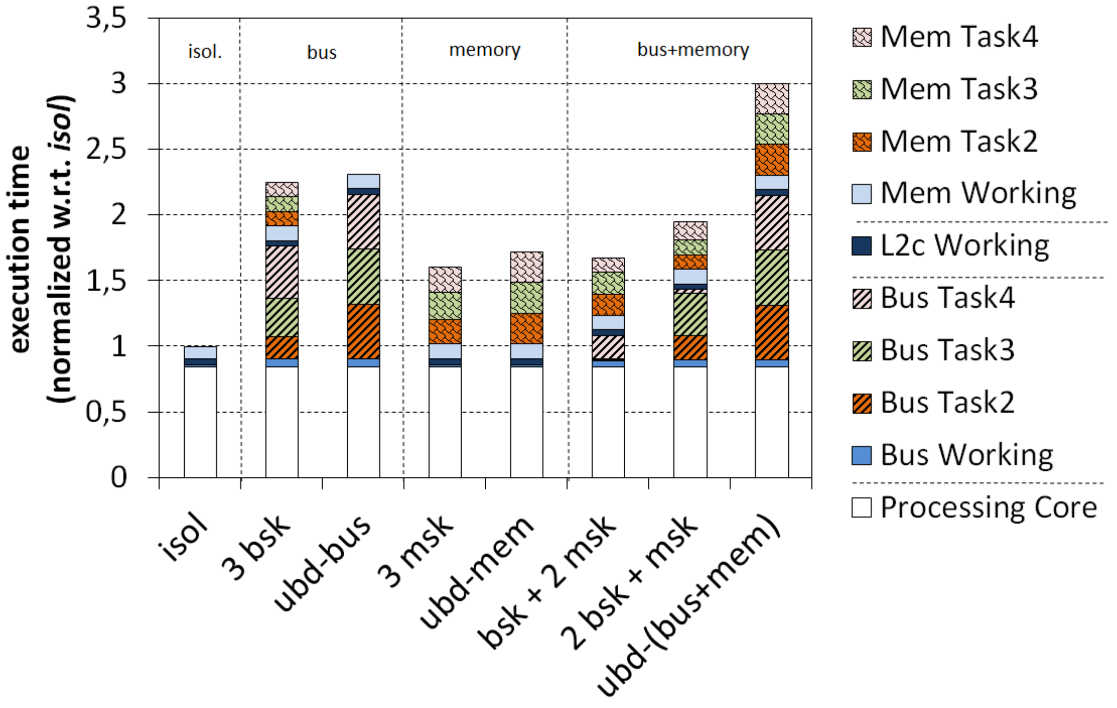


Figure 9.5: EEMBC with different contention workloads.

by the maximum delay each request can suffer – called Upper-Bound Delay or *ubd* [138] – in the access to the bus (and the memory). In the case of the bus, assuming that all accesses target the L2 cache, *ubd-bus* is 27 cycles that corresponds to the latency of three contending cores sending requests that have the highest latency (9 cycles) in our setup. In the memory case, *ubd-mem* is 69 cycles that corresponds with three complete row accesses of the contending cores, i.e. $3 \cdot t_{RC}$, with $t_{RC} = 23$.

Figure 9.5 shows the CCS obtained when running each EEMBC benchmark in isolation (*isol*); when using the measurement-based WCD model that runs each EEMBC against either three *bsk* or *msk*; and when using the theoretical WCD model for which the CCS for *ubd-bus* and *ubd-mem* are constructed replacing the interference on *bsk* and *msk* with the theoretical *ubd* for each resource respectively or for both in the *ubd-bus+mem* case. For simplicity, Figure 9.5 shows results averaged across all the EEMBC Autobench.

We observe that the measurement-based WCD model, even if *bsk* and *msk* put a high load on the resource, never exceeds the bounds provided by the theoretical *ubd*-based model. In particular for every shared resource and contending task the theoretical bounds are higher than the measured ones.

– This experiment shows that the approach based on running each TUA against

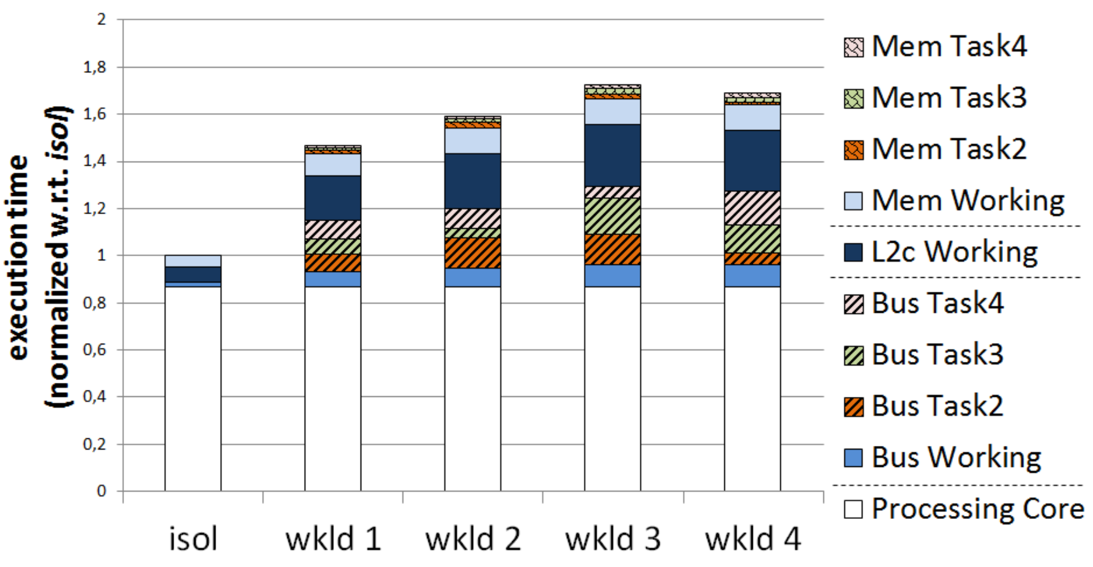


Figure 9.6: *cacheb* EEMBC in different workloads.

3 copies of *msk* and *bsk* does not capture that theoretical bound. For the case of *bsk* we observe that these three benchmarks generate some activity in the memory (i.e. `Mem Task2`, `Mem Task 3` and `Mem Task 4` are non zero). This can be the reason why these benchmarks do not generate maximum contention delay in the bus. Said that, it is the case that this models approximates quite well the maximum theoretical value using *ubd* in both, bus and memory, but not in a combined manner [144] as shown in the rightmost bar (*ubd-bus+mem*) of Figure 9.5.

– The result of this experiment also shows that in the experiments done nothing suggests that theoretical WCD bounds are violated for any of the contender tasks in any resource. This increases confidence on the validity of those bounds.

Scheduling Optimizations. In our second experiment we show how CCS could be used to improve other system metrics. For this experiment we choose *cacheb* EEMBC benchmark as TUA. Figure 9.6 shows the CCS obtained for the TUA under different workloads, composed of randomly picked EEMBC benchmarks. First, we see that most of the interference is suffered on the bus, because the L2 is filtering the accesses to the memory. We also see that Task2 in workload 2, i.e., (wkld:2,task:2)=(2,2), has high impact of TUA's execution time due to the interference on the bus. On the other hand, (2,3), (3,4) and (4,2) have a low impact on TUA's execution time. This information can influence the scheduling decisions, for instance, allocating the tasks with the highest interference, such as (2,2), into the same core as the TUA so that they do not interfere each other.

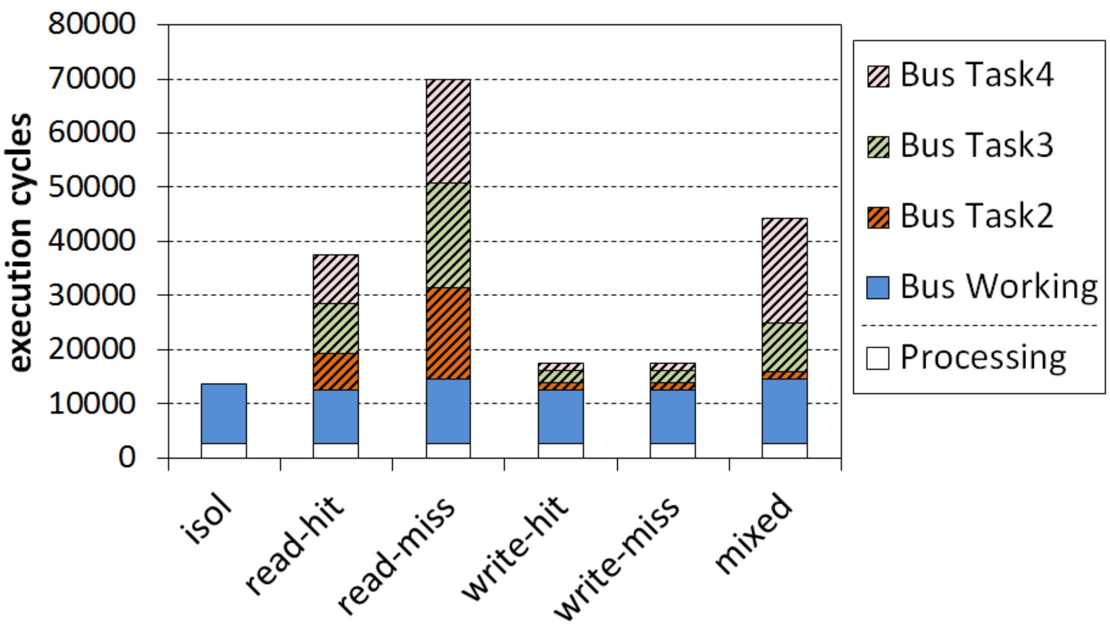


Figure 9.7: *r-bsk* under analysis against different bus contenders.

9.5.2 FPGA Evaluation

Setup. We implemented the CCS for the AMBA AHB bus on a GR740 FPGA prototype using a Xilinx ML510 evaluation board. In particular Cobham Gaisler implemented the bus CCS-aware PMCs proposed by us that measure the working and contention cycles on the processor AHB bus. We used the commercially available Cobham Gaisler GRMON2 [37] debug monitor software to directly extract the CCS from the statistics unit (L4STAT) of the GR740, without affecting execution. The CCS is directly constructed from the readings obtained in one execution of the task, i.e, no further post processing is required.

The real cost of the modifications is low since we reuse the available counters and infrastructure of the GR740 L4STAT unit. Our PMCs just require the wiring of the AMBA signals, which corresponds to 8 1-bit signals, that is 4 HBUSREQ and 4 HGRANT signals. The cost of the wiring depends on the target technology, synthesis tool and design size.

As TUA, we use bare-metal resource stressing kernels that put high load on the bus using either *read* or *write* bus requests, called *r-bsk* or *w-bsk* respectively.

Evaluation. To evaluate the design we use different *bsk* contenders with different type of requests, *read* or *write* that either *hit* (*read-hit*, *write-hit*) or *miss* (*read-miss*, *write-miss*) on the L2. Each type of request causes a different bus contention due to the different behavior of requests and L2 latencies.

Figure 9.7, shows the CCS when taking as TUA *r-bsk* against different work-

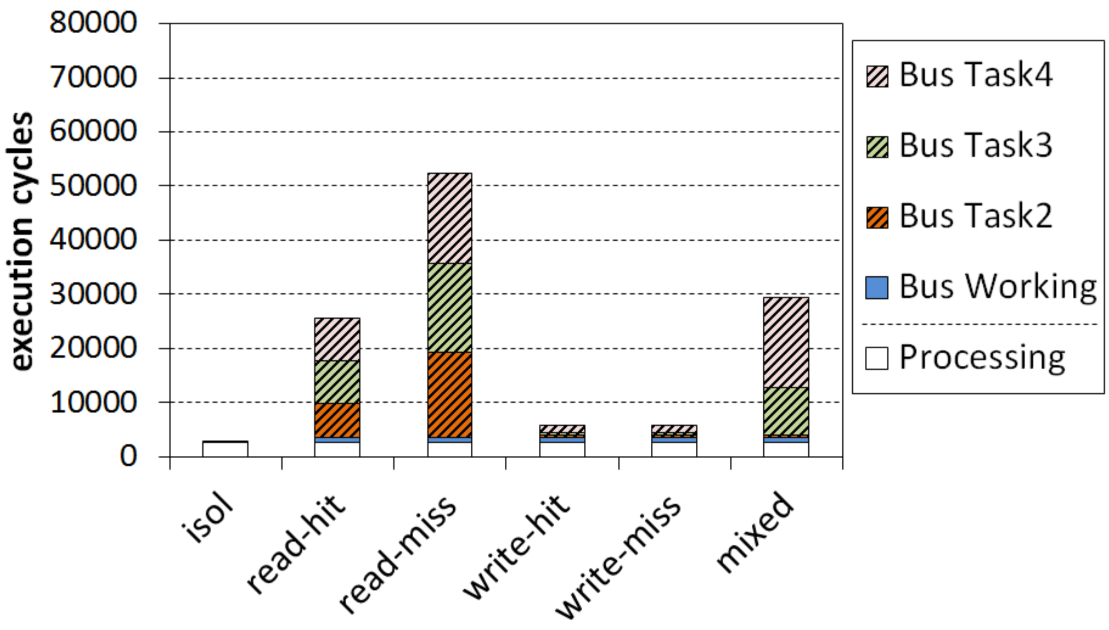


Figure 9.8: *w-bsk* under analysis against different bus contenders.

loads consisting of different types of *bsk*, shown on the x-axis. We observe that the worst effect is caused by the *read-miss* workload, followed by the *read-hit*. This happens because read requests that miss on the L2 hold the bus while accessing memory. On the other hand, read requests that hit on the L2 do not access memory, thus, requiring less time on the bus. We also see that *write-hit* and *write-miss* workloads have the same effect on the bus, which is smaller than both *read* workloads. This happens because write requests only require an acknowledge and are immediately responded, even if they miss on the L2.

The rightmost bar in Figure 9.7 shows a mixed scenario, in which Task2 uses *write misses*, Task3 *read hits* and Task4 *read misses*. The obtained CCS effectively demonstrates the capabilities to identify the contender with the highest interference in a workload in which contenders have different resource usage profiles, Task4 in this case.

Figure 9.8 shows the same contention scenarios using *w-bsk* as TUA. As shown, the bus working cycles reduce in comparison with Figure 9.7, because *write* bus requests from *w-bsk* take much shorter time in the bus than *read* requests due to the longer L2 cache read hit latency. We observe the same contender behavior with similar amount of contention, even though *w-bsk* requires less time on the bus than *r-bsk*. This happens because contention mostly depends on the amount of interfered requests and not on their duration, since once a request accesses the bus, it cannot be preempted.

Overall, CCS provides valuable and accurate information to build and validate timing models and WCD bounds.

9.6 PMC Survey

In this section we provide further details on the PMC support of different architectures that can be useful to measure contention in shared resources to some extent.

IBM. The IBM POWER7 [98] is an 8-core multicore in which each core is 4-thread Simultaneous Multithreaded. Each core is divided into two clustered execution pipelines, with each one supporting two threads. Program performance analysis in such an aggressive core architecture with resources shared among different threads is complex. The POWER7 processor comprises a Performance Monitoring Unit (PMU) with six thread-level PMCs. Four of these are programmable from software to monitor the desired (four) events at the same time. There are more than 500 possible performance events that can be read.

Based on our analysis of the POWER7 we identified counters for cycle utilization, busy and idle cycles, occurrences of a number of events, instructions of each type executed, and the amount of data transferred. PMCs in the *threshold* and *current events* were not found.

Intel. Usually Intel processors feature superscalar execution, complex branch predictors, out of order execution, and several levels of cache memories. Provided PMCs focus on providing performance metrics for a single process with counters for analyzing branch predictor effectiveness, cache misses due to speculative execution, coherence protocol metrics, etc. Most counters can be configured to measure events for either from a core or all the cores, an agent or all agents, and other kinds of specific qualifications (such as detection of all events/exclude prefetching events, or counts for different states for the coherence protocol used). In this work we have analyzed the PMCs available in the following Intel architectures: Haswell (Xeon E3-1200 v3), Ivy Bridge (Xeon E3-1200 v2), Sandy Bridge (Core i7-2xxx, Core i5-2xxx, Core i3-2xxx, Xeon E3-1200), Nehalem (Core i7, Xeon 5500 Series), and Westmere (Xeon E7-xxxx).

Intel's PMCs are to some extent similar to those of the IBM POWER7 since both processors are general-purpose high-performance ones. Still, some relevant differences exist. For instance, Intel processors don't have explicit support to measure data transferred. However, Intel's PMCs include the following counters:

– *Threshold exceeding count*. These counters measure the number of times a threshold specified in number of cycles has been exceeded for a given event. The threshold value is configured by the user.

– *Outstanding Requests*. This counter measures events such as cache requests, all offcore requests, etc. in the moment of reading the counter.

ARM v7. The ARMv7-A architecture [11] provides 6 different 32-bit counters, which can count any event available. This architecture is used, among other by the Cortex-A7, Cortex-A9, Cortex-A15 [15] and the big.LITTLE [76] system. ARMv7-A provides a Performance Monitoring Unit (PMU) with 6 performance counters. The events counted by ARM architectures are a subset of those available in high-performance processors such as Intel and IBM ones described before as shown in Table 9.1. Still this architecture is rich in counters for a number of events such as miss-predicted branches, number of exceptions, L1 write backs, number of L1/L2 refills, L1 accesses, bus accesses and data memory accesses among others.

Freescale. Freescale P4080 processor hosts eight e500mc [72] cores, which are superscalar processors that can issue and complete two instructions per clock cycle. Each core has a private L1 instruction and data cache. It also has a private L2 unified cache. The eight cores are connected through a proprietary CoreNet Fabric coherent interconnect with two shared 1MB L3 off-chip caches. Each L3 off-chip cache is connected to a separate DDR memory controller.

The PMC support in the P4080 offers dedicated core and SoC platform counters [68, 69]. At the core level, the e500mc core allows monitoring 256 different hardware events, each core being able to monitor 4 different events at a given time in 4 dedicated 32-bit registers. At the SoC level the P4080 Event Processing Unit (EPU) allows counting SoC platform events of interest. Although some events across Intel and Freescale processors differ due to their different designs, in essence, the set of PMCs in both platforms is quite similar as shown in Table 9.1.

GR740. The GR740 is a 4 core LEON4 processor developed at Cobham Gaisler under contract with the ESA. It contains one or more LEON4 Statistical Units (L4STAT). The debug driver for L4STAT provides an interface for reading and configuring the 32-bit performance counters available in a L4STAT core. Each L4STAT allows configuring any available sixteen events we want to monitor. Each counter has an associated control register. Both the counters and the control registers are mapped to the peripheral address space.

The available events can be divided in three different categories, depending on the component counting the events. Processor events: events generated by the processor, e.g., pipeline or the L1 cache; bus events: events generated by the bus, e.g., busy cycles or number of read accesses; and Device specific events: events generated by other devices such as the L2 or the IOMMU.

Again, although the PMCs across chip vendors do not match, the type of counters one can find in the GR740 is quite similar to those in the Freescale and Intel architectures, with the exception of the idle cycle counters that the GR740 does include for the bus.

9.7 Related Work

Performance Monitoring Counters (PMCs) have been traditionally used to measure average performance and power consumption [125]. One of the few works that addresses contention monitoring between tasks is [189], which uses cache scouts to monitor contention on shared caches. However, with few exceptions [163, 33], cache partitioning is the common solution in the context of CRTES due to the complexity of estimating the WCET accurately on top of shared caches. Although our work could be ported to non-partitioned caches building on [189], we leave this analysis for future work.

The IBM POWER family, starting with the POWER5, have developed a Cycles-Per-Instruction (CPI) stack that covers the resources on which each task spends its cycles. The CPI stack reports the cycles spent in each core resource. For instance in the load/store Unit (LSU) [98]. This happens when a load/store operation is stalled. The CCS instead does not focus on the local resources getting clogged by contention (e.g. the LSU) but the off-core resources identifying where (and how much) contention occurs and the contending core producing it.

In [159] authors use custom PMC to derive WCD and WCET estimates with measurement-based timing analysis on a bus-based system. Authors assume that the WCD for the bus is known, which is not always the case in real implementations, as shown for the AMBA AHB bus in Chapter 4. Several works derive bounds, during the analysis phase of the system, to the WCD that a task may suffer in different processor resources assuming static or measurement-based timing-analysis. We do not detail these works, already explained in Section 2.2, since we focus on measuring the ACD tasks suffer rather than on making a-priori predictions on WCD.

The ACD in multicore processors has been characterized mostly using resource-stressing kernels (rsk) [144, 1], for instance in a previous implementation of the NGMP [65] or in the Freescale P4080 [134]. However, these approaches provide neither a breakdown of the ACD nor means to measure it accurately online, while CCS provides both.

Authors in [135] propose a *runtime monitoring* to control the resource usage of tasks running on a multicore, preventing tasks from having resource usage limit violations. Authors make use of access count PMCs, such as bus access counts. However, it has been shown in Chapter 4 that bus latencies may differ accross different types of accesses and even for the same type. As a result, access counts does not provide the actual impact of contention time which has to be estimated. The ACD instead provides in an exact manner contention delay for each task.

9.8 Summary

Obtaining accurate ACD breakdowns in MPSoCs provides evidence about the trustworthiness of contention bounds and increases confidence on derive execution time bounds; it is also of prominent importance to detect the reasons for overruns in a critical system once it has been deployed. It also helps optimizing system's performance and improving scheduling decisions based on the knowledge of the real impact of contenders in terms of execution time. Unfortunately, to the best of our knowledge no solution exists to measure and classify the ACD in CRTES.

In this chapter we propose the Contention Cycle Stack (CCS), which provides an effective representation of the ACD that classifies contention per resource and contending task by means of measurements. The CCS relies on some existing Performance Monitoring Counters (PMCs) and introduces its own-set of low-overhead PMCs to track the events that allow building ACD per task and shared resource. Our evaluations for the Cobham Gaisler GR740 (NGMP) show the benefits of the CCS. In particular CCS enables understanding the source and magnitude of the actual contention delay (ACD) caused by contending tasks in a MPSoC, which is crucial to adopt MPSoC in CRTES.

Chapter 10

Bounding Resource Contention in the NGMP

10.1 Introduction

In a time-anomaly [149] free multicore system, time composability can be achieved by modeling at analysis time a scenario in which each access that the task under analysis (τ) makes to a hardware shared resource suffers the highest contention possible. For instance, in the case of a round-robin bus accessed by Nc cores this is equivalent to assuming that each request suffers maximum contention from each of the remaining $Nc - 1$ cores. That is, the single-access maximum (contention) delay, or *samd*, corresponds to:

$$samd = (N_c - 1) \times L_{bus} \tag{10.1}$$

where L_{bus} is the maximum bus latency for a single request. The resulting Execution Time Bound[1] (ETB) estimate in this scenario is *fully time composable* since it accounts for the maximum load that corunner tasks of τ can put (at operation time) on the target resource. This, though, comes at the cost of inflated ETB estimates (e.g. up to more than 5x times in a 4-core processor as reported in [65]). Tighter ETB estimates can be obtained by adjusting the bounds to the actual load that corunner tasks put on the target resource, which can be abstracted with an arrival curve [158]. However, time-composability is lost since the ETB for a

[1]We use Execution Time Bound (ETB) instead of Worst-Case Execution Time (WCET) estimate to refer to the upper-limits derived for tasks execution time in multicore. The reason is that WCET estimates, as they are commonly understood, establish a single value that upperbounds program's execution time under any circumstance. However, since the bounds provided in this chapter hold only under specific circumstances (subject to the characteristics of the corunners), we prefer not to use WCET.

task becomes dependent on its particular corunners. This confronts industry with the choice of time-composable inflated estimates or tighter non time-composable estimates.

In this chapter we use measurement-based timing analysis, which a large fraction of safety-related systems resort on [180] – including the space industry. We propose a contention-prediction model that captures the effect of contention in the NGMP shared resources. For a given task, τ, our model enables deriving both fully time-composable bounds to the contention delay suffered by τ or partially time-composable bounds [61] *which depend on the number of requests generated by τ's corunner tasks, N_{req}, but not on how they align with τ's requests.* Derived bounds are valid for different corunner tasks as long as they generate at most N_{req} requests.

Our approach is motivated by the fact that, while the number of requests that a task generates can be bound with existing tools like Rapita System's Verification Suite (RVS) [147], how τ's and its corunners' requests interleave is hard, if at all possible, to measure and control. Hence, instead of predicting request interleaving, our approach derives contention delays for the worst-possible time-alignment of requests. The main contributions of this chapter are as follows:

1. We make an in-depth analysis of the hardware shared resources in the NGMP, the way in which requests interact and the delay they may suffer on those resources.

2. We present a prediction model for the contention delay in the bus and the memory controller in the NGMP. Our model, which depends on the time requests take to access shared resources, deals with the case when there are several types of accesses to a resource and each type causes and suffers a different delay depending on the contending accesses. For instance, in the processor AMBA AHB bus, loads missing in the L2 take shorter than loads hitting in L2. We show how our model handles this case.

3. We evaluate our proposal in a solid setup comprising the GR740 implemented in a FPGA. Our proposal provides tighter ETBs than the fully time-composable proposal in [65], since it adapts to the contenders' load on shared resources in a still partially time-composable and friendly way.

The rest of this chapter is organized as follows: Section 10.5 presents the related work. Section 10.2 provides information on the GR740. Section 10.3 details our prediction model. Section 10.4 assesses the accuracy of our model. Finally, Section 10.6 presents the summary of this chapter.

10.2 NGMP

The NGMP quad-core processor is described in Section 3.1. The NGMP comprises 16 Performance Monitoring Counters (PMC) that can be configured with different events, providing support to measure access counts such in a way that it facilitates the implementation of our prediction model, more details are provided in Section 10.3.3. This section provides details on some aspects related to the contention in the access to NGMP's shared resources.

10.2.1 AHB Processor Bus

The AMBA AHB bus connects cores to the L2 cache and the I/O bridges[2]. The first consideration to make in the case of the bus is that there are different types of requests that can generate different inter-task contention: bus reads (loads) that either hit (*l2h*) or miss (*l2m*) on the L2 cache and bus writes (stores) that either hit (*s2h*) or miss (*s2m*) on the L2 cache. These accesses behave differently because hits hold the bus while they are served. Instead, misses wait on a miss queue and are split, i.e. the L2 cache releases the bus while processing the miss, so that other cores can use the bus. In the NGMP, the AMBA AHB bus implements round-robin arbitration.

10.2.2 L2 Cache

In our experiments we use the master-index feature of the NGMP that partitions the L2 assigning one L2 cache way to each core. Hence, a given core suffers no contention interference in the L2 due to other cores' evictions.

Each of the request types identified before (*l2h*, *l2m*, *s2h* and *s2m*) has its own L2 access latency. Interestingly, the latency of requests of the same type can be variable. That is, for each request type access there is a Best-Case (BC) and a Worst-Case (WC) latency. This jitter is caused by the type of previous requests, despite they belong to a different task and hence go to a different cache partition. Our model takes this effect into account by assuming that all latencies suffered on the experiments have the BC and when computing the contention bounds, we add a correcting value that adds for each L2 access the corresponding difference between the WC and the BC. This adds pessimism but its advantage is two-fold: it is a safe upperbound and it removes the need to track the sequence of accesses to determine their exact latency.

The WC and BC latencies are obtained from table 40 in [41] and are 8, 13, 6 and 7 for *l2h*, *l2m*, *s2h* and *s2m* respectively in WC and 5, 6, 0 and 0 for BC.

[2]In this work, we do not consider I/O related activities, which we assume managed at software level, so that only accesses to L2 interfere each other.

10.2.3 Memory Controller

The memory controller acts as an interface between the processor and the DRAM memory. We differentiate two types of request in the memory: read and write. According to the DRAM protocol, each request has a latency to be responded depending on whether it is a read or write request respectively. The latency it takes the memory to go back into idle state, once a request starts being processed, is fixed regardless of whether the request is read or write and corresponds to the time till a new request can be processed. For this chapter, we assume that the memory controller behaves as a FIFO queue. This is a simplification that helps upper bounding the memory controller latency though it introduces some pessimism. Providing a more accurate model of the memory is part of our future work.

10.3 Prediction Models

Our prediction models use measurement-based timing analysis techniques to derive a multicore ETB (ETB_{mc}) for a task τ_i, given its ETB in isolation (ETB_{isol}). To that end, the models predict the total effect of contention in the access to the multicore hardware shared resources, called Contention Delay Bound (CDB), and add it to the ETB in isolation:

$$ETB_{mc} = ETB_{isol} + CDB \tag{10.2}$$

In order to derive CDB, we add the contribution of each hardware shared resource r, CDB_r:

$$CDB = \sum_{r \in R} CDB_r \tag{10.3}$$

To derive CDB_r, we upper-bound the maximum latency that every access from τ_i to r, n_i^r, may suffer from requests generated by τ_i's corunner tasks, referred to as $c(\tau_i)$.

CDB_r for τ_i assumes that each $\tau_j \in c(\tau_i)$ performs at most a given number of accesses (n_j^r) to resource r. Therefore, ETB_{mc} estimate for τ_i is composable with any other task $\tau_j' \in c'(\tau_i)$ as long as it performs fewer accesses ($n_j'^r$) to the shared resource than $\tau_j \in c(\tau_i)$:

$$n_j'^r \leq n_j^r \tag{10.4}$$

In this chapter we follow the theoretical approach in [64] that proposes a methodology to obtain the resource access 'profile' of a given task that defines the use of resources that the task makes on a target shared resource. That profile is used to derive the contention tasks suffer and generate when accessing that resource. In this work, which is a collaboration of end-users in the Space domain (Airbus Defence and Space and ESA), hardware technology providers (Cobham Gaisler) and a research institution (Barcelona Supercomputing Center) we assess the benefits of such an approach on a real platform, the GR740 addressing issues related to NGMP specific arbitration policies and access types to the different resources.

10.3.1 Bus Prediction Model

The NGMP comprises three main shared resources in its data path: the bus, the L2 cache and memory. Since the L2 can be partitioned we do not consider contention of the different tasks in the L2. We start by predicting CDB_{bus} for the bus and later apply the same approach for memory.

We explain three different ways of upper-bounding CDB_{bus}, which present represent different trade-offs between information required, such as the number of accesses of each corunner task, and tightness of the produced bound.

A.1. Theoretical Upper-Bound Delay (UBD)

In this reference model, based on [138], we assume that every single τ_i request is delayed by a request from each of the $N_c - 1$ contenders and that contending requests cause the highest delay, L_{bus}. This is the maximum contention scenario in round-robin arbitration, where the upper-bound delay a request can suffer is given by:

$$samd = (N_c - 1) \times L_{bus} \tag{10.5}$$

Hence, for τ_i with b_i accesses to the bus, CDB_{bus} is presented in Equation 10.6, where L_{bus} is the maximum delay any interfered request can suffer from a single interfering request.

$$CDB_{bus} = b_i \times samd = b_i \times (N_c - 1) \times L_{bus} \tag{10.6}$$

Since we have four different types of requests with different latencies: l_{l2h}, l_{l2m}, l_{s2h} and l_{s2m}:

$$L_{bus} = \max(l_{l2h}, l_{l2m}, l_{s2h}, l_{s2m}) \tag{10.7}$$

This model is time-composable by definition because it assumes that all b_i are interfered by i) the highest impact request from ii) all corunners. These two assumptions are sources of pessimism that enable full time-composability.

Interestingly in this model, the worst alignment among the requests of τ_i and the requests of its corunners is assumed. In reality, it can be the case that some

τ_i requests become ready to be sent to the bus when its contenders requests have been partially processed so that each τ_i request suffers a delay smaller than L_{bus}. However, predicting how this alignment of requests can happen at operation time is hard (if at all possible). Any small shift in the execution of tasks can change it. Hence, this and the following models, provision time in CDB_{bus} for the worst-case alignment of requests.

A.2. Single-type Model

Analogously to the previous model, the one presented in this section assumes that every corunners' request causes a delay of L_{bus} on τ_i. Unlike the previous model, this one takes into account that not all τ_i requests might be interfered by one request of its corunner tasks. This usually happens when corunner tasks have fewer accesses than τ_i.

Let b_j be the number of accesses to the bus that each contender task $\tau_j \in c(\tau_i)$ performs. Given that tasks have different number of accesses, not all of them can interfere each other. In particular, for a given interfering task τ_j running in core j, *in the worst-case* only the minimum between the number of accesses of τ_i, b_i, and the number of accesses of the interfering task, b_j, suffer a contention delay of L_{bus}. That is, no more than b_i accesses can be interfered and no more than b_j can interfere. In order to compute the contention on the bus for task τ_i, we add the contribution of each interfering task τ_j:

$$CDB_{bus} = \sum_{\tau_j \in c(\tau_i)} \min\left(b_i, b_j\right) \times L_{bus} \tag{10.8}$$

A.3. Multiple-type Model

The previous model assumes that each interfering request, i.e. those generated by $c(\tau_i)$, belongs to the worst-interfering type, hence generating L_{bus} delay on τ_i. However, corunner tasks generate requests of different types, each of which incurs a different interference on τ_i. This model takes this into account and breaks down the number of requests of the corunners between *l2h*, *l2m*, *s2h* and *s2m*:

$$b_j = b_j^{l2h} + b_j^{l2m} + b_j^{s2h} + b_j^{s2m} \tag{10.9}$$

The order of these requests, from most interfering to less interfering is, *l2h*, *l2m*, *s2h* and *s2m* (see Section 10.3.3).

To compute the CDB_{bus}, we pair each interfered request (those coming from τ_i) with the worst eligible interfering request available from each contending core. We start pairing the accesses with the most interfering type (*l2h*) until this interfering type is consumed. The remaining $b_i' = max(0, b_i - b_j^{l2h})$ requests from τ_i are paired with the next interfering type (*l2m*). The remaining $b_i'' = max(0, b_i' - b_j^{l2m})$ with *s2h* and finally the remaining $b_i''' = max(0, b_i'' - b_j^{s2h})$ with *s2m*. With this CDB_{bus} is computed as follows:

$$\begin{aligned} CDB_{bus} &= \textstyle\sum_{\tau_j \in c(\tau_i)} [\quad \min\left(b_i, b_j^{l2h}\right) \times l_{l2h} + \\ &\qquad \min\left(b'_i, b_j^{l2m}\right) \times l_{l2m} + \\ &\qquad \min\left(b''_i, b_j^{s2h}\right) \times l_{s2h} + \\ &\qquad \min\left(b'''_i, b_j^{s2m}\right) \times l_{s2m}] \end{aligned} \tag{10.10}$$

It is worth noting that the type of the requests generated by τ_i are equally affected by each type of request of its corunner. That is, the interference is determined by the type of the request of the corunner task τ_j only.

10.3.2 Memory Prediction Model

To compute CDB_{mem} we apply the same models as for the bus. As explained in Section 10.2, there are two different types of request in the memory, read and write. We assume a task τ_i with m_i requests to the memory and contender tasks $\tau_j \in c(\tau_i)$ with $m_j = m_j^{read} + m_j^{write}$ accesses to the memory each and $m'_i = max(0, m_i - m_j^{read})$. The highest delay in memory is given by $L_{mem} = \max\left(l_{read}, l_{write}\right)$.

Under these constraints the theoretical Upper-Bound Delay model is given by:

$$CDB_{mem} = m_i \times samd = m_i \times (N_c - 1) \times L_{mem} \tag{10.11}$$

The model based on single request types is as follows:

$$CDB_{mem} = \sum_{\tau_j \in c(\tau_i)} \min\left(m_i, m_j\right) \times L_{mem} \tag{10.12}$$

The model based on multiple request types is as follows:

$$\begin{aligned} CDB_{mem} = \sum_{\tau_j \in c(\tau_i)} & \min\left(m_i, m_j^{read}\right) \times l_{read} + \\ & \min\left(m'_i, m_j^{write}\right) \times l_{write} \end{aligned} \tag{10.13}$$

From previous discussions it follows that our model builds on two pieces of information: the latency each request uses each shared resource and the number of accesses performed by each task to each shared resource. We describe both in the following subsections.

10.3.3 Deriving Access Latencies

Bus. Our model uses as an input the time each request uses each shared resource, which correspond to the bus and memory in our reference architecture. For the bus, in the NGMP, our model requires deriving the bus usage latency of *l2h*, *l2m*, *s2h* and *s2m*[3]. Since documentation typically does not provide this information, we derived it empirically.

To do so, first we executed a benchmark performing a given type of bus operations as the Task Under Analysis (*tua*), or interfered task; against a range of other benchmarks, or corunner tasks, performing all of them the same type of accesses (which may be a different type as for the *tua*). For instance, in one experiment the *tua* performs *l2h* accesses and the corunner tasks *s2h* accesses.

As a result of performing this process we reached the following three observations:

1. The execution time of the *tua* depends on the type of accesses performed by the corunner tasks. Thus, given a *tua*, its execution time may not be the same if corunners perform *l2h*, *l2m*, *s2h* or *s2m* accesses.

2. The impact of corunner tasks on the *tua* is linear with the number of corunners. Therefore, if the execution time of the *tua* in isolation (normalized) is 1, and it grows to $1+K$ when running against one corunner, then the execution time against C corunners can be upper bounded as $1 + C \times K$ (further considerations on this matter can be found in [63]). In the particular case where corunner tasks are run in all other cores, the execution time of the *tua* is:

$$1 + (N_c - 1) \times K \tag{10.14}$$

3. The impact of interferences in the execution time of the *tua* is independent of the particular access type performed by the *tua*. Therefore, the execution time in isolation grows by $(N_c - 1) \times K$ when all corunners perform the same type of accesses (i.e. *l2m*), but K depends solely on the type of accesses of the corunners, not on the type of accesses of the *tua*. This can be explained because the interference on the AMBA AHB bus depends only on the arbitration time (see Chapter 4), which in fact depends only on the time the higher priority corunners use the bus and not on the interfered request which is requesting the bus and has to wait the same amount of time regardless its particular access type.

To infer the latencies we take as a reference the *l2h* benchmark that constantly accesses the L2 cache and hence the bus. Further since the benchmark always hits in L2, each request on the bus has a short turn-around time. This benchmark is

[3]Please note that these latencies are not the same as those obtained in Section 10.2.2 for the L2 cache.

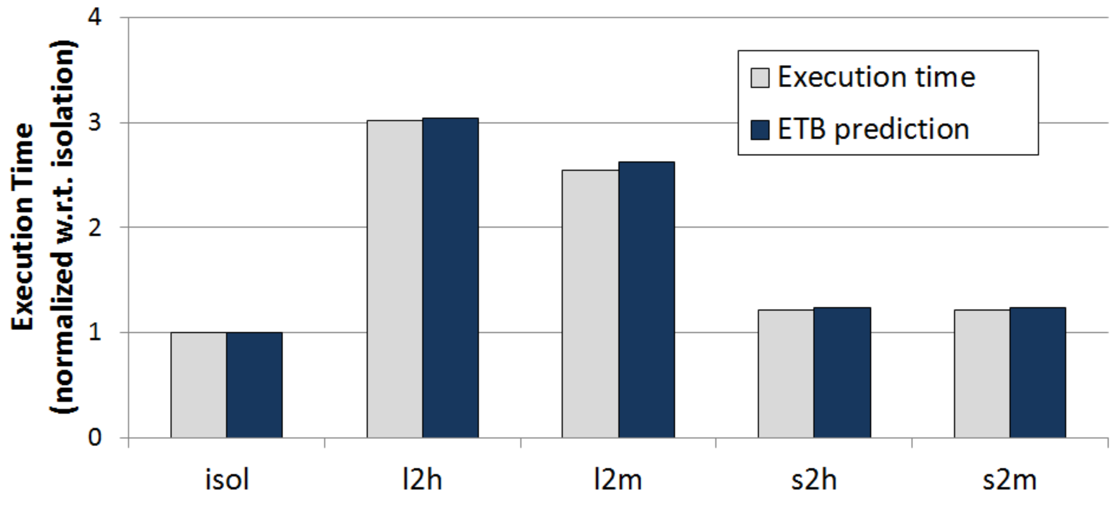


Figure 10.1: Execution time and ETB of *l2h* benchmark in different workloads

executed as *tua* in a workload comprising 3 corunner benchmarks, which correspond to the 3 remaining cores. The corunners perform accesses of the same type to the bus continuously. Hence, there are 4 different workloads depending on the type of access performed by the other three corunners: *l2h*, *l2m*, *s2h*, *s2m*.

Figure 10.1 shows the measured execution time for all workloads. To infer the bus latencies, we divide the execution time overhead of the *tua* with respect to the execution time in isolation by the amount of contenders (3 in each case) and then divide these cycles by the amount of bus accesses performed by each contender. For instance, given an execution time of T_{isol} for the *tua* in isolation and T_{l2h} for the *tua* against 3 *l2h* corunners, the interference of an *l2h* access is obtained as follows where N_{req} is the number of *l2h* requests performed by each corunner:

$$l_{l2h} = \left\lceil \frac{T_{l2h} - T_{isol}}{(N_c - 1) \times N_{req}} \right\rceil \tag{10.15}$$

This way we obtain the number of interference cycles per bus access type: 9, 7, 1 and 1 for l_{l2h}, l_{l2m}, l_{s2h} and l_{s2m} respectively[3]. With these latencies we compute CDB_{bus} with Equation 10.6 and build the ETB prediction shown in Figure 10.1, which is computed using Equation 10.2.

Techniques to improve the confidence on derived bus latencies are proposed in [63]. Part of our future work consists of integrating those methods on top of our model and compare them against our method to derive latencies. Nevertheless, our prediction models are compatible with any method to obtain the access latencies.

Memory. The approach followed to obtain memory latencies is analogous to that for bus latencies with some small differences. First, instead of using benchmarks accessing the bus, we use *l2m* as *tua*, which performs memory reads. As corunner tasks we use first 3 copies of a *l2m* benchmark, that generates memory reads. The latency of memory reads obtained in this case is 18 cycles. In the second experiment we use 3 copies of *s2m* as corunners. The latency of memory writes obtained is again 18 cycles because there is no difference between read and write operations in terms of memory interference since, in both cases, the timing is defined by the time to open and close the memory page or row, which is identical for both.

10.3.4 Deriving Access Counts

The NGMP provides 16 PMCs that can be configured with different events and can be measured using the commercially available tool GRMON2 [37]. Among other events, we are interested in the per-core bus reads and writes (0x40 - 0x50 in [42]) and per-core L2 hits and misses (0x60 - 0x61).

Bus. The total number of L2 accesses (i.e. hits and misses) corresponds to the number of bus accesses. However, there is no way to break down L2 hits/misses into reads and writes, i.e. it is not possible to determine exactly the number of l2h, l2m, s2h and s2m accesses.

In this scenario our approach is to estimate those values in the most pessimistic way: Given task τ_j, we can obtain the number of L2 hits and misses, b_j^h and b_j^m, and the number of bus read and writes, which is equivalent to the number of L2 loads and stores, b_j^l and b_j^s. Our goal is to distribute b_j^h, b_j^m, b_j^l and b_j^s into b_j^{l2h}, b_j^{l2m}, b_j^{s2h} and b_j^{s2m} such that their total impact is maximized. For the l_{l2h}, l_{l2m}, l_{s2h} and l_{s2m} latencies in our reference architecture, the following equations maximize the impact. First, we assume the maximum amount of requests from the worst possible interfering request, i.e. l2h:

$$b_j^{l2h} = \min(b_j^l, b_j^h) \tag{10.16}$$

Then we subtract this value from b_j^l and b_j^h, obtaining $b_j'^l = max(0, b_j^l - b_j^{l2h})$ and $b_j'^h = \max(0, b_j^h - b_j^{l2h})$, and repeat the algorithm with the next worst interferers, i.e. l2m, s2h and then s2m, with $b_j'^s = max(0, b_j^s - b_j^{s2h})$ and $b_j'^m = \max(0, b_j^m - b_j^{l2m})$, to obtain b_j^{l2m}, b_j^{s2h} and b_j^{s2m}.

$$b_j^{l2m} = \min(b_j'^l, b_j^m) \tag{10.17}$$
$$b_j^{s2h} = \min(b_j^s, b_j'^h) \tag{10.18}$$
$$b_j^{s2m} = \min(b_j'^s, b_j'^m) \tag{10.19}$$

Once we have all accesses properly classified as l2h, l2m, s2h and s2h for each contending task τ_j, we can proceed with the model described before.

Memory. Our current implementation does not provide access counters for the memory controller. Hence, the exact number of memory accesses cannot be obtained, even though L2 cache misses are known, since there is no way of accounting indirect memory accesses such as writes generated by evictions of dirty L2 lines. The actual number of memory accesses can either be estimated using the number of L2 misses, which is a lower bound of the memory accesses or estimated with the number of L2 misses and the number of bus writes, which is an upper bound.

10.3.5 Assumptions

Our model is based on the assumption that the number of accesses of a task is not affected by the contenders. This happens only if the L2 is partitioned, i.e. not shared. Otherwise, the number of accesses to the bus or memory for a task executed in isolation does not match those obtained when running along with other tasks.

Also our model assumes that no timing anomalies [149] are present. Timing anomalies is an open research field, and is difficult to prove that a real processor is time anomaly free [126]. Nevertheless, if timing anomalies can occur, they cannot trigger a domino effect by construction, i.e. it is a *compositional architecture with constant-bounded effects* according to the classification in [181]. In those architectures, which may experience timing anomalies but no domino effects, the impact of timing anomalies can be accounted for easily by counting how many times they can be triggered and padding ETBs by the product of this count and the maximum impact of one timing anomaly. This approach is fully in line with our prediction model that accounts for contention in shared resources. Further, note that our model has no impact on timing anomalies, which occur (or not) regardless of our model.

10.4 Experimental Results

We evaluate our proposals on a real GR740 [39] FPGA prototype on a Xilinx ML510 board. We used the commercially available Cobham Gaisler GRMON2 [37] debug monitor software to directly extract the PMC from the statistic unit of the GR740, without affecting execution. The model is directly constructed from the readings obtained in one execution of each task, i.e. no further post processing is required.

10.4.1 Bus and memory prediction models

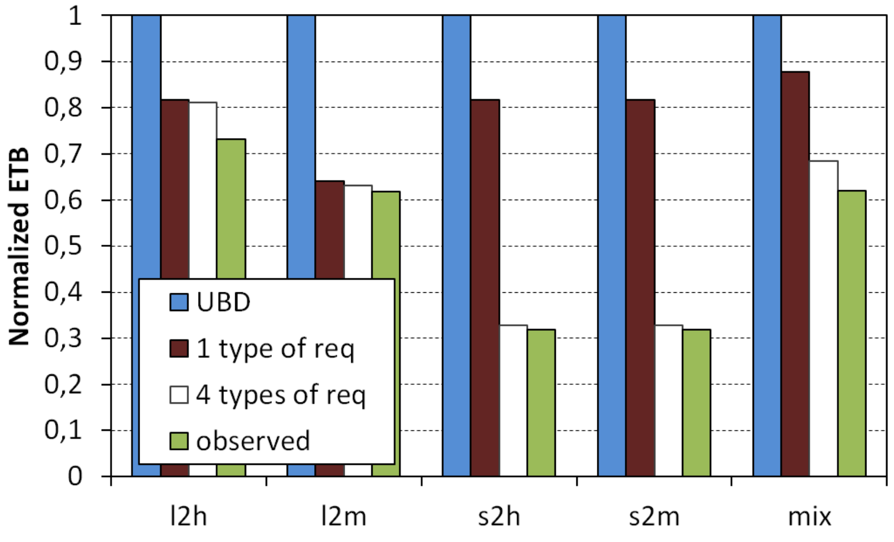
Our first experiments put the shared resources under high pressure to test the tightness of the bounds obtained with the prediction model. To that end we use as reference applications a set of synthetic kernels [65] that inject constant high pressure either on the shared bus or on the shared memory. The Bus-Stressing Kernel, or *bsk*, comprises memory read and write requests that always miss the L1 and hit the L2, thus maximizing the traffic on the bus. This is done by having 5 memory accesses that access the same set of the L1 cache, thus exceeding its 4 ways. The same approach is used for the Memory-Stressing Kernel (*msk*) that comprises memory read requests that always miss on the L1 and also miss on the L2.

In all experiments we use one reference task (*tua*) and three tasks as corunners. In particular, as reference task on which ETB is to be derived we use bsk-ld-40% in which 40% of its instructions are loads that access the bus. As corunner tasks we use *bsk* whose frequency of access is 40% and 5%, i.e. 40% and 5% of the instructions are accesses to the bus. Those *bsk* used as corunners access the bus with requests of a different type across experiments: *l2h*, *l2m*, *s2h*, *s2m* and a *mix*, which consist of a *l2h*, a *l2m* and a *s2h* *bsk* together.

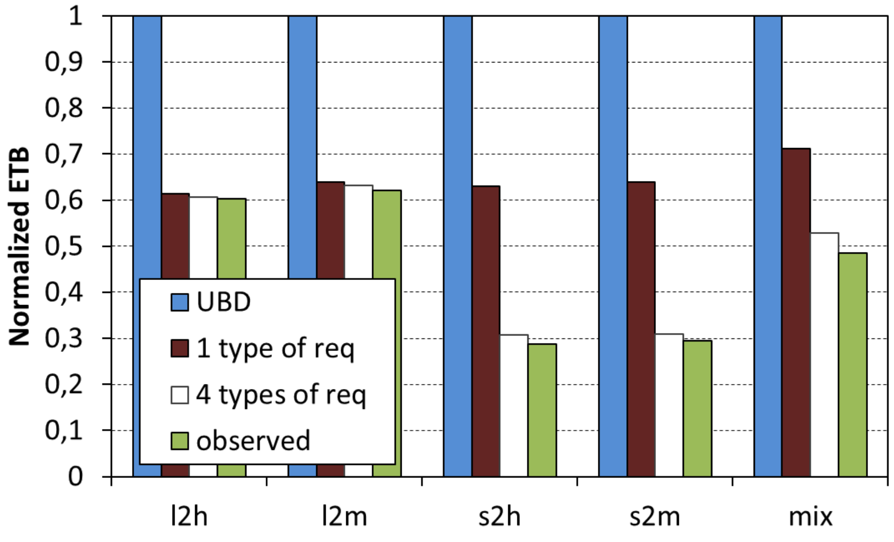
Figure 10.2a and Figure 10.2b show the result for bsk-ld-40% when the frequency of access of the contenders is 40% and 5% respectively. In both figures we show the ETB when using the UBD approach [65] or our approach with a single-type and four types of requests; and the observed execution time. In all cases, the predicted ETB estimates are above the observed execution time. The UBD model, since it assumes that every access of the task under analysis suffers *samd*, leads to the highest ETBs. Our model, that accounts only for the contention the task under analysis suffers, tightens the ETB. As presented in the previous sections, if the method is made aware of the request types and their associated latency (4 types of request) ETBs are further reduced.

In Figure 10.2b, the corunners make fewer accesses than in Figure 10.2a. For the UBD approach this has no impact since it only focuses on the number of requests of the task under analysis that remains the same. Instead, our models reduce ETBs since they effectively capture the fact that the corunners make fewer accesses.

We performed the same experiment for the memory model, using msk-ld-40% as *tua* and corunners with 40% and 5% of memory accesses. In this case, the single type of request or multiple types of request models are equivalent, since read and writes to memory have exactly the same impact. The results are analogous to those obtained for the bus model. Therefore, we omitted the figures since they provide no further insights.



(a) contenders-40%



(b) contenders-5%

Figure 10.2: ETB for each bus prediction model: UBD (fully time composable requests); And our approach with 1 and 4 request types.

10.4.2 EEMBCs

As final evaluation we apply the whole prediction model with the EEMBC Autobench suite [143] as reference applications. We run each EEMBC benchmark under a relatively high pressure scenario composed of two tasks, one continuously

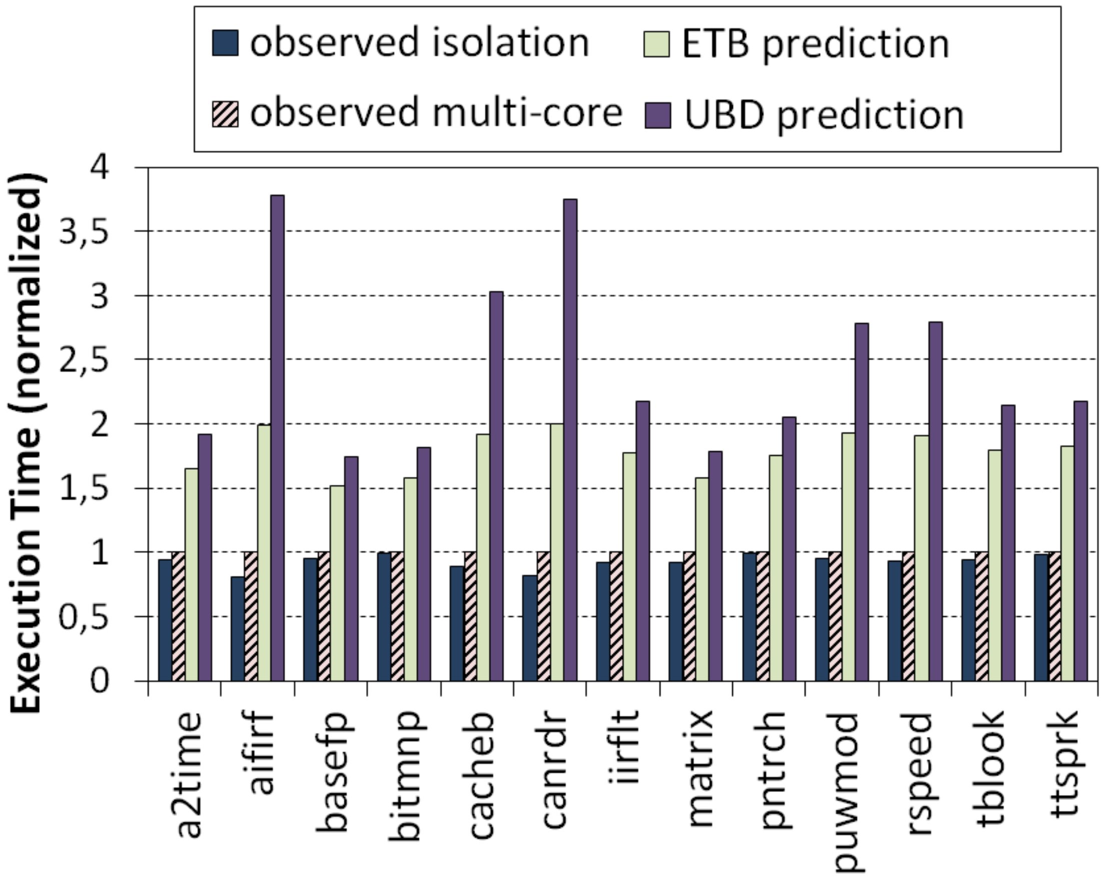


Figure 10.3: ETBs for EEMBC when assuming a no cache misses.

accessing the bus (*bsk*) and the second accessing the memory (*msk*). In this scenario, neither the bus nor the memory controller suffer the highest pressure, since that requires all remaining cores accessing simultaneously each resource [144].

No memory accesses. As presented in Section 10.3.4, there is not a specific PMC to measure the number of accesses to the memory. In order to cancel out the impact of this, in a first experiment we focus on the case in which the *tua* is run twice in a row and measurements are taken during the second run. Due to the small footprint of EEMBC Autobench, this results in almost zero misses in the second run.

Under that assumption, we present the results of our model and UBD in Figure 10.3. Recall that in each workload we run one EEMBC benchmark executed and the two contenders presented above. Results are normalized w.r.t. the execution time of the EEMBC in the workload. We show the execution time in isolation, the execution time in the workload and the predicted ETB using the multiple type of requests prediction model for the bus and memory, as well as the UBD. We can clearly see that our prediction model reduces the pessimism of the UBD model by 67% being only 79% higher than the actual execution time. In Figure 10.2, the observed execution time is much closer to the prediction when compared with Figure 10.3. This is because the scenario in Figure 10.2 is designed to experience severe contention, whereas the scenario described here experiences much lower contention (far below the upper-bounded contention).

General case. Our next step is to evaluate the natural case in which programs perform memory accesses. According to Section 10.3.4, we can estimate the access to the memory using the number of L2 misses. The number of L2 misses does not consider the dirty evictions that generate memory accesses. To take into account the dirty evictions into the memory accesses we can use either an optimistic lower bound based on the number of L2 misses or a pessimistic upper bound based on the number of L2 misses plus the bus writes. To that end, we build three scenarios:

- Pessimistic scenario. We assume that every write operation results in a dirty eviction, i.e. an access to memory.
- Accurate scenario. In this case, from a simulation tool we derive the exact number of memory accesses.
- Optimistic scenario. We disregard the dirty evictions and take the number of L2 misses as memory accesses.

Figure 10.4 shows the results obtained with our model under each of the previous scenarios and the observed execution time in the workload and isolation. These results provide a good estimate of the benefits of improving our reference design with a PMC that explicitly measures memory accesses.

As expected the pessimistic scenario that considers all writes as dirty evictions is overly pessimistic. In particular it is 138% more pessimistic than the actual observed execution time. The accurate scenario in which we assume that the PMC for access count exists leads to very tight estimates, 64% more pessimistic than the actual observed execution time and less than 0.1% more pessimistic than the optimistic scenario. This is due to the small memory footprint of the EEMBC benchmark, that fit on the L2 cache. As a result, the number of dirty evictions is close to zero in most scenarios.

10.5 Related Work

Contention on the access to hardware shared resources has been thoroughly studied in the state of the art. A taxonomic summary of the relevant works can be found

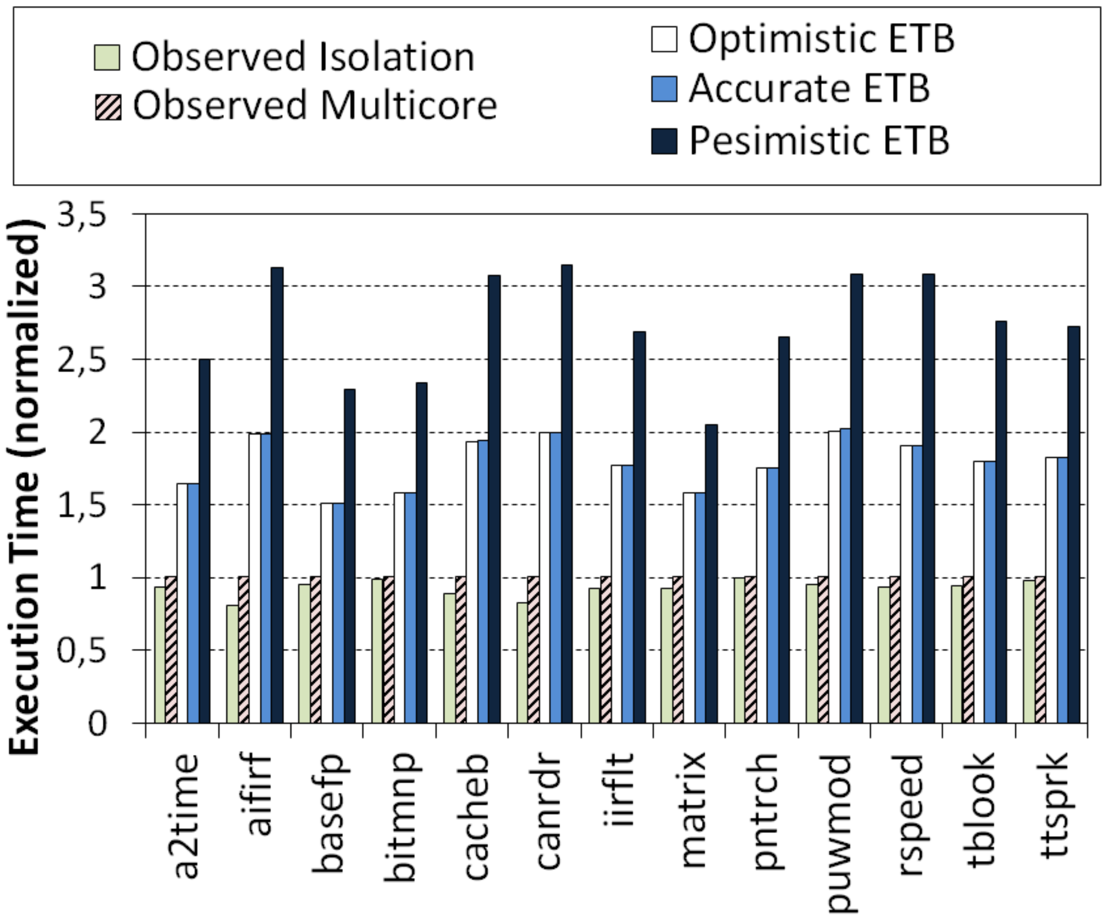


Figure 10.4: ETBs for EEMBC under the optimistic, pessimistic and accurate scenarios.

in [62]. Several techniques propose means to upper-bound, during the analysis phase of the system, the *samd* that a task may suffer on the bus or in memory. In that line, hardware support has been proposed (though not yet implemented in any architecture we are aware of) to artificially delay each request a given τ does by *samd* cycles [138, 93, 159]. Other approaches derive *samd* by using a software-only approach: τ is run against a set of resource stressing kernels that put high load on the resource [65, 63] making τ's requests suffer high contention delays.

Other techniques like those in [151] for buses rely on detailed information about resource access latencies and arbitration policies to derive *samd*. Other works, due to lack of information in the processor documentation derive *samd* from measurements and feed it into static timing analysis. In particular [135] applies this approach to analyze the impact of contention in the P4080. *samd* can also be derived for memory with [140, 7] or without [102] hardware support.

10.6 Summary

In this chapter we present a prediction model of the shared resource contention for the GR740 that takes into account the number of accesses and their type for a given task and its corunner tasks, which can be easily obtained with PMCs. The model abstracts (i.e. makes worst-case provisions) for the way in which requests interleave in time, which would challenge time composability since such time interleaving could easily change during operation.

Derived Execution Time Bounds (ETBs) are shown to be accurate and tighter than fully-time composable ETBs. Those derived estimates are valid for any workload in which the task runs as long as the number of accesses (per type) is smaller than those assumed at analysis. This provides a good balance between tightness and time composability.

Chapter 11

Conclusions and Future Work

11.1 Thesis Conclusions

Critical Real-Time Embedded Systems (CRTES) need to ensure timing correctness in addition to functional correctness. To guarantee timing correctness, timing analysis techniques derive Worst Case Execution Time (WCET) estimates which are provided as input to the corresponding task scheduler, so that it can be assessed that software runs in its assigned budget. *Time predictability*, which stands for the property of being able to bound (in a tight manner) software execution times, is required to derive WCET estimates. Time predictability depends on the complexity of the hardware and the software, and the timing analysis technique (tool) deployed.

Since mid 90's, CRTES have shifted towards an *integrated architecture* paradigm in domains such as avionics: a modular approach in which multiple functions are assigned to a single hardware unit. A key design principle is the *incremental qualification*, whereby each component can be subject to qualification in isolation. At the timing level, this requires *time composability* so that the timing properties of a software component in isolation, i.e. its WCET estimate, do not change when the system is integrated. Time composability therefore reduces the cost of system development, integration and qualification. It is worth noting that this comes at the cost of overestimated WCET.

The space industry, as several other CRTES industries, is assessing the use of multicore processors as their main computing platform to satisfy their growing performance demands. Multicore processors can provide the performance needed for future CRTES and bring the potential of integrating several software functions. However, their use also brings some challenges, from which contention in the access to shared hardware resources is one of the most prominent. Multicore contention generated by a task running on the processor impacts the execution time of the rest of the tasks running simultaneously. This challenges current timing analysis techniques by threatening time predictability and time composability.

In this thesis we have accomplished major improvements to handle shared resource contention in current multicore processors for CRTES improving both time predictability and time composability. The main approach followed in this thesis is to bound or reduce the contention that appears when several cores try to access the same shared resource simultaneously. This bound has been shown to be independent of the rest of the tasks running simultaneously on the system, hence enabling time composability.

We have focused on two of the most important shared resources in a multicore processor: the shared bus and the shared memory. While shared caches challenge significantly the timing analysis of CRTES, partitioning techniques, already implemented in real processor like the NGMP, remove the interference on shared caches. In particular, in this thesis:

- We have proposed and showed the benefits of a time composable bus interface based on AMBA AHB, one of the most broadly used, which provides time composability by design. We have investigated two options: a restricted version of AHB, resAHB, that does not change the interface and an improved version of AHB, AHRB, that requires changes in the interface. Both options make the timing of the bus independent of the specific behaviors of the components of the MPSoC, which greatly facilitates the timing analysis of MPSoCs. Our results show that AHRB offer much tighter WCET estimates than resAHB with negligible average performance degradation.

- We have provided a meaningful comparison between two of the most used bus arbitration policies for real-time on-chip buses, TDMA and IABA. Our results show that both policies offer time predictability and time composability. Overall, IABA presents worse prioritization capabilities but better WCET and average execution time with little burden for the user.

- Regarding the memory controller, we have proposed a memory controller, *DCmc*, capable of consolidating applications that require real-time execution with those that are high-performance driven at the same time. Based on the principle of dividing memory banks into real-time and high-performance, *DCmc* uses a different request scheduler policy to each bank type. *DCmc* provides tight bounds for real-time applications with a reduced impact on high-performance ones.

- We have also studied a new memory organization, PDSC, that builds on the observations that time-sharing memory has poor scalability with the core count and the processor-to-memory frequency ratio. PDSC divides the data bus into narrower independent data buses, thus removing conflicts among different tasks accessing memory, while sharing the command bus. This solution heavily reduces WCET, while providing enough flexibility to handle heterogeneous, i.e. high average and guaranteed performance, requirements of mixed-criticality multicore systems.

- In the context of PTA systems, we have described for the first time a PTA-compliant multicore design. We have proposed low-cost PTA-compliant arbitration policies that break the deterministic behavior of the shared resources in a multicore. Proposed designs fulfill PTA requirements and provide improved guaranteed performance over single-core designs.

We have also provided means to account and predict the contention on the shared resources. In particular:

- We have presented a prediction model for the GR740 that takes into account the effect of contention in shared resources to derive Execution Time Bounds (ETBs). Those ETBs are shown to be accurate and tighter than fully-time composable ETBs.

- We have also proposed some new low-overhead PMCs to measure the actual contention caused by contending tasks. We show how these PMCs provide evidence about the trustworthiness of contention bounds and increase confidence on measurement-based timing analysis, which is a common practice in CRTES industry. It is also of prominent importance to detect the reasons for a deadline miss and helps optimizing system's performance.

In this thesis, we gave an special emphasis on the space domain focusing on the Cobham Gaisler NGMP multicore processor [42], which is currently assessed by the European Space Agency for its future missions. As a proof of concept, the architectural solutions studied and proposed are tested on a space case-study consisting on a cycle-accurate simulation framework validated against a real implementation of the NGMP processor and representative software for the European Space Agency. The results will be used by the European Space Agency to influence the future designs of the NGMP processor. Nevertheless, other domains such as automotive and avionics have similar requirements, and hence our solutions are also applicable.

Overall this thesis paves the way for future works concerning multicore processors in CRTES and eases their adoption by the CRTES industry.

11.2 Impacts

The work done in this thesis has been done in direct contact with industry and also opens several research lines, already pursued by other PhD students in the Universitat Politècnica de Catalunya, that target new challenges in CRTES.

In particular, the bus and memory controllers organizations proposed on this thesis have been presented to the European Space Agency (ESA) as a way to influence the future designs of the NGMP processor. Some of them, are going to be evaluated by the ESA, whose purpose is to obtain Register Transfer Level (RTL) models of such architectural solutions. Implementing the bus architectures proposed in this thesis on real hardware is part of the future work. For instance, in the scope of the PROXIMA FP7 project (www.proxima-project.eu), Probabilistic Timing Analysis (PTA) compliant buses for multicores are going to be tested on a real platform in order to improve the *technology readiness level* of PTA.

The methods presented in this thesis to account and predict contention on shared resources, open new research lines to improve and propose new methods of timing analysis for multicore processors, some of which are already pursued by other PhD students at CAOS group. Also, a set of PMC similar to those proposed in this thesis are planned to be included on the implementation of the NGMP processor by the provider. This open new opportunities to keep improving timing analysis on top of multicore systems.

11.3 Future Work

As the core count in every new processor generation increases, future research lines emmanating from this thesis are:

Processor designs with higher core counts (or many-core processors) usually deploy clustered architectures, which organize processor resources into subsets of processors with fewer core counts. Our solutions can still be applied for such processors and also provide a solid ground to continue research on many-core processors. Of course, such a processor will have new challenges, including other types of interconnection topology or several memory controllers, that should be addressed in order to enable time composability.

In this thesis we have considered partitioned shared caches. This has the drawback of fragmentation since a fixed amount of space is assigned to each core/task in the system. One possible solution is enabling shared caches keeping time composability to some extent. The introduction of shared caches in CRTES poses new challenges that have to be investigated.

In our view, this thesis offers an excellent opportunity for future works to define a time composable Network-on-Chip (NoC) with a well-defined and standardized interface such as AMBA AXI. The NoC can be more complex than a bus, while adding specific restrictions to make it time-composable. The principles applied should be in line with those we define in this thesis.

All along this thesis, we have focused on the case in which a single memory channel is available due to package and pin restrictions. However, multi-channel memory organizations offer a wide new range of opportunities for CRTES, since the contention can be effectively reduced. This remains as a very promising future work.

References

[1] A. Abel, F. Benz, J. Doerfert, B. Drr, S. Hahn, F. Haupenthal, M. Jacobs, A. Moin, J. Reineke, B. Schommer, and R. Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In P. DArgenio and H. Melgratti, editors, *CONCUR 2013 Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 25–43. Springer Berlin Heidelberg, 2013. 24, 168

[2] J. Abella, D. Hardy, I. Puaut, E. Quiñones, and F. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 266–275, July 2014. 3, 4

[3] J. Abella, C. Hernandez, E. Quiñones, F. Cazorla, P. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *the 10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, June 2015. xxi, 1, 5, 15, 16, 18, 148, 151

[4] J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla. Understanding mbpta and its requirements on program instructions. In *http://www.proartis-project.eu/publications/MBPTA-white-paper*, 2013. 15, 124

[5] J. H. Ahn, J. Leverich, R. Schreiber, and N. Jouppi. Multicore dimm: an energy efficient memory module with independently controlled drams. *Computer Architecture Letters*, 8(1):5–8, Jan 2009. 118

[6] B. Akesson and K. Goossens. *Memory Controllers for Real-Time Embedded Systems*. Embedded Systems Series. Springer, 2011. 92

[7] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, USA, 2007. ACM. 22, 23, 76, 91, 105, 112, 187

[8] B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *the 12th Euromicro Conference on Digital System Design (DSD)*, pages 547–555, Aug 2009. 20

[9] S. Altmeyer and R. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014. 17

[10] ARC Advisory Group. Process safety system worldwide outlook. market analysis and forecast through 2012, 2012. 5

[11] ARM. *Architecture Reference Manual. ARMv7-A and ARMv7-R edition.* 167

[12] ARM. *Cortex-A15 Processor.* 126

[13] ARM. *Cortex-R4 processor manual.* 138

[14] ARM. AMBA specification (rev. 2), 1999. 26, 43, 158

[15] ARM. The ARM Cortex-A9 processors (white paper), 2009. 59, 167

[16] ARM. *Cortex-R5. Technical Reference Manual. Revision: r1p2*, 2011. 151

[17] P. Atanassov and P. Puschner. Impact of DRAM refresh on the execution time of real-time tasks. In *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34, Dec. 2001. 21

[18] AUTOSAR. *Technical Overview V2.0.1*, 2006. 6

[19] C. Ballabriga, H. Cass, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In S. Min, R. Pettit, P. Puschner, and T. Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin Heidelberg, 2010. 16

[20] B. Bhat and F. Mueller. Making DRAM refresh predictable. *Real-Time Syst.*, 47(5):430–453, Sept. 2011. 21

[21] J. Bin, S. Girbal, D. Gracia, and A. Merigot. Using monitors to predict co-running safety-critical har real-time benchmark behavior. In *5th Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-5)*, 2014. 19

[22] V. Bos. EagleEye: Evolution towards Time and Space Partitioning. In *Software & Data Systems Division Final Presentation Days, ESA 2013*, 2013. 101

[23] V. Bos, P. Mendham, P. Kauppinen, N. Holsti, A. Crespo, M. Masmano, J. A. de la Puente, and J. Zamorano. Time and Space Partitioning the EagleEye Reference Mission. In *Data Systems in Aerospace*, DASIA, 2013. 29

[24] S. Boslaugh and P. A. Watters. *Statistics in a nutshell.* O'Reilly Media, Inc., 2008. 140

[25] R. Bourgade, C. Rochange, M. De Michiel, and P. Sainrat. Mbba: A multi-bandwidth bus arbiter for hard real-time. In *Embedded and Multimedia Computing (EMC), 2010*, pages 1–7, 2010. 20, 82

[26] R. Bourgade, C. Rochange, and P. Sainrat. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4, Sept 2011. 20

[27] J. Bradley. *Distribution-Free Statistical Tests.* Prentice-Hall, 1968. 39, 140

[28] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 269–279, July 2007. 3

[29] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. *RTCSA*, Aug. 2008. 19

[30] D. Buttle. Etas gmbh, germany, real-time in the prime-time. keynote talk. In *the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2012. 5

[31] F. J. Cazorla, R. Gioiosa, M. Fernandez, and E. Q. nones. Multicore OS benchmarks. Technical Report Contract 4000102623, European Space Agency, 2012. 7, 11, 29

[32] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. PROARTIS: Probabilistically analyzable real-time systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):94:1–94:26, May 2013. 4, 15, 17, 121, 123, 124, 125, 127, 145

[33] S. Chattopadhyay, C. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 99–108, April 2012. 18, 19, 168

[34] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *DAC*, Los Angeles, CA, USA, June 2000. 19

[35] P. Clarke. Automotive chip content growing fast, says gartner. In *http://www.eetimes.com/electronics-news/4207377/Automotive-chip-content-growing-fast*, 2011. 5

[36] Cobham Gaisler. *Dual-Core LEON3-FT SPARC V8 Processor - GR712RC-UM - Data Sheet and Users Manual Draft, 2014*. 20, 41

[37] Cobham Gaisler. *GRMON2-User Manual Version 2.0.62, March 2015*. 27, 34, 164, 180, 181

[38] Cobham Gaisler. *Leon3 Processor*. `http://www.gaisler.com/`. 9, 138

[39] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - GR740-UM-DS-D1 - Data Sheet and Users Manual, 2015*. 25, 27, 33, 155, 156, 158, 160, 161, 181

[40] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011. 24, 110, 121

[41] Cobham Gaisler. *LEON4-N2X Data Sheet and User's Manual*, 2013. 8, 12, 27, 173

[42] Cobham Gaisler. *NGMP Preliminary Datasheet Version 2.1, May 2013*, 2013. xvii, 10, 19, 20, 25, 26, 34, 36, 41, 59, 68, 69, 75, 96, 110, 112, 180, 191

[43] Cobham Gaisler. *Leon4 Product sheet*. `http://www.gaisler.com/`, 2015. xvii, 27

[44] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-N2X Data Sheet and Users Manual Version 2.1*, 2015. 9, 34

[45] M. Conti, M. Caldari, G. Vece, S. Orcioni, and C. Turchetti. Performance analysis of different arbitration algorithms of the AMBA AHB bus. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 618–621, 2004. 20

[46] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, July 2012. 4, 17, 18, 39, 121, 124, 125, 126, 127, 140, 141, 145

[47] D. Dasari and V. Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *HPCC-ICESS*, pages 1450–1457, 2012. 72

[48] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011. 3

[49] M. Di Natale and A. Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, 98(4):603–620, April 2010. 5, 6

[50] J. Diemer, D. Thiele, and R. Ernst. Formal worst-case timing analysis of ethernet topologies with strict-priority and avb switching. In *the 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, June 2012. 19

[51] G. Edelin. Embedded systems at THALES: the Artemis challenges for an industrial group. In *ARTIST*, 2009. 5

[52] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *the* 22^{nd} *IEEE Real-Time Systems Symposium (RTSS)*, 2001. 4

[53] J. Elmqvist, S. Nadjm-Tehrani, K. Forsberg, and S. Nordenbro. Demonstration of a formal method for incremental qualification of IMA systems. In *Digital Avionics Systems Conference (DASC)*, 2008. 6

[54] ESA. Ngmp contract: 22279/09/nl/jk. 68

[55] European Cooperation for Space Standarization. *ECSS-E-ST-50-12C*, 2008. 19

[56] European Cooperation for Space Standarization. *ECSS-E-ST-40C*, 2009. 2

[57] European Cooperation for Space Standarization. *ECSS-Q-ST-40C*, 2009. 2

[58] European Cooperation for Space Standarization. *ECSS-Q-ST-80C*, 2009. xxi, 8, 9

[59] W. Feller. *An introduction to Probability Theory and Its Applications*. John Willer and Sons, 1996. 39

[60] C. Ferdinand. Worst case execution time prediction by static program analysis. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 125–, April 2004. 16

[61] G. Fernandez, J. Abella, E. Quinones, L. Fossati, M. Zulianello, T. Vardanega, and F. Cazorla. Seeking time-composable partitions of tasks for cots multicore processors. In *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*, pages 208–217, April 2015. 19, 172

[62] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. 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*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 31–42, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 18, 186

[63] G. Fernandez, J. Jalle, J. Abella, E. Q. nones, T. Vardanega, and F. Cazorla. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *The 52st Annual Design Automation Conference 2015, DAC '15, San Francisco, CA, USA, June 7-11, 2015*, 2015. 19, 24, 33, 36, 148, 151, 178, 179, 186

[64] G. Fernandez, J. Jalle, J. Abella, E. Q. nones, T. Vardanega, and F. Cazorla. Resource usage templates and signatures for COTS multicore processors. In *The 52st Annual Design Automation Conference 2015, DAC '15, San Francisco, CA, USA, June 7-11, 2015*, 2015. 19, 24, 174

[65] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, New York, NY, USA, 2012. ACM. 18, 24, 151, 168, 171, 172, 182, 186

[66] FlexRay Consortium. Flexray communications system protocol specification, 2005. 19

[67] D. Flynn. AMBA: enabling reusable on-chip designs. *Micro, IEEE*, 17(4):20–27, 1997. 20, 41

[68] Freescale. *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*. 167

[69] Freescale. *On-Chip Debugging of Multicore Systems*. 167

[70] Freescale. *P4080 Microprocessor*. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080. 8, 10, 11

[71] FreeScale. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. Rev 1*, 2012. 151

[72] FreeScale. *e500mc Core Reference Manual. Rev 3*, 2013. 16, 151, 167

[73] M. D. Gomony, B. Akesson, and K. Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1307–1312, 2013. 109, 113, 118

[74] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 525–530, March 2013. 22, 23, 105

[75] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013. 22, 23, 105

[76] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. ARM white paper, 2011. 167

[77] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 2009. 18

[78] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *the 30th IEEE Real-Time Systems Symposium(RTSS)*, pages 68–77, Dec 2009. 18

[79] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003. 6

[80] G. Heiser. The role of virtualization in embedded systems. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16, Glasgow, UK, Apr. 2008. ACM SIGOPS. 7

[81] C. Hernandez, J. Abella, F. J. Cazorla, J. Andersson, and A. Gianarro. Towards making a leon3 multicore compatible with probabilistic timing analysis. In *20th Data Systems In Aerospace Conference, DASIA*, 2015. 18

[82] D. Hibler. Considerations for designing an embedded intel architecture system with system memory down (white paper), 2009. 109

[83] R. Hohn, W. Ruehe, and C. Jewell. The Cryogenic System of the Herschel Extended Payload Module. In *AIP Conference Proceedings*, 2004. 8

[84] Infineon. AURIX Safety joins Performance. 8, 10, 11, 59, 68, 69, 75

[85] Infineon. *XMC4500 Microcontroller Series for Industrial Applications Reference Manual. Rev 1*, 2012. 151

[86] L. Innocenti. User cases: Active debris removal. In *ADCSS, ESA*, 2013. 9

[87] Intel. SCC external architecture specification (EAS). revision 0.94. 126

[88] Intel. Intel quark SoC x1000 platform: DDR3 dual rank memory down board layout guide (white paper), 2014. 109

[89] S. International. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. 25

[90] International Organization for Standardization. *ISO/DIS 11898. Road vehicles – Controller area network*, 1993. 19

[91] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009. 2

[92] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008. 10, 20, 22, 135

[93] J. Jalle, J. Abella, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla. Deconstructing bus access control policies for real-time multicores. In *the 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013. 96, 186

[94] J. Jalle, J. Abella, E. Quinones, L. Fossati, M. Zulianello, and F. Cazorla. Ahrb: A high-performance time-composable amba ahb bus. In *the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 225–236, April 2014. 24, 156, 159

[95] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *the 35th IEEE Real-Time Systems Symposium (RTSS)*, pages 207–217, Dec 2014. 22, 90, 91

[96] JEDEC. *DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008. 22, 27

[97] A. Jung and P.-E. Crouzet. The H2RG infrared detector: introduction and results of data processing on different platforms. Technical report, European Space Agency, 2012. 8, 28

[98] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. POWER7: IBM's next-generation server processor. In *IEEE Micro*, 2010. 23, 126, 166, 168

[99] Kalray. *MPPA 256 Many-Core Processor*, 2013. 114

[100] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era. In *MICRO-44*, 2011. 104

[101] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2011. 20, 68, 70, 76

[102] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014. 19, 23, 88, 91, 93, 96, 98, 100, 105, 187

[103] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, Paris, France, 2013. 10, 19

[104] Kingston. *KVR667D2S5/2G Datasheet*, 2011. 30, 99, 161

[105] D. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium (RTSS)*, 1989. 10, 19

[106] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 333–339, May 2008. 18

[107] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003. 18, 70

[108] H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524–533, 1993. 19

[109] L. Kosmidis, J. Abella, E. Quiñones, and F. Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *the 34th IEEE Real-Time Systems Symposium (RTSS)*, pages 360–371, Dec 2013. 18

[110] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 513–518, March 2013. xviii, 4, 18, 121, 124, 125, 128, 145

[111] L. Kosmidis, J. Abella, F. Wartel, E. Quiñones, A. Colin, and F. Cazorla. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 276–287, July 2014. 18

[112] L. Kosmidis, C. Curtsinger, E. Quiñones, J. Abella, E. Berger, and F. J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 603–606, March 2013. 18

[113] L. Kosmidis, E. Quiñones, J. Abella, G. Farrall, F. Wartel, and F. J. Cazorla. Containing timing-related certification cost in automotive systems deploying complex hardware. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 22:1–22:6, New York, NY, USA, 2014. ACM. 18

[114] L. Kosmidis, T. Vardanega, J. Abella, E. Q. nones, and F. Cazorla. Applying measurement-based probabilistic timing analysis to buffer resources. In *WCET workshop*, 2013. 18

[115] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000. 18, 124

[116] C. M. Krishna. *Real-Time Systems*. McGraw-Hill Higher Education, 1st edition, 1996. 1

[117] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. A rank-switching, open-row dram controller for time-predictable systems. In *the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 27–38, July 2014. 91, 105, 107, 109, 113, 118

[118] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs. In *Design Automation Conference (DAC)*, DAC '01, pages 15–20, 2001. 122, 129

[119] E. A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009. 2

[120] A. F. Leon. ASIC FPGA rad effects. In *ESTEC TEC-E Plenary meeting*, 2014. 9

[121] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Syst.*, 48(6):638–680, Nov. 2012. 18

[122] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *the 4th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1997. 10, 19

[123] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee. A pret microarchitecture implementation with repeatable timing and competitive performance. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 87–93, Sept 2012. 18

[124] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 367–376, New York, NY, USA, 2012. ACM. 96, 113

[125] Q. Liu, M. Moreto, V. Jimenez, J. Abella, F. J. Cazorla, and M. Valero. Hardware support for accurate per-task energy metering in multicore systems. *ACM Trans. Archit. Code Optim.*, 10(4):34:1–34:27, Dec. 2013. 23, 168

[126] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1999. 6, 181

[127] M. Masmano, A. Crespo, and J. Coronel. XtratuM hypervisor for LEON4. Technical report, European Space Agency, 2012. 101

[128] E. Mezzetti and T. Vardanega. On the industrial fitness of WCET analysis. *11th International Workshop on WCET Analysis*, 2011. 18, 127

[129] S. Milutinovic, J. Abella, D. Hardy, E. Quiones, I. Puaut, and F. Cazorla. Speeding up static probabilistic timing analysis. In L. M. P. Pinho, W. Karl, A. Cohen, and U. Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015*, volume 9017 of *Lecture Notes in Computer Science*, pages 236–247. Springer International Publishing, 2015. 17

[130] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007. 93

[131] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1995. 10, 19

[132] E. Musol. Hardware-based load balancing for massive multicore architectures implementing power gating. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, volume 29, 2010. 126

[133] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA*, 2008. 104, 105

[134] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012. 18, 24, 168

[135] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 109–118, July 2014. 16, 19, 24, 148, 151, 168, 186

[136] J. Owens. Delphi automotive, the design of innovation that drives tomorrow. Keynote talk. In *Design Automation Conference (DAC), 2015*, July 2015. 5

[137] M. Panic et al. Parallel many-core avionics systems. In *EMSOFT*, 2014. 114

[138] M. Paolieri, E. Q. nones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009. 10, 18, 19, 20, 45, 51, 59, 63, 67, 68, 72, 77, 82, 96, 99, 114, 117, 133, 151, 162, 175, 186

[139] M. Paolieri, E. Q. nones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letters (ESL)*, 2009. 11, 23, 45, 59, 76, 87, 91, 92, 96, 105, 135

[140] M. Paolieri, E. Quiñones, and F. 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. 22, 23, 63, 90, 93, 109, 112, 114, 187

[141] Parasoft. Iso 26262 software compliance: Achieving functional safety in the automotive industry. white-paper., 2011. 7

[142] M. Patte and V. Lefftz. System impact of distributed multi core systems. Technical Report Contract 4200023100, European Space Agency, 2011. 7, 8, 87, 110, 114

[143] J. Poovey. *Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007. 27, 28, 38, 59, 76, 99, 114, 137, 161, 183

[144] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 2012. 24, 29, 151, 163, 168, 184

[145] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium*, 1988. 52

[146] Rapita Systems. Rapitime. http://www.RapitaSystems.com/RapiTime. 17, 39, 77

[147] Rapita systems Ltd. Rapita Verification Suite. `http://www.rapitasystems.com/products/rvs`. 5, 17, 147, 172

[148] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011. 18, 22, 23, 90, 105, 109, 114

[149] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *WCET*, 2006. 171, 181

[150] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM. 23, 88, 94, 96, 100, 104, 105

[151] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *the 28th IEEE Real-Time Systems Symposium(RTSS*, 2007. 18, 20, 68, 70, 186

[152] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*, 1992. 2

[153] RTCA and EUROCAE. DO-178C, software considerations in airborne systems and equipment certification, 2011. 147

[154] SAE International. ARP4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment., 2001. 141

[155] SAE International. ARP4754A: Guidelines for development of civil aircraft and systems., 2011. 7

[156] E. Salminen, T. Kangas, V. Lahtinen, J. Riihimäki, K. Kuusilinna, and T. D. Hämäläinen. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *Journal of Systems Architecture*, 2007. 10, 43

[157] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 759–764, March 2010. 19

[158] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. In *the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 215–224, April 2010. 19, 171

[159] H. Shah, A. Coombes, A. Raabe, K. Huang, and A. Knoll. Measurement based wcet analysis for multi-core architectures. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 257:257–257:266, New York, NY, USA, 2014. ACM. 18, 20, 23, 168, 186

[160] H. Shah, A. Raabe, and A. Knoll. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2011. 20

[161] Silabs. *http://www.silabs.com/Marcom%20Documents/Resources/automotive-applications-guide.pdf*, 2013. 5

[162] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. Cazorla. DTM: Degraded test mode for fault-aware probabilistic timing analysis. In *the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 237–248, July 2013. 18

[163] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Time-analysable non-partitioned shared caches for real-time multicore systems. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 198:1–198:6, New York, NY, USA, 2014. ACM. 18, 19, 168

[164] SoCLib. http://www.soclib.fr/trac/dev, 2003-2012. 29

[165] J. Staschulat and M. Bekooij. Dataflow models for shared memory access latency analysis. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, 2009. 76

[166] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *HPCA*, 2010. 10, 43

[167] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. Zaykov, Z. Petrov, B. Boddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Casse, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – multi-core execution of parallelised hard real-time applications supporting analysability. In *the 16th Euromicro Conference on Digital System Design (DSD)*, pages 363–370, 2013. 7

[168] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *SIGMETRICS*, 2003. 3

[169] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *the 28th IEEE Real-Time Systems Symposium(RTSS*, 2007. 7, 87

[170] E. Wandeler and L. Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *ASP-DAC*, 2006. 70

[171] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005. 30, 161

[172] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013. 10, 19

[173] F. Ware et al. Improving power and data efficiency with threaded memory modules. In *ICCD*, 2006. 118

[174] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Quiñones, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-Grosjean, T. Vardanega, and F. J. Cazorla. Timing analysis of an avionics case study on complex hardware/software platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, DATE '15, pages 397–402, San Jose, CA, USA, 2015. EDA Consortium. 17

[175] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *the 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 241–248, June 2013. 4, 17

[176] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference (DASC)*, 2007. 6

[177] I. Wenzel, R. Kirner, B. Rieder, and P. P. Puschner. Measurement-based timing analysis. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, pages 430–444, 2008. 17

[178] A. West. NASA study on flight software complexity. Final report. Technical report, NASA, 2009. 5, 9, 147

[179] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm. Static timing analysis for hard real-time systems. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 3–22, Berlin, Heidelberg, 2010. Springer-Verlag. 16

[180] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008. xvii, 3, 4, 15, 16, 45, 147, 172

[181] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, July 2009. 181

[182] J. Windsor, M.-H. Deredempt, and R. De-Ferluc. Integrated modular avionics for spacecraft - user requirements, architecture and role definition. In *Digital Avionics Systems Conference (DASC)*, 2011. 6

[183] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *the 34th IEEE Real-Time Systems Symposium (RTSS)*, 2013. 11, 23, 87, 94, 97, 105, 109, 112, 114, 118

[184] J. Yan and W. Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 80–89, April 2008. 18

[185] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 295–306, June 2011. 118

[186] M.-K. Yoon, J.-E. Kim, and L. Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 227–238, Nov 2011. 68

[187] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014. 96

[188] W. Zhang and J. Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 455–463, Aug 2009. 18

[189] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 339–352, Sept 2007. 23, 168

[190] M. Ziccardi, E. Mezzetti, T. Vardanega, J. Abella, and F. J. Cazorla. Epc: Extended path coverage for measurement-based probabilistic timing analysis. In *the 36nd IEEE Real-Time Systems Symposium (RTSS)*, 2015. 18

Glossary

ACD Actual Contention Delay

AHB Advanced High-performance Bus

AHRB Advanced High-performance Real-time Bus

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

ASB Advanced System Bus

AUTOSAR Open System Architecture

AXI Advanced eXtensible Interface

BID Bus Inter-task Delay

BLP Bank Level Parallelism

BSC Barcelona Supercomputing Center

CCS Contention Cycle Stack

CDB Contention Delay Bound

COTS commercial-off-the-self

CPI Cycles Per Instruction

CRTES Critical Real-Time Embedded Systems

CS Chip Select

DCmc Dual-Criticality memory controller

DQS Data Query Strobe

DTA Deterministic Timing Analysis

ESA European Space Agency

ETB Execution Time Bound

ETP Execution Time Profile

EVT Extreme-Value Theory

FR-FCFS First Ready - First Come First Served

GNC Guidance and Navigation Control

HYDTA Hybrid Deterministic Timing Analysis

IABA Interference-Aware Bus Arbiter

IMA Integrated Modular Avionics

IP Intellectual Property

IPC Instructions Per Cycle

ISA Instruction Set Architecture

MBDTA Measurement-based Timing Analysis

MBPTA Measurement-Based Probabilistic Timig Analysis

MLP Memory Level Parallelism

MMS Memory Mapping Scheme

MpKI Memory accesses per Kilo Instruction

MPSoC Multiprocessor System-on-Chip

NGMP Next Generation Microprocessor

NoC Network-on-Chip

PDSC Private-Data bus Shared-Command bus

PMC Performance Monitoring Counters

PTA Probabilistic Timing Analysis

pWCET probabilistic WCET

RBL Row Buffer Locality

resAHB restricted AHB

RSK Resource-Stressing Kernels

RTES Real-time Embedded Systems

RTL Register Transfer Level

samd single-access maximum (contention) delay

SDTA Static Deterministic Timing Analysis

SPTA Static Probabilistic Timing Analysis

TDMA Time-Division Multiple Access

TUA Task Under Analysis

UBD Upper-Bound Delay

WCD Worst Contention Delay

WCET Worst-Case Execuion Time